МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ

НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Физический факультет
Кафедра физико-технической информатики физического факультета

Иртегов Д.В

Системные вызовы и библиотеки Unix System V Release 4

Учебное пособие

Новосибирск
2013

Иртегов Д.В. Системные вызовы и библиотеки Unix System V Release 4. Учеб. пособие / Новосиб. гос. ун-т. Новосибирск, 2013. – 449 с.

Учебное пособие представляет изложение материала по теме «Системные вызовы ОС Unix», соответствующего программе одноименнго спецкурса кафедры ФТИ ФФ НГУ. Пособие состоит из десяти разделов (лекций). Пособие содержит общее описание ОС Unix и базовые сведения о системном программировании. Описываются: понятие процесса, создание процессов, управление ими; понятие файла, работа с файлами (ввод, вывод, произвольный доступ, отображение на память, терминальный интерфейс), управление файлами и каталогами (создание, удалене, управление правами, просмотр атрибутов, создание жестких и символических связей, просмотр содержимого каталога), сигналы и средства межпроцессного взаимодействия. Пособие может использоваться как краткий справочник по системному программированию для ОС семейства Unix с использованием POSIX (IEEE 1003) API.

Также пособие содержит сведения об истории семейства Unix (AT&T Unix, BSD Unix, Unix System V, Linux, OS X, Android и др.), а также об особенностях реализации POSIX API в ОС Oracle Solaris 10/11.

Пособие предназначено для студентов третьего курса кафедры ФТИ физического факультета и для студентов второго курса факультета информационных технологий.

Учебное пособие подготовлено в рамках реализации
Программы развития НИУ-НГУ на 2009-2018 г.г.

Оглавление

Введение

Системные вызовы и библиотеки Unix

Граница между системным и прикладным программированием достаточно условна. Любой программист так или иначе вынужден пользоваться сервисами операционной системы. Эти сервисы могут быть скрыты от программиста средой исполнения языка высокого уровня или различными «кроссплатформенными» библиотеками и фреймворками. Однако абстракции, представляемые средой исполнения и библиотеками неидеальны, они «протекают». Главные направления таких протечек — это, в первую очередь, производительность и процесс отладки.

Когда программа работает не так, как запланировано — это может происходить как из-за ошибки в библиотеке, так и, чаще, из-за ошибочного использования библиотеки — в процессе отладки часто приходится спускаться на низкий уровень и анализировать происходящее не в терминах объектов и шаблонов ЯВУ, а в терминах указателей и системных вызовов. Поэтому знакомство с системным программированием может оказаться полезным и для тех, кто собственно системным программированием заниматься не планирует.

С точки зрения программиста на относительно низкоуровневом языке, таком, как C, сервисы ОС выглядят как набор функций, предоставляемых системными библиотеками, и собственно системные вызовы.

Системные вызовы — это обращения к ядру операционной системы. В ОС, поддерживающих виртуальную память, пользовательский код не может самостоятельно исполнять ряд операций, в том числе — выполнять операции ввода-вывода, обращаться к структурам данных ОС и к памяти других процессов. Системный вызов включает в себя переход в системный (привилегированный) режим работы процессора, передачу управления ядру и копирование параметров в память ядра. Ядро проверяет наличие прав на исполнение запрошенной операции и, если права есть, исполняет саму операцию.

Главные функции, выполняемые системными вызовами, включают в себя ввод-вывод (в том числе доступ к файловой системе и к сети), межпроцессное взаимодействие и доступ к настройкам системы и глобальным данным, например, к показаниям системных часов.

Функции стандартной системной библиотеки исполняются в пользовательском режиме работы процессора. Некоторые из функций могут содержать внутри себя один или несколько системных вызовов, некоторые другие полностью работают в пользовательском режиме.

Задачи функций библиотеки отличаются большим разнообразием. Некоторые из функций, такие, как буферизованный ввод-вывод (известный также как библиотека стандартного ввода-вывода) пытаются сократить количество системных вызовов. Системные вызовы — относительно более дорогая операция, чем простой вызов функции, поэтому, например, считывать данные из файла или с терминала по одному символу считается нежелательным. Библиотека предоставляет буфер в пользовательской памяти, к которому программа может обращаться удобным для нее образом, например, считывая по одному символу или по строкам. Когда буфер опустеет или, наоборот, заполнится, библиотека исполняет системный вызов и считывает или записывает новую порцию данных.

Другие библиотечные функции предоставляются для удобства или для совместимости со старыми версиями ОС или для обеспечения соответствия стандартам.

Системы семейства Unix

Границы семейства Unix точно не определены. В это семейство входят операционные системы, архитектура и/или доступные пользователю сервисы которых, в основном, унаследованы от оригинальной ОС Unix, разработанной Д.Томпсоном и К.Ритчи в начале 1970х в Bell Laboratories (в то время — исследовательское подразделение компании AT&T).

В первом приближении, ОС семейства Unix можно разбить на следующие группы:

1. Системы, наследующие авторские права на код и архитектурные решения оригинального Unix. Из поддерживаемых на 2012 год, это ряд систем, основанных на Unix Sysvem V Release 3, в первую очередь IBM AIX и HP HP/UX, а также системы, основанные на Unix System V Release 4, в первую очередь, Oracle Solaris.
2. Системы, разработанные без использования кода, авторские права на который принадлежат или принадлежали AT&T, но, в основном, воспроизводящие архитектуру традиционного Unix. Это ветви BSD Unix (FreeBSD, OpenBSD, NetBSD), Minix, Linux. BSD Unix первоначально использовал код AT&T Unix v6/7, опубликованный на условиях public domain, но в начале 1990х ветви BSD были переписаны, чтобы избавиться от соответствующего кода и претензий к нарушению авторских прав. Minix и Linux никогда не содержали кода AT&T
3. ОС специального назначения с оригинальной (чаще всего, микроядерной) архитектурой, при разработке которых ставилась цель обеспечить определенную степень совместимость с Unix, главным образом, для облегчения переноса средств разработки (компиляторов, отладчиков и др.) и сетевых средств. К этой категории следует отнести и Apple OS X и Apple iOS. Однако, наиболее распространенные ОС из этой категории — это ОС реального времени, такие, как QNX и VxWorks. По-видимому, эти ОС долгое время наиболее распространенными ОС семейства Unix, так как они широко используются в массовых встраиваемых устройствах (автомобильных компьютерах, контроллерах медицинского и бытового оборудования и т. д.). Возможно, к 2012 году, из-за распространения портативных устройств под управлением Android и других встраиваемых и специализированных компьютеров под Linux (например, сетевых маршрутизаторов), положение уже изменилось и на данный момент лидером является Linux. Точно определить численность специализированных компьютеров под управлением конкретной ОС затруднительно, так как сводной статистики по этому вопросу в открытом доступе нет.
4. Строго говоря, эти ОС не следует считать принадлежащими к семейству Unix, но в некоторых обзорах их также включают в это семейство. Это ОС оригинальной архитектуры, развивавшиеся независимо от Unix, но в настоящее время, обеспечивающие достаточную степень совместимости со стандартами POSIX и x/Open. Некоторые из этих ОС оказались достаточно совместимы, чтобы пройти сертификацию x/Openи получить право на использование торговой марки UNIX. Примерами таких ОС являются IBM OS/390, IBM z/OS, HP OpenVMS.

Более подробное описание истории семейства Unix и различий между основными ветвями семейства приведено в приложении «История Unix».

Различия между разными системами семейства Unix, как на уровне внутренней организации, так и на уровне внешних интерфейсов, конечно же, существуют, так что нельзя сказать, что, изучив одну ОС, вы изучили их все. Однако существуют стандарты, которым, в той или иной мере, пытаются поддерживать все ОС семейства. Знание этих стандартов позволяет разрабатывать переносимое программное обеспечение и значительно облегчает изучение конкретных ОС.

Стандарты

За время существования семейства Unix был предпринят ряд попыток стандартизовать системные интерфейсы, чтобы облегчить перенос программного обеспечения между разными системами семейства. В настоящее время, наиболее влиятельный из стандартов — это Single Unix Specification, документ, принятый и поддерживаемый консорциумом Open Group (http://www.opengroup.org). Этот документ имеет также статус стандарта IEEE Std 1003 (POSIX) и ISO/IEC 9945.

Стандарт SUS/POSIX специфицирует API (Application Programming Interface), то есть интерфейс, доступный прикладному программисту. Этот стандарт обеспечивает совместимость на уровне исходного кода на языках C/C++. Адаптация программы к другой реализации стандарта может потребовать (и, обычно, требует) перекомпиляции. Так, API определяет минимальный набор полей в структурах данных, но не указывает, может ли структура содержать другие поля и в каком порядке эти поля должны быть размещены в структуре; API допускает реализацию функции как в виде, собственно, функции языка C, так и в виде препроцессорного макроса.

Практика показала, что стандарт SUS/POSIX обеспечивает достаточно большую свободу в выборе методов реализации, так что даже многие ОС, не входящие в семейство Unix, например, IBM z/OS или HP OpenVMS, декларируют ту или иную степень совместимости с POSIX.

Кроме API, существует более низкий уровень спецификации системных интерфейсов, называемый ABI (Application Binary Interface). ABI включает в себя:

- спецификацию системы команд
- «соглашение о вызовах», то есть способ передачи параметров функциям, способ передачи кода возврата функции, формат стекового кадра и списков регистров, которые обязаны сохраняться при вызовах
- формат исполняемых файлов и разделяемых библиотек
- структуру виртуального адресного пространства
- способ исполнения системных вызовов и передачи параметров
- номера системных вызовов и списки параметров каждого вызова
- формат структур данных, передаваемых библиотечным функциям и системным вызовам в качестве параметров. В данном случае, формат означает не только точные списки полей структур, но и точные размеры и размещение полей и размер самих структур с точностью до байта
- численные значения различных параметров, например, размещение битовых флагов или номера кодов ошибок

ОС, предоставляющие совместимые ABI, не требуют перекомпиляции пользовательских программ, то есть обеспечивают бинарную совместимость. Обычно, бинарная совместимость обеспечивается в пределах одной линии ОС: новые версии бинарно совместимы со старыми. ОС Unix System V, в том числе Solaris, поддерживают стандарт iBCS (Intel Binary Compatibility Standart).

В этом курсе мы будем изучать стандарт SUS/POSIX на примере его реализации Unix System V Release 4, например, Solaris.

Системные вызовы

Системные вызовы - это интерфейс между пользовательским процессом и операционной системой. Существуют системные вызовы для:

. Ввода/вывода: например, read, write и ioct

. Управления файлами: например, chmod, unlink и mknod

. Доступа к системным данным: например, getuid и getpid

. Управления процессами и их синхронизации: например, signal, wait и semop

. Управления памятью: например, brk, sbrk и mmap

В Unix, все системные вызовы имеют функции-«обертки» в стандартной библиотеке языка C, так что системные вызовы можно вызывать как обычные функции языка C. Многие из системных вызовов принимают аргументы; также, многие вызовы возвращают значение. Обычно, возвращаемое значение -1 сигнализирует об ошибке. В этом случае, системный вызов (точнее, его функция-обертка) размещает код ошибки в переменной errno (в многопоточных программах, в действительности, это не переменная, а препроцессорный макрос, но этот макрос является lvalue, т. е. ему можно присваивать значения и получать указатель на него). Во многих старых учебниках рекомендуют описывать errno как extern int errno, но для совместимости с многопоточными программами рекомендуется использовать определение, предоставленное в заголовочном файле <errno.h>.

Переменная errno устанавливается при ошибках; в соответствии со стандартом POSIX, системный вызов имеет право изменять errno при успешном завершении, поэтому при успешном вызове значение errno следует считать неопределенным. По этой же причине, не следует ориентироваться на значение errno для проверки успешности системного вызова.

Многие библиотечные функции, содержащие внутри системный вызов, также устанавливают errno.

Для всех допустимых значений errno в заголовочном файле <errno.h> определены символьные константы, которые соответствуют мнемоническим именам ошибок. В разных Unix-системах номера ошибок могут различаться, поэтому, при анализе значения errno следует использовать эти константы, а не численные значения. Это значительно облегчит адаптацию вашей программы к другим ОС.

Организация системного руководства

Системное руководство Unix доступно из командной строки через команду man(1). При каждом вызове, команда man выдает содержимое указанной страницы руководства, например, команда man man выдает руководство по самой команде man. Существуют также графические программы для просмотра системного руководства, например, xman(1). В системном руководстве описаны все системные вызовы, все функции стандартной библиотеки и других библиотек, входящих в поставку системы, все команды, доступные из командной строки и ряд другой информации.

Руководство разделено на пять секций:

1 Команды, доступные из командной оболочки shell (подсекция 1M — команды, требующие административных привилегий)

2 Системные вызовы.

3 Функции и библиотеки. Этот раздел имеет ряд подсекций, в том числе:

3C Библиотечные функции, реализованные на Си или ассемблере, составляющие стандартную библиотеку языка Си. Эти функции содержатся либо в /usr/libc.so (для разделяемых библиотек), либо в /usr/lib/libc.a (для архивных библиотек). В Solaris 9, архивной версии библиотеки libc.a не предоставляется. При компиляции программы на языке C, редактор связей автоматически подключает одну из этих библиотек.

3G Библиотечные функции общего назначения. Необходима опция -lgen в командной строке cc для поиска в библиотеке /usr/lib/libgen.so.

3M Математические библиотечные функции, образующие математическую библиотеку. Необходима опция -lm в командной строке компилятора. Объявления этих функций могут быть получены из <math.h>.

3X Специализированные библиотечные функции. Распределены по нескольким библиотекам. Прочитайте соответствующую страницу Руководства, чтобы определить библиотеку, которая должна быть задана.

4 Форматы файлов, описывает форматы системных файлов. Например, passwd(4) описывает формат файла /etc/passwd, в котором хранится БД учетных записей.

5 Остальные средства. Например, карта символов ASCII на странице ASCII(5), полезные сведения о системном вызове fcntl(2) на FCNTL(5), полезные сведения о системном вызове wait(2) на WSTAT(5), сведения о сигналах на страницах SIGINFO(5) и SIGNAL(5).

Использование системного руководства

Ссылки на системное руководство в тексте учебного пособия, а также и в самом системном руководстве, выглядят так: name(section), где name — имя страницы, а section — номер или название секции.

Основным средством доступа к системному руководству является команда man(1). Простейший вызов этой команды выглядит так: `man ls`. Эта форма команды выдает страницу руководства по команде ls(1). Если необходимо указать секцию руководства, в Solaris необходимо использовать форму команды `man -s 2 read` — эта форма команды выдает страницу read(2). Чаще всего, секцию руководства необходимо указывать, когда в разных секциях существуют одноименные страницы, например, read(1) (комадна shell) и read(2) (системный вызов) или passwd(1) (команда shell для смены пароля) и passwd(4) (формат файла БД учетных записей).

При выводе на терминал, команда man пропускает вывод через фильтр more(1), позволяющий листать текст по страницам, а также искать в тексте строки, используя шаблоны (регулярные выражения).

По умолчанию, прокрутка на одну строку делается нажатием <ENTER>, а прокрутка на страницу (на один экран терминала) — нажатием пробела. Перемещение на строку назад делается символом 'b', а на страницу — Ctrl-B. Для поиска необходимо ввести символ /, шаблон для поиска и <ENTER>. Для повторного поиска того же шаблона можно ввести /<ENTER>.

Синтаксис шаблона аналогичен шаблонам, используемым командами grep(1) и sed(1), а также многими программистскими редакторами, такими, как vi/vim, emacs, gedit, NetBeans, Eclipse. Большинство символов соответствуют самим себе. Символ '.' соответствует любому символу, символ '*' соответствует нулю или более вхождений предыдущего выражения (например, предыдущего символа), '+' - одному или более вхождениям. Выражение в квадратных скобках задает диапазон символов, например, [0-9] соответствует любой десятичной цифре. Точный синтаксис регулярных выражений, поддерживаемых more(1), может быть найден на странице руководства regex(5)

Для выхода из more(1) (при этом команда man также завершается) можно использовать команду 'q' (quit).

По умолчанию, man(1) ищет страницы руководства в каталоге /usr/share/man, где расположены страницы, входящие в стандартную поставку системы. Для просмотра руководства по дополнительным пакетам, например, по компилятору SunStudio, может быть необходимо подключить дополнительные каталоги для поиска. Это может быть сделано опцией -M, например, команда man -M /opt/SolarisStudio12.3-solaris-x86-bin/solarisstudio12.3/man cc выдаст руководство по компилятору языка C из пакета SunStudio (точное значение параметра -M должно соответствовать местоположению пакета SunStudio в вашей системе). Также, вместо параметра -M дополнительные каталоги можно подключать при помощи переменной среды MANPATH.

Еще одна полезная команда, связанная с man(1) — это команда apropos(1). Эта команда осуществляет поиск по заголовкам страниц руководства. Так, apropos file выводит список всех страниц руководства, в заголовке (точнее, в секции NAME) которых содержится подстрока file.

У ряда Unix-систем, в том числе у Solaris, страницы системного руководства выложены в Интернет в виде веб страниц, у Solaris — на сайте http://docs.oracle.com. При использовании интернет-версий, необходимо внимательно следить, соответствуют ли страницы на сайте вашей версии системы. На сайте Oracle доступны руководства от всех версий Solaris, начиная с 2.4.

Формат страницы руководства

. ЗАГОЛОВОК (TITLE) - обычно имя библиотечной функции или системного вызова

. СЕКЦИЯ (SECTION) - раздел Руководства

. БИБЛИОТЕКА (LIBRARY) - для секции 3, библиотечных функций, например C, S, M, E, X или G.

. ИМЯ (NAME) - имя и краткое описание системного вызова или библиотечной функции (в одной строке)

. ИСПОЛЬЗОВАНИЕ (SYNOPSIS) - как вызвать системный вызов/библиотечную функцию. Для функций библиотек языка C и системных вызовов в этой секции указан прототип функции, а также все заголовочные файлы, которые необходимо включить для корректной работы с функцией, например, те, где описан прототип функции и типы, используемые этой функцией в качестве параметров или кода возврата.

. ОПИСАНИЕ (DESCRIPTION) - описывает работу системного вызова или функции. Часто, самый длинный раздел руководства.

. ВОЗВРАЩАЕМОЕ ЗНАЧЕНИЕ (RETURN VALUE) - как интерпретировать возвращаемый код.

. ОШИБКИ (ERRORS) — для системных вызовов и библиотечных функций, использующих системные вызовы, перечисляет все допустимые значения errno и их значение.

. СМ. ТАКЖЕ (SEE ALSO) - страницы Руководства, имеющие отношение к этой странице.

. АТРИБУТЫ (ATTRIBUTES) — стандарты, которым соответствует функция, а также особенности при использовании в многопоточной программе и при работе с длинными файлами (файлами, длина которых превышает 2Гб)

Кроме того, страница Руководства может содержать разделы:

. ПРИМЕРЫ (EXAMPLES)

. ФАЙЛЫ (FILES)

. СООБЩЕНИЯ (DIAGNOSTICS)

. ЗАМЕЧАНИЯ (NOTES)

. (ПРЕДУПРЕЖДЕНИЯ) WARNINGS

. (ОШИБКИ) BUGS

. (ПРОБЛЕМЫ) CAVEATS

Пример страницы руководства (секция 3)

Библиотечные функции перечислены в Секции 3.

(3C) означает что эта функция размещена в стандартной библиотеке Си (в /usr/lib/libc.so).

ИМЯ имя функции и краткое описание.

ИСПОЛЬЗОВАНИЕ соответствующий include-файл и объявление функции perror. Объявление говорит, что perror(3C) ничего не возвращает (void) и требует один аргумент. Аргумент должен быть адресом символа, в данном случае - адресом символьной строки, оканчивающейся нулевым байтом.

ОПИСАНИЕ работа функции. perror(3C) печатает текст, переданный в качестве аргумента, затем двоеточие и пробел, затем сообщение об ошибке по текущему значению errno и перевод строки. Описание сообщений об ошибках можно найти на странице Руководства INTRO(2) и в файле <errno.h>, а также в разделе ERRORS страниц руководства по системным вызовам.

Сообщение об ошибке может быть включено в отформатированную пользователем строку при помощи strerror(3C). Эта функция возвращает указатель на сообщение об ошибке по значению errno. Эта функция использует внешнюю переменную errno как индекс во внешнем массиве, называемом sys_errlist[]. Это массив указателей на различные сообщения об ошибках, связанных со значением errno. Эти две переменные описываются на странице STRERROR(3C).

Пример: printf("A %s caused the error\n", strerror(errno));

Пример страницы руководства (Секция 2)

ИМЯ название системного вызова и однострочное описание

ИСПОЛЬЗОВАНИЕ необходимые include-файлы. time(2) возвращает значение типа time_t, представляющее время в секундах с 1 января 1970 года, 00:00:00 UTC. time требует один

параметр, адрес time_t. Тип time_t представляет собой typedef, определенный в <sys/types.h>. Если параметр ненулевой, возвращаемое значение также запоминается по адресу, заданному в качестве параметра.

ОПИСАНИЕ указывает, что time(2) возвращает значение time_t, соответствующее текущему времени, измеренному в секундах с 1 января 1970 года, 00:00:00 UTC.

time(2) возвращает -1 в случае неуспеха. Некоторые системные вызовы могут по различным причинам завершаться неудачно. Многие страницы Руководства перечисляют коды ошибок, начинающиеся с буквы 'E', которые обозначают конкретную причину неуспеха.

На 32-битных платформах, time_t определен как 32-битное целое. 19 января 2038 года в 03:14:07 UTC 32-битный time_t должен переполниться. Это известно как «Проблема 2038 года». На 64-битных платформах используется 64-битный time_t и этой проблемы не существует. Официальная позиция Oracle состоит в том, что ни одна из 32-битных версий ОС, поддерживаемых Oracle, не имеет ожидаемого срока жизни, достигающего 2038 года, поэтому на 2012 год решения для 32-битных ОС не предлагается.

Поскольку можно предположить, что к 2038 году значительное число встраиваемых и портативных устройств по прежнему будет использовать 32-битные процессоры, можно ожидать появления переходного ABI, аналогичного переходному ABI для использования 64-битного off_t в 32-битных программах (ABI для 64-битного off_t будет обсуждаться в разделе «Системные вызовы ввода-вывода»)ю

ВОЗВРАЩАЕМОЕ ЗНАЧЕНИЕ объясняет значения, возвращаемые при успехе или неуспехе.

СМ. ТАКЖЕ перечисляет функции или системные вызовы, имеющие отношение к данному. stime(2) используется суперпользователем для установки часов. ctime(3C) используется для преобразование значения time(2) в удобный для человека формат.

Примеры использования time(2)

На следующей странице приведены примеры различных способов использования системного вызова time.

. описания и инициализации. Должны быть включены файлы, перечисленные в Руководстве. Они объявляют системный вызов time(2) и описывают typedef time_t. Описаны четыре переменные типа time_t и указатель на time_t. При этом указатель инициализируется.

. нулевое значение параметра time(2) означает, что должно использоваться возвращаемое значение. Под результат не было выделено памяти, и предполагается использовать возвращаемое значение. . инициализированный указатель в качестве аргумента. Время будет сохранено в ячейке памяти, на который указывает переменная tp. Поскольку tp инициализирован, *tp совпадает с переменной t2. Кроме того, то же самое значение будет присвоено t3.

. адрес переменной в качестве аргумента. Прочитанное значение системных часов будет помещено в переменную типа time_t. Преобразование типа возвращаемого значения в void означает, что возвращаемое значение не будет использоваться.

ПРИМЕРЫ ИСПОЛЬЗОВАНИЯ time(2)

. объявления и инициализации

```
#include <sys/types.h>
#include <time.h>
time_t t1, t2, t3, t4;
time_t *tp = &t2;
```

. нулевое значение параметра time(2) означает, что должно использоваться возвращаемое значение

```
t1 = time ( 0 );
```

. инициализированный указатель в качестве аргумента

```
t3 = time ( tp );
```

. адрес переменной в качестве аргумента

```
(void) time ( &t4 );
```

Пример страницы руководства (секция 3)

В разделе ИСПОЛЬЗОВАНИЕ описано несколько функций.

. Должен быть включен файл <time.h>.

. ctime, localtime и gmtime все требуют по одному параметру. Это адрес переменной или поля структуры типа time_t. Значение этой переменной может быть установлено обращением к time(2).

- ctime возвращает адрес символа. Это адрес начала символьной строки, завершающейся нулевым символом. Формат строки не зависим от настроек языка. Для вывода даты и времени на национальном языке необходимо использовать функции strftime(3C) и strptime(3C).

- localtime и gmtime возвращают адрес структуры tm.

Раздел ОПИСАНИЕ описывает возвращаемое значение и элементы структуры tm.

Замечания:

1. localtime сначала вызывает tzset, чтобы получить значение переменной среды TZ. На основании этого значения tzset инициализирует массив указателей на строки tzname и внешние переменные timezone и altzone. timezone устанавливается равной разнице времени между UTC и данной временной зоной, измеренному в секундах. Например, в Нью Джерси,

```
tzname[0] == "EST" и timezone == 5*60*60
tzname[1] == "EDT" и altzone == 4*60*60
```

2. Поле tm_isdst структуры tm при вызове функций localtime и gmtime устанавливается положительным, если действует сезонное изменение времени, нулевым, если оно не действует, и отрицательным, если информация недоступна.

3. Внешняя переменная daylight устанавливается положительной, только если в данной TZ используется сезонное изменение времени.

4. localtime реализована следующим образом: она вычитает timezone из прочитанного значения time и вызывает gmtime. Раздел ПРОБЛЕМЫ говорит о «перезаписи» содержимого: функции ctime, asctime, localtime и gmtime возвращают указатель на внутренний буфер. Содержимое этого буфера может быть перезаписано последующим вызовом соответствующих функций.

5. В разделе АТРИБУТЫ указано, что функции ctime, asctime, localtime и gmtime в Solaris 10 возвращают указатели на локальные данные нити, и поэтому являются безопасными для использования в многопоточной программе (в каждой нити эти функции будут возвращать разные указатели). Тем не менее, использование этих функций не рекомендуется, потому что стандарт POSIX не требует, чтобы эти функции были безопасны в многопоточных программах. Рекомендуется использовать реентерабельные версии этих функций: localtime_r, gmtime_r и т.д. Также указано, что сами по себе функции безопасны, но прямая модификация переменных timezone, altzone и daylight во время работы этих функций может приводить к непредсказуемым результатам.

Библиотечная функция mktime(3C) может быть использована для преобразования структуры tm в календарное время (количество секунд с 00:00:00 1 января 1970).

Библиотечная функция difftime(3C) возвращает разницу между двумя календарными временами как double. Эта функция нужна, потому что, в соответствии с требованиями POSIX, для типа time_t может быть не определено основных арифметических операций.

Пример использования time(2) и ctime(3C)

Эта программа демонстрирует использование time(2) и ctime(3C). Она работает следующим образом:

9 Определяет переменную now типа time_t.

10 Определяет указатель sp на struct tm.

12 Вызывается time(2). Время в секундах от 00:00:00 UTC 1 января 1970 сохраняется в переменной now.

14 Библиотечная функция ctime(3C) преобразует календарное время в ASCII-строку формата date(1). Адрес, возвращенный этой функцией, используется в качестве параметра printf для печати ASCII-строки.

16 Вызывается библиотечная функция localtime(3C). Эта функция заполняет значениями поля структуры tm.

17-20 Распечатываются значения полей структуры tm.

Файл: ex_time.c

ПРИМЕР ИСПОЛЬЗОВАНИЯ
time(2) И ctime(2)

```
 1 #include <sys/types.h>
 2 #include <stdio.h>
 3 #include <time.h>
 4 #include <stdlib.h>
 5 extern char *tzname[];
 6
 7 main()
 8 {
 9     time_t now;
10     struct tm *sp;
11
12     (void) time( &now );
13
14     printf("%s", ctime( &now ) );
15
16     sp = localtime(&now);
17     printf("%d/%d/%02d %d:%02d %s\n",
18         sp->tm_mon + 1, sp->tm_mday,
19         sp->tm_year, sp->tm_hour,
20         sp->tm_min, tzname[sp->tm_isdst]);
21     exit(0);
22 }
```

1. СРЕДА ИСПОЛНЕНИЯ

Обзор

В данном разделе изучается среда исполнения программ в Unix. Вы ознакомитесь с понятием процесса, основными атрибутами процесса и методами доступа к этим атрибутам.

Определение процесса

Процессом называется экземпляр исполняющейся программы вместе с данными этой программы и набором атрибутов, хранящихся в ядре операционной системы. Например, файл /bin/date - это программа. Каждый раз при запуске этой программы создаётся процесс, который выполняет считывание показаний системных часов, переводит их в читаемый человеком формат и выводит на терминал.

Каждый процесс в Unix имеет идентификатор процесса (process id, pid). Строго говоря, этот идентификатор не является уникальным: система переиспользует свободные значения идентификаторов при создании новых процессов. Однако, одновременно в системе не может быть двух процессов с одинаковыми pid.

В разные моменты времени процесс может исполнять разные программы. Переключение исполняемой программы осуществляется вызовом exec(2), который будет изучаться в разделе «Запуск процессов и исполнение программ». Поскольку при замене программы pid процесса и многие другие атрибуты не изменяются, говорят, что это тот же самый процесс, но программа в нем была заменена.

Виртуальная память

Системы семейства Unix работают на машинах с диспетчером памяти и используют виртуальную память. Диспетчер памяти — это устройство, осуществляющее трансляцию виртуальных адресов в физические. Обычно диспетчер памяти осуществляет трансляцию на основе таблиц, формируемых операционной системой. Главная функция виртуальной памяти — это защита процессов друг от друга, а ядра системы — от процессов.

Ошибки работы с памятью, например, обращения по неинициализированным указателям или выход индекса за границы массива могут приводить к повреждению кода и данных программы, что, в свою очередь, может приводить к записи повреждённых данных в файлы, то есть к потере данных. Также, злоумышленник может модифицировать код ядра или других процессов для совершения различных вредоносных действий: доступа к чужим данным, запуска саморазмножающихся программ («вирусов»), рассылки «спама» и т. д. Чтобы защититься от всего этого, ОС при помощи диспетчера памяти ограничивает диапазоны адресов, к которым может обращаться пользовательская программа, и отображает эти диапазоны адресов разных процессов на разные страницы физической памяти. Таким образом, процесс не может обратиться к памяти ядра или другого процесса.

Все процессоры с диспетчером памяти имеют хотя бы два разных режима работы с разным уровнем привилегий (у некоторых процессоров таких режимов больше). Unix использует два режима: режим с высшим уровнем привилегий, называемый системным, и режим с низшим уровнем привилегий, называемый пользовательским.

В системном режиме, код может менять настройки диспетчера памяти, осуществлять доступ к внешним устройствам, менять некоторые настройки самого процессора, а также переключать режим, то есть переходить в пользовательский режим. Внешние прерывания и аппаратные исключения, например, исключение по делению на ноль, приводят к переключению в системный режим. В пользовательском режиме все перечисленное запрещено, за исключением переключения режима.

В пользовательском режиме доступна специальная команда. У разных процессоров эта команда называется по разному, например, у PDP-11 — EMT, у Intel 80286 — вызов шлюза, у современных процессоров x86 производства Intel – SYSCALL. Эта команда переключает режим на системный и одновременно с этим передаёт управление по определённому адресу. Таким образом, гарантируется, что пользователь не может исполнить произвольный код в привилегированном режиме.

Код, исполняющийся в привилегированном режиме, называется системным, или кодом ядра. Код, исполняющийся в пользовательском режиме, называется пользовательским, даже если этот код, как стандартная библиотека языка C или стандартные утилиты, такие, как /bin/date, поставляется вместе с системой.

Точка, в которую передаёт управление команда включения системного режима, называется диспетчером системных вызовов. Пользовательский код использует вызовы этой точки для передачи ядру запросов на исполнение различных функций, которые он не может выполнять сам: запросов на ввод-вывод, на получение дополнительной памяти (это требует изменения настроек диспетчера памяти), для взаимодействия с другими процессами, для получения информации из системных структур данных и т.д.

Ядро, строго говоря, не является процессом в. Ядро не имеет идентификатора процесса и других атрибутов процесса. Во многих Unix-системах есть процессы, весь код которых исполняется в режиме ядра. Так, в Solaris существует процесс sched, имеющий pid=0. Его можно увидеть, выведя список всех процессов командой pid -aef. Эти процессы не являются самим ядром, они задействуют лишь часть кода ядра и работают лишь с частью его данных. Фактически, это нити исполнения, которые ядро создаёт для выполнения различных задач.

Виртуальное адресное пространство

Диапазон адресов, который доступен процессу, называется виртуальным адресным пространством. Поскольку каждый процесс имеет свою таблицу трансляции виртуальных адресов в физические, говорят, что процесс имеет своё адресное пространство.

Теоретически, максимальный объем адресного пространства ограничен длиной виртуального адреса и на 32-битных машинах составляет 4Гб. На практике, некоторые адреса зарезервированы системой для различных целей. Например, на процессорах x86/x64 ядро системы отображено в адреса пользовательского процесса. Это связано с особенностями диспетчера памяти x86, которые в нашем курсе детально не обсуждаются. На рисунках структуры адресного пространства эта область называется контекстом ядра (kernel context).

Также, большинству программ не нужно 4Гб памяти. Система отводит процессу столько памяти, сколько запросила программа, а остальное адресное пространство защищает от доступа, например, устанавливая на соответствующие страницы защиту как от чтения, так и от записи.

Виртуальная память пользователя для каждого процесса подразделяется на три обязательных сегмента: текст, данные и пользовательский стек. Когда программа загружается в память, информация из файла a.out используется для размещения и инициализации сегментов текста и данных. Кроме того, в современных системах, процесс обычно имеет динамические сегменты — сегменты кода и данных разделяемых библиотек, отображенные на память файлы, разделяемую память System V IPC и др.

Редактор связей вводит переменные для пометки определенных частей программы (смотри END(3C)). Адрес etext указывает на конец текстового сегмента. Адреса edata и end указывают соответственно на конец инициализированной и не инициализированной области данных.

Сегмент текста содержит команды, полученные из кода, написанного пользователем. Текст всегда загружается по одному и тому же виртуальному адресу. Это виртуальный адрес зависит от типа машины и от параметров, указанных редактору связей. Если два процесса исполняют одну и ту же программу с разделяемым сегментом текста, то только одна копия программы загружается в память. Таблицы трансляции всех процессов, использующих эту программу, настраиваются так, чобы указывать на эту память.

Всякий раз когда программа с разделяемым текстом загружается с память, производится просмотр таблицы сегментов кода чтобы выяснить, не используется ли уже текст этой программы. Если используется, тогда новый процесс будет разделять уже загруженный программный текст.

За сегментом текста следует сегмент данных, содержащий статические и внешние переменные. Секция неинициализированных данных содержит неинициализированные статические и внешние переменные и очищается при загрузке процесса.

Стек пользователя используется для сохранения активационных записей (activation records), содержащих локальные переменные, аргументы вызовов функций, значения регистров, возвращаемые значения функций и другую информацию. Код, создающий и уничтожающий эти записи, генерируется компилятором и вставляется в начало и конец каждой функции языка C. При каждом вызове функции запись активации размещается я в стеке. При возвращении из функции запись активации освобождается. Место расположения стека и направление его роста машинно-зависимы. Попытка обращения к памяти между окончанием сегмента данных и вершиной стека приводит к ошибке обращения к памяти.

Порядок расположения секций, расстояние между секциями и направление роста стека зависят от центрального процессора (ЦП) и версии ОС. На следующих слайдах приведены

структуры адресного пространства Solaris для x86 и x64.

Пользовательская область (user area)

Операционная система сохраняет информацию о процессах в структурах данных, размещаемых в памяти ядра: в дескрипторах процесса и пользовательских областях (user area). На каждый процесс заводится только одна пользовательская область. Пользовательская область — это системный сегмент данных небольшого фиксированного размера, который содержит информацию, необходимую при исполнении этого процесса, например дескрипторы открытых файлов, реакцию на сигналы, информация о системных ресурсах.

Пользовательская область, несмотря на название, не является непосредственно доступной для пользователя, так как она находится в памяти ядра. Даже если ядро отображено в адресное пространство процесса, на соответствующих страницах памяти стоят атрибуты, делающие эти страницы доступными только в системном режиме. Для получения/установки информации, хранящейся в пользовательской области, должны использоваться системные вызовы.

Кроме атрибутов процесса, пользовательская область содержит стек, которым ядро пользуется при выполнении системных вызовов. Функции ядра не могут размещать свои записи активации на пользовательском стеке, ведь тогда другая нить пользовательского процесса могла бы подменить параметры функций или адрес возврата и вмешаться в работу кода ядра. Если в рамках процесса исполняется несколько нитей, то пользовательская область должна содержать соответствующее количество стеков, чтобы каждая нить могла исполнять системные вызовы независимо от других.

Программа, показывающая расположение сегментов текста, данных и стека

Эта программа показывает расположение переменных различных классов хранения в виртуальном адресном пространстве программы.

2 Описание макроподстановки для печати адресов переменных.

3 Объявление внешних переменных etext, edata и end

4 Инициализация внешней статической переменной. Это класс хранения используется для закрытия доступа к переменным извне данного файла.

5 Инициализированные и неинициализированные внешние переменные.

10 Инициализированные и неинициализированные статические переменные.

11 Инициализированные и неинициализированные локальные переменные.

13 Печать виртуальных адресов пользователя внутри текстового сегмента.

16 Печать адресов инициализированных статических и внешних переменных.

19 Печать адресов неинициализированных статических и внешних переменных.

22-25 Печать адресов локальных переменных, включая аргументы командной строки

29 Вызов sub1(), она также печатает адреса переменных.

Файл: tds_loc.c

ВЫЗОВ:
$tds_loc
1 main at 800000f4 and sub1 at 80000246
2 end of text segment at 8000305c
3 s at 80005064
4 b at 80005068
5 d at 8000506c
6 end of initialized data at 800058d0
7 a at 80005b2c
8 c at 800058d0
9 end of uninitialized data at 8000634c
10 m at c00200d4
11 n at c00200d5
12 argc at c00200a4
13 argv at c00200a8
14 argv[0] at c002007c
15 t at 800058d4
16 p at c00200d8
17 v at c0020100

ПРОГРАММА, ПОКАЗЫВАЮЩАЯ РАСПОЛОЖЕНИЕ СЕГМЕНТОВ ТЕКСТА, ДАННЫХ И СТЕКА

```
 1 #include <stdio.h>
 2 #define PRADDR(A) printf(#A " at %p\n", &A)
 3 extern etext, edata, end;
 4 static char s = 'S';
 5 int a, b = 1;
 6
 7 main(int argc, char *argv[])
 8 {
 9    void sub1(int);
10    static int c, d = 1;
11    char m, n = 'n';
12
13    printf("main at %p and sub1 at %p\n", main, sub1);
14    printf("end of text segment at %p\n", &etext);
15
16    PRADDR(s); PRADDR(b); PRADDR(d);
17    printf("end of initialized data at %p\n", &edata);
18
19    PRADDR(a); PRADDR(c);
20    printf("end of uninitialized data at %p\n", &end);
21
22    PRADDR(m); PRADDR(n);
23    PRADDR(argc); PRADDR(argv);
24    for (b = 0; b <argc; b++)
25    printf("argv[%d] at %p\n", b, &argv[b]);
26    sub1(c);
27 }
28
29 void sub1(int p)
30 {
31    static int t;
32    char v;
33    PRADDR(t); PRADDR(p); PRADDR(v);
34 }
```

Взаимодействие процессов

Каждый процесс имеет родительский процесс. Процесс 0 (sched) создается ядром при запуске системы необычным образом и является собственным родителем. Как уже говорилось, весь код и данные процесса 0 исполняются в режиме ядра. Можно сказать, что этот процесс работает в своей собственной пользовательской области, или, что то же самое, все время проводит внутри системного вызова.

Первый «нормальный» процесс, который имеет пользовательский код, порождается sched и имеет pid=1. По умолчанию, этот процесс исполняет програму /sbin/init. При загрузке ядра можно указать, что запускать в качестве init. Это может быть полезно при восстановлении системы после аварии, но при нормальной работе не используется.

Init — это процесс, порождающий все другие пользовательские процессы. Он читает командный файл /etc/inittab (см. inittab(4)) и запускает все остальные задачи в системе, используя fork(2) и exec(2). inittab определяет, какие процессы должны быть порождены на конкретном уровне запуска системы (runlevel). В обычном inittab Unix System V содержится запуск скриптов из каталогов /etc/rc?.d, где символ ? меняется в зависимости от уровня запуска. Уровень запуска 2 обычно соответствует многопользовательскому уровню, когда пользователям разрешено входить в систему. Скрипты в каталоге /etc/rc2.d определяют, какие файловые системы должны монтироваться и какие процессы-демоны (daemon) должны быть запущены. В частности, в многопользовательском режиме init запускает ttymon для консоли и контроллер сервисов sac.

/usr/lib/saf/sac - контроллер сервисов в Unix SVR4. Он стартует, когда машина переводится в многопользовательский режим. Главная функция sac — чтение командного файла /etc/saf/_sactab и запуск определенных в нем программ, например менеджера графических дисплеев gdm(1M) и сервера удаленного входа в систему sshd(1M). Далее sac опрашивает эти программы и отслеживает их статус. Sac позволяет управлять сервисными процессами при помощи команд svcs(1M) и sacadm(1M), подобно тому, как это делается в OS/2 и Windows командами net start и net stop. В других Unix-системах, сервисные процессы запускаются напрямую скриптами из /etc/rc?.d или даже из /etc/inittab.

. /usr/lib/saf/ttymon обслуживает терминалы. Он устанавливает переменную среды TERM (тип терминала) и определяет активность терминального порта и выдает приглашение входа в систему. После ответа на приглашение входа в систему, ttymon запускает программу /bin/login.

. login проверяет регистрационное имя пользователя и его пароль. При успешной проверке, login запускает поверх себя командный интерпретатор shell, определенный в файле passwd или в другой БД учетных записей. login производит установку следующих переменных среды исполнения:

HOME=шестое поле в файле паролей

LOGNAME=регистрационное имя пользователя

MAIL=/var/mail/[регистрационное имя пользователя]

PATH=/usr/bin

SHELL=седьмое поле файла паролей (установка происходит, только если 7-ое поле не равно нулю)

. После того как login запускает поверх себя shell, то shell сначала читает команды из файла /etc/profile, а затем из файла $HOME/.profile, если этот файл существует. Затем shell выдает приглашение для ввода команды. После этого shell интерпретирует команды пользователя и выполняет их как порожденные процессы.

Взаимодействие процессов (продолжение)

Графический вход в систему происходит аналогично тому, как вход через терминал, только роль ttymon и login играет программа X Display Manager или, в Solaris 10/11, Gnome Display Manager — gdm(1M). Эта программа обслуживает вход в систему как с локальных дисплеев, так и по сети по протоколу XDMCP (по умолчанию, сетевой вход обычно запрещен). На локальном дисплее, gdm запускает сервер X Window, устанавливает с ним соединение и выдает окно с запросом на ввод имени и пароля пользователя.

При успешном входе, gdm устанавливает все перечисленные на предыдущей странице переменные среды, а также переменную DISPLAY=имя дисплея X Window

Эта переменная используется программами, поддерживающими протокол X Window (X(5)), для установления соединения с сервером.

Затем gdm запускает командный интерпретатор shell, который интерпретирует файл /etc/gdm/Xsession. Этот файл считывает стандартные стартовые файлы shell /etc/profile и $HOME/.profile, так что все настройки переменных среды, которые сделали администратор системы и пользователь, также загружаются. Наконец, Xsession запускает менеджер графических сессий gnome-session(1), который и запускает, собственно, графическую пользовательскую среду. Если из графической среды вам необходимо запустить интерактивный shell или другие команды, ориентированные на работу с текстовым терминалом или со стандартными потоками ввода и вывода, можно запустить терминальный эмулятор gnome-terminal. Этот терминальный эмулятор создает псевдотерминал — специальное псевдоустройство, которое играет роль терминального порта для процессов соответствующей сессии.

Обычно запущенные с пользовательского терминала процессы, принадлежат одной и той же сессии (session). Сессия также создается для графической среды пользователя. При запуске терминальных эмуляторов, для каждого терминального окна создается своя сессия. Процессы одной сессии могут принадлежать разным группам процессов. Процесс может создать новую группу процессов (став лидером группы), вызвав setsid(2) или setpgid(2). Порожденные процессы наследуют от родительского процесса идентификаторы группы процессов, сессии и управляющий терминал. Группы процессов и сессии важны для управлениями заданиями и обработки сигналов. В частности, у терминала в каждый момент времени есть «основная» группа процессов. Только процессы этой группы могут читать данные с терминала. Это будет рассматриваться в разделе «Терминальный ввод-вывод».

Командные процессоры с управления заданиями, такие, как ksh(1) и bash(1), создают новую группу процессов для каждой запускаемой команды, а сами исполняются в своей собственной группе, содержащей только сам процесс shell. Исполнение процесса из основной группы, связанной с терминалом, может быть остановлено вводом стоп-символа (<CTRL z> по умолчанию). После этого shell делает свою группу основным процессом и выдает на терминал приглашение для ввода новой команды. Процессы остановленной группы затем могу быть запущены в фоновом режиме командой bg или снова сделаны основной группой командой fg. Имея несколько остановленных или фоновых групп, пользователь может переключаться между ними. Каждая такая группа и называется «заданием» (job). Например, переключение заданий можно использовать для того, чтобы иметь несколько запущенных текстовых редакторов (если ваш редактор не поддерживает одновременное редактирование нескольких файлов) или редактор и утилиту man для чтения страницы руководства. Управление заданиями в ksh(1) и bash(1) описано на соответствующих страницах руководства.

Среда исполнения процесса

Процесс имеет некоторый набор параметров, который называется его средой исполнения.Термин «среда» (environment) имеет два значения, узкое и широкое.

В узком смысле этого слова, «среда» обозначает совокупность экспортированных переменных командного интерпретатора shell.

Термин «среда исполнения процесса» шире и включает экспортированные переменные shell, открытые файлы, текущую директорию, устанавливаемые по умолчанию права доступа при создания файла и т. д. Также, частью среды исполнения процесса могут считаться глобальные параметры настройки системы, влияющие на работу процесса, например, системное ограничение на максимальную длину путевого имени файла или максимальный суммарный объем параметров exec(2).

Когда процесс начинает свое исполнение, он наследует большинство параметров среды исполнения от родителя. Затем процесс может изменять свою среду. Для этого в языке C доступны библиотечные функции и системные вызовы, а в shell — встроенные команды. Если, после изменения своей среды, процесс запустит другой процесс (станет его родителем), новый процесс унаследует измененную среду.

Среда инициализируется процессом init и модифицируется при входе пользователя в систему программами login или gdm. При удаленном входе в систему при помощи ssh(1) или XDMCP также могут передаваться переменные среды. Так, по умолчанию, ssh передает переменные среды TERM (тип терминала) и TZ (временная зона, часовой пояс), так что удаленная сессия живет в соответствии с настройками часового пояса той системы, за клавиатурой которой сидит пользователь.

Shell изменяет свою среду исполнения при интерпретации входного файла /etc/profile. Этот файл содержит все команды и установки переменных среды, которые системный администратор хочет исполнить для каждого входящего в систему. Если личная директория содержит файл .profile, shell читает команды из этого файла и модифицирует среду конкретного пользователя. Поскольку все процессы терминальной сессии являются потомками входного shell, то все они наследуют сделанные в /etc/profile и $HOME/.profile настройки.

Поменять большинство параметров среды другого процесса (как родителя, так и потомка) после его создания штатными средствами невозможно. Два исключения из этого правила — это идентификатор группы процессов и идентификатор родителя. Используя системный вызов setpgid(2), можно менять групповую принадлежность других процессов вашей терминальной сессии. Изменение идентификатора родительского процесса осуществляется косвенным путем: если родительский процесс завершается раньше, чем какой-то из его потомков, «осиротевший» процесс усыновляется процессом init (процессом с pid=1).

Самый простой способ изменить остальные параметры среды исполнения другого процесса — это подключиться к этому процессу отладчиком и исполнить в контексте этого процесса функции или системные вызовы, изменяющие его среду. Вполне возможно, что такие изменения могут нарушить работу процесса. Так, например, изменение временной зоны (переменной TZ) в тот момент, когда программа исполняет функцию localtime(3C), может иметь труднопредсказуемые последствия. Поэтому и штатных средств для внесения таких изменений не предусмотрено. Именно поэтому, в shell, команды изменения среды, такие, как cd, umask, ulimit, export реализованы как встроенные команды, а не как внешние программы.

Среда исполнения процесса (продолжение)

Информация о среде исполнения содержится в двух местах: в пользовательской области процесса и в стеке процесса. Как уже говорилось, параметры, хранящиеся в пользовательской области, напрямую не доступны пользовательскому коду, и некоторые из этих параметров процесс не может менять. В пользовательской области размещены:

. Номер самого процесса и его родителя

. Идентификатор группа процессов. Процесс входит в группу процессов. Группы используются при управлении заданиями, а также при отправке сигналов (можно отправить сигнал всем процессам группы) и при некоторых других операциях.

. Ограничения процесса - максимально доступное время ЦП, максимальный размер сегмента данных, максимальный размер стека, максимальный размер создаваемого файла, максимальное число открытых файлов.

. Идентификатор сессии. Группы процессов являются членами сессии. Сессия может иметь не более одного управляющего терминала. Определенные символы, посланные с управляющего терминала, вызывают посылку сигналов группам процессов соответствующей сессии. Управляющий терминал играет важную роль в обработке сигналов остановки и прерывания и в управлении заданиями.

. Права доступа пользователя: реальный и эффективный идентификаторы пользователя, группы, роли.

. Информацию о файловой системе.

. Действия, совершаемые при получении сигнала.

Пользовательский стек также содержит информацию об среде исполнения процесса. Размещенные в стеке параметры передаются процессу при системном вызове exec(2) и включают в себя:

. Параметры командной строки (argc/argv)

. Переменные среды (экспортированные переменные shell)

И параметры командной строки, и переменные среды, представляют собой наборы строк, заканчивающихся символом '\0' (как и строковые литералы в языке C).

Как получить доступ к среде исполнения

В программе на языке С можно получить прямой доступ только к той части среды исполнения, которая сохраняется в стеке. Доступ к аргументам командной строки можно осуществить через первые два параметра функции main: int argc и char ** argv. Параметр argc определяет количество аргументов, массив argv содержит указатели на них. Количество аргументов можно также определить по тому, что массив argv всегда заканчивается нулевым указателем. Штатных средств для изменения значений аргументов не предусмотрено.

Для разбора аргументов командной строки доступна функция getopt(3C), которая описывается в приложении к этому разделу.

При запуске программ из shell следует иметь в виду, что secли один из аргументов команды содержит символы *, ? или [, shell интерпретирует такой аргумент как шаблон имени файла (точный формат шаблона описан на страницах руководства fnmatch(5) и sh(1)). Shell находит все файлы, соответствующие шаблону (если шаблон содержит также символы /, поиск может вестись в других каталогах; так, шаблон */* соответствует всем файлам во всех подкаталогах текущего каталога) и заменяет шаблон на список аргументов, каждый из которых соответствует одному из имён найденных файлов. Если файлов, соответствующих шаблону, не найдено, шаблон передаётся команде без изменений. Если вам нужно передать команде сам шаблон (например, команда find(1) или некоторые архиваторы ожидают шаблон имени файла, который следует найти), соответствующий аргумент необходимо экранировать одиночными или двойными кавычками, например find . -name '*.c' -print.

Доступ к переменным среды можно получить через третий аргумент main, char ** envp, или через внешнюю переменную char ** envp, определенную в библиотеке libc. Общее количество переменных среды можно определить по тому, что массив envp также оканчивается нулевым указателем, как и argv. Каждая строка массива envp представляет собой описание одной переменной. Значение строки обязательно содержит символ =, например, TZ=Asia/Novosibirsk. Часть строки слева от = считается именем переменной (TZ в предыдущем примере), а справа — значением (Asia/Novosibirsk в предыдущем примере). Если = - последний символ в строке перед '\0', говорят, что переменная имеет пустое значение.

Также для доступа к переменным среды можно пользоваться библиотечными функциями getenv(3C) и putenv(3C). Функция getenv(3C) возвращает значение переменной с указанным именем или нулевой указатель, если такой переменной нет. Функции putenv(3C) и setenv(3C) позволяют изменять переменные среды. Если переменная с указанным именем уже была, putenv и setenv заменяют ее значение; если такой переменной не было, они создают новую переменную.

При добавлении новых переменных, putenv/setenv могут выделить новую память для размещения массива envp при помощи malloc(3C). При этом изменится значение переменной envp, но НЕ третьего параметра main.

Рекомендуется обращаться к переменным среды при помощи getenv(3C), потому что на не Unix-системах список переменных среды может иметь другую структуру, и программа, анализирующая массив envp самостоятельно, потребует переделки. Кроме того, в многопоточной программе, вызов putenv(3C) в одном потоке параллельно со сканированием envp в другом потоке может привести к непредсказуемым результатам. Функции putenv(3C) и getenv(3C) в Solaris адаптированы к работе в многопоточной среде.

Доступ и изменение остальной информация о среде исполнения может производиться только с помощью системных вызовов или библиотечных функций, содержащих системные вызовы.

Системные вызовы для доступа к среде исполнения процесса

. Идентификатор процесса:

- getpid(2) возвращает идентификатор процесса. Например: pid=getpid();

- getppid(2) возвращает идентификатор родительского процесса. Например: ppid=getppid();

- getpgid(2) возвращает идентификатор группы для процесса, идентификатор которого равен pid, или для вызывающего процесса, если pid равен 0. Например: pgid=getpgid(0); Замечание: группы процессов обсуждаются позже в этом курсе.

- setpgid(2) устанавливает идентификатор группы процесса.

- getsid(2) возвращает идентификатор сессии для процесса, идентификатор которого равен pid, или для вызывающего процесса, если pid равен 0. Например: sid=getsid(0);

- setsid(2): создает новую сессию и делает текущий процесс её лидером, или (что то же самое) устанавливает идентификаторы группы и сессии вызывающего процесса равными значению идентификатора данного процесса и освобождает управляющий терминал.

. Идентификаторы пользователя и группы

- getuid(2) возвращает реальный идентификатор пользователя. Например: uid=getuid();

- geteuid(2) возвращает эффективный идентификатор пользователя. Например: euid=geteuid(); Замечание: обычно реальный и эффективный идентификаторы совпадают. Способы установки идентификатора пользователя обсуждаются далее в этом разделе.

- getgid(2) возвращает реальный идентификатор группы. Например: gid=getgid();

- getegid(2) возвращает эффективный идентификатор группы. Например: egid=getegid(2);

- setuid(2), setgid(2) устанавливают идентификаторы пользователя и группы. Замечание: только суперпользователь (super-user) может устанавливать произвольные значения uid. Все остальные могут приравнять реальный uid эффективному или эффективный реальному.

- setgroups(2) устанавливает список групп доступа вызываемого процесса. Замечание: только суперпользователь может исполнять этот системный вызов.

- getgroups(2) получает список групп доступа вызываемого процесса. group(1) использует getgroups(2) для печати списка групп доступа данного пользователя.

- initgroups(3C) инициализирует список групп доступа всеми группами, которым принадлежит учетная запись. Только суперпользователь может выполнять этот системный вызов.

. Ресурсы процесса:

- getrlimit(2) получает информацию о некоторых программных и аппаратных ограничениях процесса.

- setrlimit(2) устанавливает некоторые программные и аппаратные ограничения процесса.

. Терминал процесса

- ttyname(3C) возвращает путевое имя специального терминального файла с заданным дескриптором. Например: char *buf; buf=ttyname(0);

Системные вызовы для доступа к системным параметрам

В ядре Unix есть ряд глобальных настроек, которые влияют на исполнение процессов или отдельных системных вызовов. Некоторые из этих параметров могут устанавливаться в конфигурационных файлах ядра и изменяются после перезагрузки системы. Некоторые другие параметры можно изменять только путем перекомпиляции ядра. Некоторые параметры, например, максимальная длина имени файла, зависят от файловой системы.

В старых учебниках, для проверки значения этих параметров рекомендовалось использовать препроцессорные макросы, определенные в заголовочном файле <limits.h>. В Solaris этот файл генерируется автоматически при генерации системы=, и значения в нем соответствуют значениям, использованным при компиляции ядра системы. Если программа исполняется на другой версии ядра, значения некоторых параметров могут измениться.

Так, в <limits.h> определена символьная константа PATH_MAX, обозначающая максимальную длину путевого имени файла. В Solaris 11 эта константа равна 1024. Если в Solaris 12 по каким-то причинам этот лимит будет увеличен, то ваша программа не сможет работать с некоторыми файлами. Чтобы адаптировать вашу программу к новому ядру, ее необходимо будет перекомпилировать.

Чтобы избежать перекомпиляции прикладных программ, в действующей версии стандарта POSIX были введены системные вызовы, позволяющие определить во время исполнения параметры ядра системы и налагаемые этими параметрами ограничения.

Системный вызов sysconf(2) имеет один параметр, имя запрашиваемой системной переменной. Имена определены в заголовочном файле <unistd.h>; их полный список приведен на странице руководства sysconf(2). При обсуждении некоторых системных вызовов в этом курсе будут упоминаться параметры sysconf, влияющие на поведение соответствующего вызова.

Пример - определение максимальной длины имени часового пояса (допустимое значение переменной TZ):

```
long val;
val=sysconf(_SC_TZNAME_MAX);
```

Системные вызовы pathconf(2) и fpathconf(2) используется для получения параметров, связанных с файловой системой. Большинство из этих параметров могут быть различны для разных файловых систем, а некоторые также могут различаться и для. Поскольку в Unix практически любая директория может использоваться в качестве точки монтирования другой файловой системы, такие параметры необходимо заново запрашивать в каждой новой директории.

Параметры pathconf и fpathconf

fd — дескриптор открытого файла (это понятие будет обсуждаться в разделе «Файловый ввод-вывод»). Параметры будут определены для файловой системы, в которой размещен этот файл.

path — путевое имя файла или каталога, определяющее файловую систему, для которой необходимо определить параметр.

name — имя параметра. В качестве имен рекомендуется использовать символьные константы, определенные в <unistd.h>. Полный список имен приведен на странице руководства pathconf(2).

Пример — определение максимальной допустимой длины путевого имени файла:

```
long val;
val=pathconf("/", _PC_PATH_MAX);
```

Программа доступа к переменным среды

Эта программа показывает, как переменные среды первоначально расположены в стеке, и что environ и третий аргумент main() указывают на одну и ту же таблицу адресов. Также демонстрируется использование getenv(3C) и putenv(3C).

4 Внешняя переменная environ содержит адрес таблицы, в которой хранятся указатели на все переменные среды.

6 При запуске main значение третьего параметра envp такое же, как и у environ.

10 printenv() печатает значение env и environ, а также адреса из таблицы и строку (имя=значение), расположенную по этому адресу. Останавливается по достижению нулевого адреса.

11 putenv("TZ=PST8PDT") изменяет значение TZ. Адрес, указывающий значение переменной TZ, указывает уже не на стек, а на сегмент данных, т.к. используется строковая константа.

13-14 Переменная WARNING вноситься в среду и печатается содержание среды.

15 Печать значения, возвращенного getenv(3C), в виде строки.

18-30 Описание printenv()

Файл: envex.c

```
$ envex
 1 envp contains c0020060
 2 environ contains c0020060
 3 My environment variables are:
 4 (c0020060) = c0020006 -> HOME=/home/jrs
 5 (c0020064) = c0020015 -> LOGNAME=jrs
 6 (c0020068) = c0020021 -> MAIL=/var/mail/jrs
 7 (c002006c) = c0020034 -> PATH=/usr/bin:.
 8 (c0020070) = c0020049 -> TZ=EST5EDT
 9
10 envp contains c0020060
11 environ contains c0020060
12 My environment variables are:
13 (c0020060) = c0020006 -> HOME=/home/jrs
14 (c0020064) = c0020015 -> LOGNAME=jrs
15 (c0020068) = c0020021 -> MAIL=/var/mail/jrs
16 (c002006c) = c0020034 -> PATH=/usr/bin:.
17 (c0020070) = 80002d98 -> TZ=PST8PDT
18
19 envp contains c0020060
20 environ contains 80006498
21 My environment variables are:
22 (80006498) = c0020006 -> HOME=/home/jrs
23 (8000649c) = c0020015 -> LOGNAME=jrs
24 (800064a0) = c0020021 -> MAIL=/var/mail/jrs
25 (800064a4) = c0020034 -> PATH=/usr/bin:.
26 (800064a8) = 80002d98 -> TZ=PST8PDT
27 (800064ac) = 80002da4 -> WARNING=Don't use envp after putenv()
28
29 value of WARNING is: Don't use envp after putenv()
```

Замечание: putenv(3C) для новой переменной вызывает malloc(), перемещая таблицу адресов из стека в сегмент данных. Вся таблица перемещается в сегмент данных.

ПРОГРАММА ДОСТУПА К ПЕРЕМЕННЫМ СРЕДЫ

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3
 4 extern char **environ;   /* system variable */
 5
 6 main(int argc, char *argv[], char *envp[])
 7 {
 8    void printenv(const char **);
 9
10    printenv(envp);
11    putenv("TZ=PST8PDT");
12    printenv(envp);
13    putenv("WARNING=Don't use envp after putenv()");
14    printenv(envp);
15    printf("value of WARNING is: %s\n", getenv("WARNING"));
16 }
17
18 void printenv(const char **envp)
19 {
20    char **p;
21
22    printf("envp contains %8x\n", envp);
23    printf("environ contains %8x\n", environ);
24
25    printf("My environment variables are:\n");
26 /* loop stops on encountering a pointer to a NULL address*/
27    for (p = environ; *p; p++)
28    printf ("(%8x) = %8x -> %s\n", p, *p, *p);
29    putchar('\n');
30 }
```

Использование переменных среды (PATH)

Некоторые переменные среды воздействуют на поведение некоторых программ и библиотечных функций. Так, переменная среды PATH воздействует на поведение shell и функций execvp(2) и execlp(2) (эти функции описаны в разделе руководства 2, которое посвящено системным вызовам, но на самом деле это функции- «обертки» над вызовами execl и execv). Эта переменная среды представляет собой список разделенных двоеточиями имен директорий. При запуске команды, shell и execlp/execvp ищут файл, совпадающий с именем команды, во всех директориях, перечисленных в PATH. Если в разных каталогах лежит несколько одноименных файлов, будет запущен тот файл, который был найден первым, поэтому порядок директорий в PATH важен.

В отличие от командных процессоров DOS, OS/2 и Windows, командные процессоры Unix по умолчанию НЕ ищут исполняемые файлы в текущем каталоге. Если вы хотите, чтобы поиск в текущем каталоге происходил, необходимо добавить имя . или пустую строку в PATH. Это можно сделать командами

```
$ PATH=$PATH:.
```
или

```
$ PATH=$PATH:
```
Оба примера добавляют текущий каталог в конец PATH, так что он будет просматриваться после остальных директорий, а не перед ними, как в OS/2 и Windows. На самом деле, добавлять текущий каталог в PATH опасно, а в начало PATH это может быть даже недопустимо. Ведь при таком PATH вы можете по ошибке исполнить файл из текущего каталога вместо стандартной команды. Особенно это опасно если вы находитесь в чужом каталоге, содержимое которого вам неизвестно. Если PATH не содержит текущего каталога, то написанные вами программы можно запускать с указанием путевого имени, например:

```
$ ./a.out
```

Использование переменных среды (TZ)

Еще одна переменная, которая влияет на поведение многих программ и важных библиотечных функций — это переменная TZ. Системные часы Unix используют Всемирное координированное время (UTC), которое приблизительно соответствует среднему времени по Гринвичу (GMT). Именно такое время возвращает системный вызов time(2) и такое же время используется во многих временных штампах, например во временах создания и модификации файлов. Значение, возвращаемое time(2) определено как число секунд с 0 часов 0 минут 0 секунд по UTC 1 января 1970 года. Разумеется, если часы вашего компьютера идут неточно, time(2) может возвращать ошибочные значения.

Для перевода системного времени и системных временных штампов в местный часовой пояс используются такие функции, как ctime(3C) и localtime(3C). Эти функции определяют «местный часовой пояс» и местные правила перевода летнего-зимнего времени на основании значения переменной среды TZ или содержимого файла /etc/timezone, если переменная TZ не определена. Таким образом, изменяя TZ, можно «переселять» процессы в разные часовые пояса. Это может быть удобно, например, если вы удаленно зашли по ssh(1) на машину, расположенную в другом городе.

Допустимые значения TZ описаны на странице руководства environ(5). Наиболее употребительный формат значения TZ, применяемый в современных Unix-системах — это имя файла в каталоге /usr/share/lib/zoneinfo/ . В поставку Solaris входит обширная база описаний часовых поясов, основанная на так называемой IANA Time Zone Database, содержащая не только смещения от UTC и правила перевода летнего и зимнего времени для практически всех административных часовых поясов в мире, но и исторические изменения указанных сведений. Так, для всех часовых поясов на территории бывшей Российской Империи указано, что до 1919 года время на этих территориях имело смещение от UTC, не кратное часу, потому что отсчитывалось не от Гринвича, а от меридиана Пулковской обсерватории. При изменениях часовых поясов, база данных обновляется. Пользователи Solaris получают эти обновления вместе с остальными обновлениями операционной системы.

В Solaris, описания часовых поясов из IANA Time Zone Databaze компилируются специальной программой zic(1) в небольшие бинарные файлы, которые интерпретируются функциями стандартной библиотеки.

Бит установки идентификатора пользователя и setuid(2)

При входе в систему идентификаторы пользователя и группы устанавливается на основе учетной записи пользователя, например из файла /etc/passwd. Реальные и эффективные идентификаторы для процесса обычно совпадают. Эффективный идентификатор пользователя и список групп доступа (показываемый getgroups(2)) используются для определения прав доступа к файлам. Владелец любого файла, созданного процессом, определяется эффективным идентификатором пользователя, а группа файла — эффективным идентификатором группы.

В Unix, исполняемые файлы могут иметь специальные атрибуты: биты установки идентификатора пользователя или группы (setuid- и setgid-биты). Эти биты устанавливаются с помощью команды chmod(1) или системного вызова chmod(2). Биты setuid и setgid показываются командой ls -l буквой 's' на месте обычного расположения 'x', например:

```
$ ls -l `which su`
-r-sr-xr-x   1 root     sys        38656 Oct 21  2011 /usr/bin/su
```

Если один или оба этих бита установлены, при запуске такого файла, эффективный идентификатор пользователя и/или группы у процесса становится таким же, как и у владельца и/или группы файла с программой. Благодаря этому механизму можно стать «заместителем» или «представителем» привилегированной группы.
Используя программы с setuid-битом, можно получить доступ к файлам и устройствам, которые обычным образом недостижимы. Например, если какой-либо файл данных доступен по чтению и записи только для владельца, другие пользователи не могут получить доступ к этому файлу. Если же владелец этого файла напишет программу доступа к этому файлу и установит setuid-бит, тогда все пользователи данной программы смогут получить доступ к файлу, ранее недостижимому. Патент на механизм установки идентификатора пользователя принадлежит Дэннису Ричи, одному из разработчиков первой версии Unix.

Некоторые стандартные команды ОС UNIX имеют setuid-бит. Например, информация о паролях находится в файле /etc/shadow. Этот файл может читать только владелец — привилегированный пользователь (root). Тем не менее, каждый пользователь может изменить свой пароль. Это возможно, потому что владельцем программы изменения пароля /bin/passwd является root и у этой программы установлен setuid-бит. Другой пример - /sbin/ping. Благодаря этой программе пользователь может отправлять и получать пакеты ICMP ECHO и ECHO REPLY, что требует открытия так называемого «сырого» (raw) сокета. Это недоступно обычному пользователю.

Программа с setuid- или setgid-битом будет иметь эффективный идентификатор отличный от реального либо до конца времени выполнения процесса, либо до применения setuid(2) или setgid(2). С помощью этих системных вызовов, эффективный идентификатор вновь можно сделать равным реальному. setuid(2) или setgid(2) используются в программе, если владелец программы хочет предоставить пользователю возможность быть его «представителем», но в программе есть часть, где нет необходимости в специальных привилегиях.

Суперпользователь может применять setuid(2) для установки идентификатора, равному идентификатору любого пользователя в системе.

Программа /bin/login, запускаемая от имени root использует числовые идентификаторы пользователя и группы, обнаруженные в файле /etc/passwd, как аргументы setuid(2) и setgid(2) для установки реального идентификатора пользователей при входе в систему.

Поскольку суперпользователь может в любой момент стать любым другим пользователям, в традиционных Unix-системах он также имеет другие привилегии, например, возможность читать и писать любые файлы, не обращая внимания на права. В Solaris есть возможность более гибкого управления такими привилегиями, называемая RBAC (Role-Based Access Control, управление доступом основанное на ролях, privileges(5)). Администратор системы

может дать привилегию читать файлы, не обращая внимание на права, любому пользователю. Это может быть нужно, например, оператору резервного копирования.

Программа, использующая механизм setuid

Это программа демонстрирует механизм установки идентификатора пользователя. Она имеет setuid-бит и ее владельцем является владелец файла answer. Так как setuid-бит установлен, обучаемые получают право писать в файл преподавателя.

Сопутствующий протокол вывода показывает, что у программы quiz установлен setuid-бит.

6 Открывается файл answer для дозаписи.

10-12 Задается вопрос обучаемым.

13-16 Принимается только ответ a или b.

17 Ответ записывается в файл answer.

Вывод программы, демонстрирующий механизм установки идентификатора пользователя.

. Директория, содержащая answer, доступна для записи толькопользователю instr. Никто больше не может создавать или уничтожить там файлы.

. Заметим, что владелец instr может читать/писать в файл answer.

Программа quiz имеет setuid-бит и может быть выполнена кем угодно. Бит setuid может быть установлен командой chmod~u+s~quiz. Если преподаватель просмотрит содержимое файла answer после того, как пользователь с реальным идентификатором 507 закончит исполнение программы quiz, он увидит, что ответ "пользователя 507" записан в файл answer, писать в который разрешено только владельцу.

. Преподаватель просматривает содержимое файла answer.

. Обучаемый пытается просмотреть содержимое файла answer, а затем выполняет программу quiz.

ПРОГРАММА, ИСПОЛЬЗУЮЩАЯ МЕХАНИЗМ УСТАНОВКИ ИДЕНТИФИКАТОРА ПОЛЬЗОВАТЕЛЯ

```
 1 #include <unistd.h>
 2 #include <stdlib.h>
 3 #include <stdio.h>
 4 main()
 5 {
 6     FILE *fp; static char response;
 7     if((fp = fopen("answer","a+")) == NULL){
 8     fprintf(stderr,"Cannot open answer\n");
 9     exit(1);
10      }
11      printf("When opening a file,\n");
12      printf("which id is checked?\n");
13      printf(" (a)real (b) effective\n");
14      while( response != 'a' && response != 'b'){
15      printf("Answer(a/b)? ");
16      scanf("%c%*c", &response);
17      }
18      fprintf(fp,"%5ld:%c\n", getuid(), response);
19      fclose(fp);
20 }
```

Два сеанса работы за двумя терминалами:

```
$ id                                $ id
uid=75(instr) gid=21(unixc)         uid=507(stu1) gid=1(other)
$ chmod u+s quiz

$ ls -ld . quiz answer | cut -c1-24,55-  # at both terminals
drwxr-xr-x   1 instr  .
-rw-------   1 instr  answer
-rwsr-xr-x   1 instr  quiz

$ cat answer                        $ cat answer
  511:b                             cat: cannot open answer
  503:b                             $ quiz
  505:a                             When opening a file,
  508:b                             which id is checked?
                                  (a)real (b) effective
                                  Answer(a/b)? b
```

Приложение. Разбор опций из командной строки

getopt(3C) - это функция общего назначения для обработки опций командной строки. Командная строка должна удовлетворять следующим правилам:

Общий формат: имя_команды [опции] [другие аргументы]

имя_команды Имя выполняемого файла.

опции Должны начинаться со знака минус. Каждая опция представляет собой один символ. Каждая опция может иметь собственный разделительный минус или несколько опций могут совместно использовать один знак минус. Некоторые опции требуют аргументов. Аргумент может следовать непосредственно за символом опции или быть отделен от нее пробелом. Смотрите intro(1) для сверки с общепринятым синтаксисом командной строки.

другие аргументы Следуют за опциями и не обязательно должны начинаться со знака минус.

Пример:

```
ls -lt /tmp;   pr -n file1 file2;
```

getopt(3C) обычно исполняется в начале программы. Она вызывается в цикле для последовательной обработки опций программы. У getopt(3C) три аргумента:

- Целый argc. Обычно первый аргумент main().

- Указатель на вектор символов argv. Обычно второй аргумент main().

- Указатель на символ (строка параметров). Это строка допустимых опций. Если у опции есть аргументы, то за соответствующим символом строки должно стоять двоеточие.

getopt(3C) возвращает одно из следующих целых значений:

буква верной опции

-1 при обработке первого аргумента не опции

getopt(3C) использует четыре внешних переменных:

optarg указатель на символ. Когда getopt(3C) обрабатывает опцию, у которой есть аргументы, optarg содержит адрес этого аргумента.

optind целое. Когда getopt(3C) возвращает -1, argv[optind] указывает на первый аргумент не-опцию.

opterr целое. Когда getopt(3C) обрабатывает недопустимые опции, сообщение об ошибке выводится на стандартный вывод диагностики. Печать может быть подавлена установкой opterr в ноль.

optopt целое. Когда getopt(3C) возвращает '?', optopt содержит значение недопустимой опции.

Для разбора многобуквенных опций можно использовать функцию getopt_long(3C), которая в этом курсе не рассматривается.

Использование getopt(3C) в программе

6 Создание строки допустимых опций. Опции, за которыми следует двоеточие ":", требуют соответствующих параметров.

7 Объявление флагов необязательных опций.

10 Печать числа входных параметров.

11 Вход в цикл вызова getopt(3C) для просмотра командной строки, возвращается одна опция за один проход цикла.

12 Вход в выбор switch для обработки опций. Обычно флаг устанавливается для указания присутствия конкретной опции. Флаг используется в программе позже.

16,20 Если опции нужен параметр, тогда переменная optarg будет содержать адрес этого параметра. Если этот адрес будет использоваться позже, то он должен быть сохранен в символьном указателе.

24,25 Если обнаружена недопустимая опция, getopt(3C) вернет '?' и выдаст сообщение об ошибке на стандартный вывод диагностики. Выдача сообщения может быть выключена установкой opterr в ноль. optopt содержит значение недопустимой опции.

35-37 Когда getopt(3) возвращает -1, тогда argv[optind] указывает на первый аргумент командной строки, отличный от опции.

Файл: getopt_ex.c

```
$ getopt_ex -db -f abc -c -g hij -d file1 file2
  1 argc equals 10
  2 getopt_ex: illegal option -- b
  3 invalid option is b
  4 getopt_ex: illegal option -- c
  5 invalid option is c
  6 dflg equals 2
  7 f_ptr points to abc
  8 g_ptr points to hij
  9 invalid equals 2
 10 optind equals 8
 11 next parameter = file1
```

ИСПОЛЬЗОВАНИЕ getopt(3C) В ПРОГРАММЕ

```
 1 #include <stdlib.h>
 2 #include <stdio.h>
 3
 4 main(int argc, char *argv[])
 5 {
 6     char options[ ] = "f:dg:";  /* valid options */
 7     int c, invalid = 0, dflg = 0, fflg = 0, gflg = 0;
 8     char *f_ptr, *g_ptr;
 9
10      printf("argc equals %d\n", argc);
11      while ((c = getopt(argc, argv, options)) != EOF) {
12       switch (c) {
13       case 'd':
14           dflg++;
15           break;
16       case 'f':
17           fflg++;
18           f_ptr = optarg;
19           break;
20       case 'g':
21           gflg++;
22           g_ptr = optarg;
23           break;
24       case '?':
25           printf("invalid option is %c\n", optopt);
26           invalid++;
27       }
28      }
29      printf("dflg equals %d\n", dflg);
30      if(fflg)
31       printf("f_ptr points to %s\n", f_ptr);
32      if(gflg)
33       printf("g_ptr points to %s\n", g_ptr);
34      printf("invalid equals %d\n", invalid);
35      printf("optind equals %d\n", optind);
36      if(optind < argc)
37       printf("next parameter = %s\n", argv[optind]);
38 }
```

2. СИСТЕМНЫЕ ВЫЗОВЫ ВВОДА И ВЫВОДА

Обзор

В этом разделе вы узнаете о системных вызовах ввода/вывода (input/output, I/O). В язык C не встроены операторы ввода/вывода. Все операции ввода/вывода осуществляются системными вызовами или библиотечными функциями, которые, в свою очередь, обращаются к системным вызовам. В этом разделе обсуждаются системные вызовы нижнего уровня. Список функций стандартной библиотеки ввода/вывода (stdio) приведён в конце раздела. Студенты ФИТ НГУ изучали стандартную библиотеку ввода-вывода в курсе «Программирование на языке высокого уровня».

Основным понятием ввода-вывода в Unix является файл. Файл в Unix — это либо именованная совокупность данных в файловой системе, либо интерфейс для доступа к физическому или виртуальному устройству, либо интерфейс для доступа к некоторым средствам межпроцессного (трубы) и сетевого (сокеты) взаимодействия. Иногда файлы устройств, трубы, сокеты и некоторые другие объекты называют специальными файлами, а совокупности данных в файловой системы — обычными (регулярными) файлами.

Предоставляются системные вызовы, которые открывают файл, читают из файла, пишут в файл и закрывают файл. Кроме того, есть стандартные вызовы, которые изменяют текущую позицию чтения/записи в файле и которые управляют доступом к файлу. Большинство этих системных вызовов работают сходным образом как для регулярных, так и для специальных файлов.

В конце раздела обсуждаются файлы, отображённые в память.

Что такое файл?

Файл представляет собой последовательность байтов, никак не организованных операционной системой. Прикладной программе файл представляется непрерывной последовательностью байтов (это не означает, что соответствующие байты занимают непрерывное пространство на физическом устройстве). Не существует разницы между файлами, представляющими двоичные данные и текстовые данные. Ваша программа несет ответственность за преобразование (если оно необходимо) внешнего формата представления данных в требуемое представление. Например, функция atoi(3) удобна для преобразование чисел из текстового вида во внутреннее машинное двоичное представление.

Каждый байт регулярного файла адресуется индивидуально. Вновь созданный файл не содержит пространства для данных. Пространство под данные предоставляется при записи в дисковый файл. Система выделяет блоки физического диска под данные по мере необходимости.

Размер файла хранится в структуре данных, называемой inode. В самом файле не присутствует признака конца файла.

Операционная система UNIX рассматривает файл, как универсальный интерфейс с физическим устройством. Многие системные вызовы, применимые к файлам, могут быть применены и к байт- или блок-ориентированным устройствам. Однако некоторые вызовы, наподобие lseek(2), которые изменяют позицию ввода/вывода, неприменимы к файлам некоторых устройств, например, терминальных устройств.

Рассмотрим следующую программу:

```
main()
{
    printf("hi world\n");
}
```

В ней вызывается printf(3C), который, в свою очередь, вызывает write(2) от файлового дескриптора 1. Программа не потребует изменений, если вывод перенаправляется в файл или на другой терминал. Система UNIX ищет устройство или файл, ассоциированное с дескриптором 1, и вызывает требуемый драйвер устройства.

Обзор - стандартные функции ввода/вывода

Стандартные библиотечные функции ввода/вывода (stdio), реализованные как обёртка над системными вызовами, предоставляют больше возможностей и большую функциональность. Например, эти функции позволяют вводить и выводить:

- символ
- строку
- данные с преобразованием формата
- многочисленные структуры, содержащие двоичные и/или текстовые данные

Библиотечные функции ввода/вывода уменьшают количество системных вызовов за счёт буферизации вводимых и выводимых данных, поэтому эти функции называют функциями буферизованного ввода/вывода.

Схема буферизации предполагает перенос данных из буфера сегмента данных программы во вторичный буфер. Вторичный буфер по умолчанию имеет размер BUFSIZ, определенный в <stdio.h>. Когда вторичный буфер становится полным при выводе или пустым при вводе, производится необходимый системный вызов.

Системный вызов write(2) производит запись вторичного буфера в системный буфер, который, в конце концов, попадает на устройство вывода. read(2), в свою очередь, переносит данные из устройства ввода в системный буфер, оттуда — в пользовательский буфер. Из этого буфера символы попадают в обычно меньший по размерам буфер, предоставляемый стандартными библиотечными функциями ввода/вывода.

Замечания по поводу стандартных библиотечных функций ввода/вывода:

5. Буфер будет выведен, только если он полон, или при вызове fflush(3C). Это называется блочной буферизацией.
6. Файловые потоки могут быть строчно-буферизованы. Тогда буфер выводится при записи символа новой строки ASCII LF (в языке C этот символ обозначается как '\n'), при полном буфере или при запросе ввода. По умолчанию, вывод на терминал является строчно-буферизованным. Стандартный поток вывода (stdout) является строчно-буферизованным, если файловый дескриптор 1 назначен на терминал, и блочно-буферизованным, если файловый дескриптор 1 назначен на регулярный файл или какое-то другое устройство.
7. Файловые потоки могут быть небуферизованы. Стандартное устройство диагностики stderr небуферизовано.

Функция setbuf(3S) позволяет сделать потоки небуферизованными или переключить режим буферизации. Функция setvbuf(3S) также позволяет изменить размер буфера.

В общем, лучше использовать не системные вызовы ввода/вывода, а библиотечные функции. Во многих случаях это эффективнее, в частности — при переносе небольших порций данных. Использование системных вызовов может быть целесообразно в ситуациях, когда вам необходимо гарантировать, что операции ввода/вывода и другие системные вызовы будут исполнены в определённой последовательности, например, при синхронизации с другим процессом. Даже в таких ситуациях может быть удобнее выключить буферизацию на уровне библиотеки или использовать fflush(3C).

Список стандартных функций ввода/вывода приведен в конце данного раздела.

Открытие файла

Системный вызов open(2) открывает файл, то есть ассоциирует файловый дескриптор с обычным или специальным файлом. Файл должен быть открыт до того, как будет осуществляться чтение или запись. Файловый дескриптор используется для идентификации файла при обращении к другим системным вызовам. Файловый дескриптор представляет собой небольшое неотрицательное целое число. Аргументы open(2):

path указывает на строку символов, содержащую путевое имя файла. Имя может быть либо абсолютным (относительно корневого каталога), либо относительным (относительно текущего каталога). Длина имени ограничена параметром PATH_MAX, значение которого можно получить вызовом pathconf(2) с параметром _PC_PATH_MAX. Также, файловая система может накладывать ограничения на длину базового имени файла. Значение этого ограничения можно определить вызовом pathconf(2) с параметром _PC_NAME_MAX.

oflag указывает, в каком режиме следует открыть файл. Например, следует ли файл открыть для чтения и/или записи, создать и т.д. Значение этого флага получается с помощью побитового ИЛИ символьных констант, определённых в <fcntl.h>.

mode используется для установки битов прав доступа при создании файла. Внимание: ненулевое значение umask может изменить (уменьшить) права доступа, указанные в mode.

Значение, возвращаемое open(2), является наименьшим доступным файловым дескриптором от нуля до административного предела. Этот дескриптор используется системными вызовами read(2) и write(2), а также рядом других системных вызовов и некоторыми библиотечными функциями.

При запуске процесса из стандартных командных оболочек, таких, как sh, ksh, bash, файловые дескрипторы 0, 1 и 2 уже определены при старте программы. В других контекстах, процесс может запускаться с другими наборами предопределенных дескрипторов. Файловый дескриптор имеет смысл либо до закрытия вызовом close(2), либо до завершения программы. Файловые дескрипторы разных процессов могут иметь одинаковые значения, но при этом указывать на разные файлы.

Максимальный номер файлового дескриптора на единицу меньше, чем максимально допустимое для процесса количество одновременно открытых файлов (в стандарте POSIX это ограничение обозначается OPEN_MAX). Значение OPEN_MAX ограничено «мягким» и «жестким» пределами. Мягкий (административный) предел устанавливается setrlimit(2) с командой RLIMIT_NOFILE или командой ulimit(1). Жёсткий предел устанавливается настройками ядра системы. Значение жёсткого предела можно определить системным вызовом sysconf(2) с параметром _SC_OPEN_MAX. В Solaris, жесткий предел устанавливается параметром rlim_fd_max в файле /etc/system (system(4)); его изменение требует административных привилегий и перезагрузки системы.

В Solaris 10, максимальное значение rlim_fd_max равно MAXINT (2147483647).

По умолчанию, на 32-битной версии Solaris, OPEN_MAX равен 256, так как 32-битная версия стандартной библиотеки ввода-вывода использует байт для хранения номера дескриптора. Также, на 32-битной версии Solaris, используемая по умолчанию версия select(3C) поддерживает не более 1024 дескрипторов. Функция select(3C) обсуждается в разделе «Мультиплексирование ввода-вывода».

В 64-битной версии Solaris, rlim_fd_max по умолчанию равен 65536. 64-битная версия select(3C) поддерживает не более 65536 дескрипторов.

open(2) - Флаги

Следующие флаги, определенные в <fcntl.h>, могут быть объединены с

помощью побитового ИЛИ и использованы совместно в аргументе oflag

вызова open(2). Замечание: должен быть использован хотя бы один из

первых трех флагов, иначе вызов open(2) окончится неуспехом.

O_RDONLY Открывает файл для чтения.

O_WRONLY Открывает файл для записи.

O_RDWR Открывает файл для чтения и для записи.

O_APPEND Перед каждой записью помещает указатель файла в конец файла. Иными словами, все операции записи будут происходить в конец файла.

O_CREAT Создает файл, если он не существует.

O_TRUNC Стирает данные файла, устанавливая размер файла равным нулю.

O_EXCL Используется совместно с O_CREAT. Вызывает неуспех open(2), если файл уже существует.

O_SYNC Заставляет write(2) ожидать окончания физической записи на диск.

O_NDELAY,O_NONBLOCK Для некоторых устройств (терминалов, труб, сокетов), чтение может приводить к блокировке процесса, если в буфере устройства нет данных. Установка этих флагов приводит к переводу устройства в режим опроса. В этом режиме, чтение из устройства с пустым буфером возвращает управление немедленно.

O_NOCTTY Не позволяет терминальному устройству стать управляющим терминалом сессии.

Чтобы открыть файл с флагом O_RDONLY, необходимо иметь право чтения из этого файла, а с флагом O_WRONLY — соответственно, право записи. Права доступа проверяются только в момент открытия, но не при последующих операциях с файлом.

Без использования флага O_CREAT, open(2) открывает существующий файл и возвращает ошибку ENOENT, если файла не существует. При использовании флага O_CREAT, open(2) пытается открыть существующий файл с указанным именем, и пытается создать такой файл, если его не было.

Аргумент mode должен быть использован совместно с флагом O_CREAT при создании нового файла. Права доступа у вновь открытого файла будут установлены в непредсказуемое значение, если mode будет опущен. Биты прав доступа устанавливаются в значение mode & ~umask, где umask — значение полученное после выполнения команды umask(1). Обратите внимание, что аргумент mode объявлен как необязательный, поэтому компилятор C не выдаёт синтаксическую ошибку при его отсутствии.

Флаги O_NDELAY и O_NONBLOCK воздействуют на именованные программные каналы и специальные байт-ориентированные файлы. Использование флага O_NDELAY при открытии терминального файла и различие между O_NDELAY и O_NONBLOCK обсуждаются далее в этом разделе. Их воздействие на программные каналы обсуждается в разделе «Программные каналы».

Права доступа к файлу

Маска прав доступа в Unix представляет собой 12-разрядную битовую маску, которую удобно записывать в виде 4- или 3-разрядного восьмеричного числа. Старшие 3 бита (восьмеричные тысячи) кодируют setuid, setgid и sticky-биты. Следующие 9 бит кодируют собственно права доступа. Биты с 9 по 7 (восьмеричные сотни) кодируют права доступа хозяина файла, биты с 6 по 4 (восьмеричные десятки) — права группы, младшие три бита с 3 по 1 (восьмеричные единицы) — права всех остальных пользователей. Старший бит в тройке (4 в восьмеричной записи) кодирует право чтения, средний бит (2 в восьмеричной записи) — право записи, младший бит (1 в восьмеричной записи) — право исполнения.

Порядок бит, кодирующих права, соответствует порядку символов, описывающих права, в выдаче команды ls -l и ряда других стандартных утилит Unix. Так, запись -rw-r—r-- в выдаче ls -l соответствует битовой маске 0644 и означает, что хозяин имеет право чтения и записи, а группа и все остальные — только права чтения.

Атрибут процесса cmask (creation mask), который может быть установлен системным вызовом umask(2), представляет собой 9-битовую маску. При создании файла, в маске, указанной при вызове open(2), биты, соответствующие битам, установленным в cmask, очищаются. Так, cmask 0022 очищает биты права записи для группы и всех остальных и оставляет в неприкосновенности остальные биты. Таким образом, cmask позволяет задавать права доступа к файлам по умолчанию или, точнее, ограничения на права по умолчанию. Типичные значения cmask, применяемые на практике — 0022 (пользователь сохраняет все права, группа и все остальные не имеют права записи) и 0027 (пользователь сохраняет все права, группа не имеет права записи, все остальные не имеют никаких прав).

Права доступа к файлам и управление ими подробнее рассматриваются в разделе «Управление файлами».

Открытие файла - Примеры

Приведенные ниже объявления необходимы для перечисленных на этой странице примеров. Включение файла <fcntl.h> необходимо для использования символьных имен флагов open(2).

```
#include <fcntl.h>
#define TMPFILE "/tmp/tmpfile"
char account[] = "account";
int logfd, acctfd, fd, fdin, fdout;
char *file;
```

Файл account, находящийся в текущем каталоге, открывается для чтения.

```
acctfd = open(account, O_RDONLY);
```

Файл, на имя которого указывает file, открывается для записи. Если файл не существует, он создается с маской прав доступа 0600 (возможно, модифицированной umask). Если файл существует, он будет усечен до нулевого размера.

```
file = TMPFILE;
fd = open(file, O_WRONLY | O_CREAT | O_TRUNC, 0600);
```

Файл с абсолютным путевым именем ("/sys/log") открывается для записи. Все записи производятся в конец файла. Если файл не существует, он создаётся с правами доступа 0600.

```
logfd = open("/sys/log", O_WRONLY | O_APPEND | O_CREAT, 0600);
```

Файл, имя которого было передано main как первый аргумент, открывается на чтение/запись.

```
fdin = open(argv[1], O_RDWR);
```

Файл открывается на запись. Если он не существует, он создается. Если файл существовал, вызов open(2) будет завершен неуспешно. Заметьте, что код возврата open(2) проверяется в условии оператора if. Программа должна предпринять некие действия в случае неуспеха; в данном примере она распечатывает код ошибки.

```
if ((fdout = open(TMPFILE, O_WRONLY | O_CREAT | O_EXCL,
      0666)) == -1)
      perror(TMPFILE);
```

Флаг O_EXCL используется для предотвращения непреднамеренной перезаписи уже существующего файла. Используется совместно с O_CREAT. Вызов open(2) окончится неуспехом, если файл уже существует. Этот флаг не означает, что программа открыла файл исключительно для себя, или что это единственный файл, открытый программой.

Что же делает вызов open(2)?

Информация об открытых файлах хранится в ядре UNIX. Таблица файловых дескрипторов, размещенная в пользовательской области процесса, содержит указатели на системные структуры, описывающие файл. Сами эти структуры могут разделяться несколькими процессами, и поэтому находятся в разделяемой памяти ядра за пределами пользовательской области. Файловый дескриптор — это число в диапазоне от 0 до MAX_OPEN-1, которое является индексом в таблице файловых дескрипторов.

Системные файловые структуры содержат информацию о конкретном открытом файле. Для каждого вызова open(2) выделяется собственная структура, даже если два разных вызова открыли один и тот же файл на диске.

В структуре содержится следующая информация:

- . указатель на текущую позицию в файле. Этот указатель изменяется на прочитанное/записанное количество байт при каждом вызове read(2) и write(2). Кроме того, позицию в файле можно установить явно вызовом lseek(2).
- . копия флагов открытия. Эти флаги передаются вторым аргументом open(2).
- . счетчик ссылок. Это число различных файловых дескрипторов из одной или различных пользовательских областей, которые совместно используют данную системную структуру. Процесс может создавать новые дескрипторы, указывающие на имеющиеся структуры (ранее открытые файлы), системным вызовом dup(2). Кроме того, при создании нового процесса, он наследует все открытые родителем файлы — это соответствует ссылкам на одну структуру из разных пользовательских областей.
- . указатель на структуру информации о файле (образ инода в памяти).

Структура информации о файле имеет следующее строение:

- . номер устройства, на котором размещен файл, и номер инода
- . пользовательский идентификатор владельца файла и идентификатор группы файла.
- . счетчик ссылок - количество файловых дескрипторов, ссылающихся на данную структуру.
- . связи – количество записей в директориях, указывающих на данный файл.
- . тип файла – обычный, директория, специальный файл и пр.
- для специальных файлов устройств - «старшее» (major) и «младшее» (minor) числа и указатель на минорную запись устройства; «Старшее» число идентифицирует драйвер, а «младшее» - номер устройства, управляемого этим драйвером. Минорная запись (minor record) — системная структура данных, описывающая устройство, и содержащая блок переменных состояния устройства и указатели на функции драйвера.
- . права доступа данного файла.
- . размер файла в байтах.
- временные штампы создания файла, последней модификации и последнего доступа к файлу.
- . список номеров блоков для блоков данных на диске.

Закрытие файла

Системный вызов close(2) служит для закрытия файла, то есть для уничтожения файлового дескриптора. После вызова close(2), файловый дескриптор становится недействительным. Когда все файловые дескрипторы, указывающие на системную структуру данных, закрываются, структура данных также уничтожаются.

close(2) освобождает системные ресурсы. Если программы не закрывает файл, это производится системой при завершении работы программы, например, при вызове exit(2).

Если ваша программа использует много файловых дескрипторов, например, для работы с сокетами TCP/IP, необходимо закрывать неиспользуемые дескрипторы, иначе ваш процесс может столкнуться с нехваткой дескрипторов. Также, закрытие сокетов, труб и некоторых специальных файлов может иметь другие побочные эффекты. Так, закрытие дескриптора сокета TCP приводит к разрыву TCP-соединения, то есть к освобождению ресурсов как на локальном компьютере, так и на том узле сети, с которым было установлено соединение.

Чтение из файла

Чтение из файла осуществляется системными вызовами read(2) и readv(2). read(2) читает байты в единый буфер, в то время как readv(2) позволяет осуществлять разбросанное чтение в несколько несмежных буферов одним вызовом. Аргументы для этих вызовов:

fildes файловый дескриптор, полученный при предшествующем вызове open(2) с флагами O_RDONLY или O_RDWR.

buf указывает на место, в которое должны быть помещены прочитанные байты. Места должно хватить для nbyte байт.

nbyte максимальное число байт, которые следует считать. На самом деле может быть прочитано меньшее число байт.

iov указывает на массив структур struct iovec со следующими полями:

```
caddr_t   iov_base;
int       iov_len;
```

Каждая структура iovec содержит адрес и длину области памяти, куда будут помещены байты вызовом readv(2). Вызов заполняет последовательно указанные области памяти, переходя от одной к другой.

iovcnt количество структур iovec.

Значение, возвращаемое read(2) и readv(2) показывает количество байт, прочитанных на самом деле. Например, в файле осталось 10 байт, а nbyte равен 15. Значение, возвращаемое read(2) будет 10. Если read(2) будет вызвана вновь, она вернет 0. Это индикация конца файла.

Чтение из специального байт-ориентированного файла также может возвращать меньшее число байт, чем требуемое. Так, по умолчанию, терминалы буферизуют ввод по строкам (эта буферизация осуществляется драйвером терминального устройства; не следует путать ее с буферизацией библиотеки ввода-вывода). При чтении, драйвер терминала возвращает одну строку, даже если в буфере есть еще символы. Чтение из труб или сокетов, как правило, возвращает те данные, которые находятся в приемном буфере на момент вызова read(2), и не ожидает прихода дополнительных данных.

Значение nbyte никогда не должно превышать размера буфера buf. Нарушение этого условия может привести к срыву буфера — опасной ошибке программирования, которая может быть использована для исполнения произвольного кода с привилегиями вашего процесса, то есть для внедрения вирусов, червей и других вредоносных программ.

При чтении из регулярных файлов и из специальных файлов устройств, поддерживающих lseek(2), вызовы read(2) и readv(2) перемещают указатель чтения-записи. Это может быть неудобно в многопоточных программах или при работе с разделяемым файловым дескриптором из нескольких процессов. В таких ситуациях рекомендуется использовать вызов pread(2). У этого вызова есть дополнительный параметр off_t offset, указывающий позицию в файле, откуда следует начать чтение. pread(2) не перемещает указатель чтения-записи.

Запись в файл

Системные вызовы write(2) и writev(2) записывают данные в открытый файл. write(2) записывает данные из единого буфера, а writev(2) позволяет осуществлять запись из нескольких несмежных областей памяти одним вызовом. Аргументы для этих вызовов:

fildes файловый дескриптор, полученный при предшествующем вызове open(2) с флагами O_WRONLY или O_RDWR.

buf буфер с записываемыми байтами. Этот аргумент может быть указателем или именем массива.

nbyte максимальное число байт, которые следует записать. На самом деле, write может записать меньше байт, чем запрошено.

iov указывает на массив структур struct iovec, со следующими полями:

```
caddr_t iov_base;
int iov_len;
```

Каждая структура iovec содержит адрес и длину области памяти, откуда будут записаны байты вызовом writev(2).

iovcnt количество структур iovec.

Значение, возвращаемое write(2) и writev(2), показывает количество на самом деле записанных байт. Если достигнут предел размера файла (см. ulimit(2)), исчерпана дисковая квота или место на физическом устройстве, количество записанных байт будет меньше, чем nbyte.

Как и read(2), при работе с регулярными файлами и устройствами, поддерживающими lseek(2), вызовы write(2) и writev(2) перемещают указатель чтения-записи. В случаях, когда это нежелательно, рекомендуется использовать вызов pwrite(2), аналогичный вызову pread(2). Если вам нужно гарантировать, чтобы все записи всегда происходили в конец файла (например, если несколько потоков или процессов пишут сообщения в один лог-файл), следует использовать при открытии файла флаг O_APPEND.

Поскольку read(2) и write(2) являются системными вызовами, их использование ведет к накладным расходам, связанным со всеми системными вызовами. При работе с дисковыми файлов эти вызовы наиболее эффективны, если данные считываются блоками размером 512 байт или более (на практике, чем больше блок, тем лучше). Для чтения/записи одиночных байт, небольших структур данных или коротких строк эффективнее использовать функции буферизованного ввода-вывода.

Копирование ввода в вывод - Пример

Программа копирует данные из стандартного ввода, файловый дескриптор 0, в стандартный вывод, файловый дескриптор 1. Файловые дескрипторы 0, 1 и 2 (стандартный вывод диагностики) открываются при запуске программы из стандартных командных процессоров. Обычно они связаны с терминальным устройством. Это можно изменить, используя перенаправление ввода/вывода в командной строке.

Эта программа работает следующим образом:

2 файл <stdio.h> содержит определение BUFSIZ.

7 buf объявлен достаточно большим для вызова read(2)

10-11 требуется прочесть BUFSIZ байт. Истинное число считанных байт присваивается переменной n и, как правило, равно BUFSIZ. n может быть меньше BUFSIZ, если читается конец файла файла или если ввод осуществляется с терминала. Если данных больше нет, возвращается 0, что указывает конец файла. В каждой итерации цикла записывается n байт.

Для запуска программы наберите:

```
$ stdcopy <file1 >file2
```

Файл: stdcopy.c

КОПИРОВАНИЕ ВВОДА В ВЫВОД - ПРИМЕР

```
 1 #include <unistd.h>
 2 #include <stdio.h>
 3 #include <stdlib.h>
 4
 5 main()
 6 {
 7     char buf[BUFSIZ];
 8     int n;
 9
10     while ((n = read(0, buf, BUFSIZ)) > 0)
11         write(1, buf, n);
12     exit(0);
13 }
```

Копирование файла - Пример

Этот пример похож на предыдущий, но в этом случае копирование осуществляется из одного файла в другой. Этот пример похож на команду cp(1).

13-17 Проверка правильности числа аргументов

18-21 Первый аргумент программы – имя копируемого файла, который открывается на чтение

22-26 Второй аргумент – имя файла, открываемого на запись. Если файл не существует, он будет создан. Иначе он будет усечен до нулевого размера.

PMODE - символьная константа, используемая для установки битов прав доступа к файлу. В «настоящей» команде cp следовало бы копировать права доступа старого файла, но мы будем проходить чтение битов прав доступа только в разделе «Управление файлами»

28-29 Этими операторами производится цикл копирования. Возможно, хорошей идеей является сравнение количества действительно записанных байт (значение, возвращаемое функцией write(2)) с требуемым количеством (в данном случае n). Например, эти значения могут не совпадать, если достигнут предел размера файла или произошла ошибка записи на устройство.

Файл: copy.c

КОПИРОВАНИЕ ФАЙЛА - ПРИМЕР

```
 1 #include <stdio.h>
 2 #include <sys/types.h>
 3 #include <stdlib.h>
 4 #include <unistd.h>
 5 #include <fcntl.h>
 6 #define   PMODE     0644
 7
 8 main(int argc, char *argv[])
 9 {
10     int fdin, fdout, n;
11     char buf[BUFSIZ];
12
13     if (argc != 3) {
14     fprintf(stderr, "Usage: %s filein fileout\n",
15         argv[0]);
16     exit(1);
17     }
18     if ((fdin = open(argv[1], O_RDONLY)) == -1) {
19     perror(argv[1]);
20     exit(2);
21     }
22     if ((fdout = open(argv[2], O_WRONLY | O_CREAT |
23     O_TRUNC, PMODE)) == -1 ) {
24     perror(argv[2]);
25     exit(3);
26     }
27
28     while ((n = read(fdin, buf, BUFSIZ)) > 0)
29     write(fdout, buf, n);
30
31     exit(0);
32 }
```

Создание файла информации о служащих - Пример

Программа создает "записи" о служащих. Структура employee определена в файле employee.h:

```
 1 #define NAMESIZE  24
 2
 3 struct employee {
 4    char name[NAMESIZE];
 5    int salary;
 6 };
```

Каждой записи предшествует заголовок, содержащий дату/время создания записи и идентификатор пользователя, создавшего запись. Заголовок определен в файле empheader.h:

```
 1 struct recheader {
 2       char date[24];
 3       uid_t uid;
 4 };
 5 static void init_header(struct recheader *);
```

Программа работает следующим образом:

8-10 Перечисляются некоторые включаемые файлы. <sys/uio.h> содержит описание структуры для struct iovec.

15-16 Объявляются структуры данных записи о служащих и заголовка.

17 Объявляется массив из двух struct iovec.

23-27 Создается файл, имя которого задается первым аргументом (если он не существует). Если файл существует, open(2) завершается неуспехом и выдается сообщение об ошибке. Поскольку используется O_SYNC, writev(2) в строке 42 будет ждать завершения физической записи.

29-32 Инициализируются элементы массива iov — подставляются корректные адреса буферов и длины областей памяти.

34 Эта функция инициализирует заголовок.

36-42 Этот цикл создает запись о служащем на основе информации, поступающей со стандартного ввода. Заголовок и структура данных записываются в файл, заданный в командной строке.

Файл: create.c

СОЗДАНИЕ ФАЙЛА ЗАПИСЕЙ О СЛУЖАЩИХ - ПРИМЕР

```
 8 #include <sys/uio.h>
 9 #include "employee.h"
10 #include "empheader.h"
11
12 main(int argc, char *argv[])
13 {
14     int fd;
15     struct employee record;
16     struct recheader header;
17     struct iovec iov[2];
...
23     if ((fd = open(argv[1], O_WRONLY | O_CREAT |
24     O_SYNC | O_EXCL, 0640)) == -1) {
25     perror(argv[1]);
26     exit(2);
27     }
28
29     iov[0].iov_base = (caddr_t)&header;
30     iov[1].iov_base = (caddr_t)&record;
31     iov[0].iov_len = sizeof(header);
32     iov[1].iov_len = sizeof(record);
33
34     init_header(&header);
35
36     for (;;) {
37     printf("Enter employee name <SPACE> salary: ");
38     scanf("%s", record.name);
39     if (record.name[0] == '.')
40         break;
41     scanf("%d", &record.salary);
42     writev(fd, iov, 2);
43     }
44     close(fd);
45     exit(0);
46 }
```

Ожидание физической записи на диск

По умолчанию, Unix использует для работы с файлами отложенную запись. Системные вызовы write(2), writev(2) и pwrite(2) завершаются после переноса данных в системные буферы и не ожидают завершения физической записи на устройство. При использовании флага O_SYNC при открытии файла, система будет работать в режиме прямой или сквозной записи. При этом, вызовы write(2) будут ожидать физического завершения записи.

Если ожидание завершения физической записи необходимо только для некоторых операций, вместо флага O_SYNC можно использовать системный вызов fsync(2). Этот вызов завершается после переноса всех ожидавших записи данных, связанных с указанным файловым дескриптором, на физический носитель.

fsync(2) может быть использован программами, которым необходимо, чтобы файл к определенному моменту находился в заданном состоянии. Например, программа, которая содержит простейшие возможности выполнения транзакций, должна использовать fsync(2), чтобы гарантировать, что все модификации файла или файлов, осуществленные в процессе транзакции, были записаны на носитель, прежде чем оповещать пользователя о завершении транзакции.

Системный вызов sync(2) запрашивает запись на диск всех ожидающих записи блоков в системе (всего содержимого дискового кэша), но может вернуться прежде, чем эти операции будут завершены. Кроме пользовательских данных, дисковый кэш содержит модифицированные суперблоки, модифицированные иноды и запросы отложенного ввода-вывода других процессов. Этот вызов должен использоваться программами, которые анализируют файловую систему, и не рекомендуется к использованию прикладными программами общего назначения. Система автоматически производит запись данных из дискового кэша на диск, что, в определенном смысле, соответствует периодическому вызову sync(2).

Перемещение позиции чтения/записи файла

Системный вызов lseek(2) устанавливает позицию чтения/записи в открытом файле. Последующие вызовы read(2) и write(2) приведут к операции с данными, расположенными по новой позиции чтения/записи.

Параметр fildes является дескриптором файла, полученным после вызова open(2). lseek(2) устанавливает позицию файла fildes следующим образом:

Если whence равно SEEK_SET (символьная константа 0), позиция устанавливается равной offset.

Если whence равно SEEK_CUR (символьная константа 1), то позиция устанавливается равной offset плюс текущая позиция.

Если whence равно SEEK_END (символьная константа 2), позиция устанавливается равной размеру файла плюс offset.

Константы для whence определены в <unistd.h>. При удачном завершении, возвращается новая позиция чтения/записи, относительно начала файла. offset может быть как положительным, так иотрицательным. Попытка переместиться за начало файла вызывает неуспех и устанавливает код ошибки в errno.

lseek(2) может установить позицию в конец файла или за конец файла. При позиционировании в или за конец файла, read(2) вернет нулевое число прочитанных байт. Однако с этой позиции можно записывать данные. Блоки данных будут выделяться только при записи в блок.

Позиционирование за пределы файла и последующая запись может создать так называемый «разреженный файл», в некоторые участки которого запись никогда не производилась. Это не ошибка. Система не будет выделять блоки данных под участки, в которые никогда не было записи. Чтение из таких участков будет возвращать блоки, заполненные нулевыми байтами. При записи в такой участок, на диске будут выделены соответствующие блоки. При подсчете длины файла, «пропущенные» участки будут учитываться. Таким образом, длина файла обозначает не общий объем данных в файле, а максимально возможное логическое смещение в файле, из которого могут быть прочитаны данные.

Поддержка длинных файлов

На 32-разрядных системах, off_t по умолчанию имеет размер 32 бита. Учитывая то, что off_t — это знаковый тип, это затрудняет работу с файлами длиннее 2147483647 байт (2 гигабайта). Это связано с тем, что 32-битный ABI установился еще в 1980е годы, когда не были доступны диски и другие запоминающие устройства такого объема. В действительности, ядро современных версий Unix использует 64-разрядное смещение и может работать с файлами, превышающими 2Гб. Программы, использующие 32-битный off_t, могут последовательно читать данные из таких файлов, но не могут осуществлять произвольный доступ и даже не могут правильно определять размеры таких файлов.

Для решения этой проблемы, в Solaris 10 реализован переходный ABI, позволяющий 32-битным программам использовать 64-битный off_t. Для использования этого ABI из программы на языке C, рекомендуется определить препроцессорный символ _FILE_OFFSET_BITS равным 64 до включения любых системных include-файлов. Это приведет к использованию 64-битного определения off_t и к замене всех библиотечных функций и системных вызовов, использующих off_t или структуры, содержащие off_t, на 64-битные версии. Подробнее см. lfcompile(5).

Получение информации о служащих - Пример

Этот пример работает с «базой данных» записей о служащих. Эта база данных может быть создана программой create.c. Запись о служащем выводится по ее номеру. Нумерация начинается с 1. lseek(2) используется для перемещения позиции указателя к необходимой записи. Схема работает, только если все записи имеют одинаковую длину.

Программа работает следующим образом:

7-10 Перечисляются некоторые включаемые файлы. <unistd.h> определяет символьные константы SEEK_SET, SEEK_CUR, SEEK_END.

15-16 Объявляется запись о служащих и заголовок записи.

17 Объявлен массив из двух структур iovec.

24-27 Каждый элемент массива iov инициализируется адресом буфера и длиной области памяти, куда будут прочитаны данные.

29-41 В этом цикле пользователь вводит номер записи. lseek(2) помещает позицию на начало требуемой записи. readv(2) пытается прочесть данные заголовка и данные самой записи. readv(2) возвращает число прочитанных байт, если чтение произошло успешно. Иначе выводится сообщение "not found". Такое сообщение может возникнуть при попытке чтения за концом файла.

Файл: inquire.c

ПОЛУЧЕНИЕ ИНФОРМАЦИИ О СЛУЖАЩИХ - ПРИМЕР

```
 7 #include <unistd.h>
 8 #include <sys/uio.h>
 9 #include "employee.h"
10 #include "empheader.h"
11
12 main(int argc, char *argv[])
13 {
14     int fd, recnum;
15     struct employee record;
16     struct recheader header;
17     struct iovec iov[2];
...
24     iov[0].iov_base = (caddr_t)&header;
25     iov[1].iov_base = (caddr_t)&record;
26     iov[0].iov_len = sizeof(header);
27     iov[1].iov_len = sizeof(record);
28
29     for (;;) {
30    printf("\nEnter record number: ");
31    scanf("%d", &recnum);
32    if (recnum == 0)
33        break;
34    lseek(fd, (recnum-1)*(sizeof(record)+sizeof(header)
35        SEEK_SET);
36    if (readv(fd, iov, 2) > 0)
37        printf("Employee: %s\tSalary: %d\n",
38       record.name, record.salary);
39    else
40        printf("Record %d not found\n", recnum);
41     }
42     close(fd);
43     exit(0);
44 }
```

Создание копии дескриптора файла

Вызов dup(2) дублирует файловый дескриптор. Новый файловый дескриптор ассоциируется с тем же файлом, имеет ту же позицию в файле и те же права доступа, что и fildes. dup(2) всегда возвращает наименьший возможный (или, что то же самое, первый свободный) файловый дескриптор. Вызов dup2(2) также дублирует дескриптор, но начинает поиск свободного файлового дескриптора не с 0, как dup(2), а с указанного номера.

Основная задача dup(2) - обеспечить перенаправление ввода/вывода. Действительно, чтобы переназначить стандартный поток вывода (дескриптор 1), командный процессор должен закрыть старый дескриптор 1 и открыть новый файл. Если открытие нового файла по какой-то причине завершится неудачей, запускаемый процесс рискует остаться вовсе без стандартного потока вывода. Чтобы избежать этого, командные процессоры сначала открывают новый файл, затем, если файл открылся успешно, закрывают старый стандартный поток вывода и только потом переназначают новый файл на поток вывода вызовом dup(2).

В командных процессорах sh, ksh, bash и ряде других, переназначение вывода осуществляется символами > и >> (форма > обозначает перезапись файла, а форма >> обозначает запись выводимых данных в конец файла), а переназначение ввода символом <. Также, можно переназначать дескрипторы с 2 по 9, используя комбинации символов 2> и т. д.

Пример: запуск команды ls -l с переназначением вывода в файл (пробелы вокруг символа > не обязательны):

```
$ ls -l > file
```

Пример: запуск команды find(1) с переназначением стандартного потока диагностики (дескриптор 2) в /dev/null (псевдоустройство, запись в которое игнорируется):

```
$ find / -name '*.c' -print 2> /dev/null
```

Что делает dup(2)

Системный вызов dup(2) копирует указатель на системную файловую структуру в таблице дескрипторов файлов в новую ячейку таблицы. Это позволяет двум файловым дескрипторам совместно использовать ту же самую позицию указателя, поскольку они указывают на одну и ту же системную файловую структуру.

Ядро поддерживает счетчик количества файловых дескрипторов, ссылающихся на системную структуру открытого файла. Закрытие каждого из дескрипторов приводит к уменьшению значения этого счетчика. При закрытии последнего дескриптора, счетчик становится равным нулю, и структура данных уничтожается.

Перенаправление ввода/вывода - Пример

Программа переназначает стандартный вывод и стандартный вывод диагностики в указанный файл или устройство, приведенное в качестве первого аргумента командной строки.

В командном процессоре такое перенаправление производится так:

```
prog >file 2>&1
```

или

```
prog 2>file 1>&2
```

Когда командный процессор замечает символ &, за которым следует десятичная цифра, он интерпретирует цифру как номер дескриптора файла и вызывает dup(2).

Библиотечная функция printf(3S) вызывает write(2) с файловым дескриптором 1. printf(3S) эквивалентен fprintf(3S), использующему stdout. Используя stderr в качестве первого аргумента, fprintf вызовет write(2) с дескриптором 2. Программа работает следующим образом:

11-16 Сначала закрывается файловый дескриптор 1. Следующий вызов open(2), если он завершится удачно, возвратит 1 как наименьший свободный дескриптор. Теперь любой вывод в файловый дескриптор 1 будет производиться в заданный файл или устройство.

18-20 Закрывается файловый дескриптор 2. Затем копируется дескриптор 1, возвращая 2, наименьший свободный дескриптор. После этих действий любой вывод в файловый дескриптор 1 или 2 будет производиться в один и тот же файл или устройство. Оба дескриптора указывают на одну и ту же системную файловую структуру с той же самой позицией чтения/записи.

22 printf(3S) вызывает write(2) для дескриптора 1.

23 fprintf(3S) с stderr вызывает write(2) с файловым дескриптором 2.

24_25 Эти два оператора пишут в стандартный вывод и стандартный вывод диагностики соответственно.

Этот пример демонстрируется следующим образом:

```
$ dupdirect /dev/tty
first line to stdout (uses fd 1)
first line to stderr (uses fd 2)
second line to stdout
second line to stderr
```

Если в качестве аргумента указан файл, порядок выходных строк будет иным:

```
first line to stderr (uses fd 2)
second line to stderr
first line to stdout (uses fd 1)
second line to stdout
```

Это связано с тем, что stdout, направленный на терминал, буферизован построчно. Однако, если stdout направлен в регулярный файл, он будет буферизован по блокам. stderr никогда НЕ буферизуется, поэтому весь вывод в этот поток будет выведен сразу, а вывод в sdout — только при заполнении буфера или во время завершения программы. При аварийном завершении программы это может привести к тому, что часть данных, выводимых через stdout, будет потеряна.

Файл: dupdirect.c

ПЕРЕНАПРАВЛЕНИЕ ВВОДА/ВЫВОДА - ПРИМЕР

```
 1 #include <unistd.h>
 2 #include <stdlib.h>
 3 #include <fcntl.h>
 4 #include <stdio.h>
 5
 6 /* demonstrate dup(2) */
 7
 8 main(int argc, char *argv[])
 9 {
10
11     close(1);
12     if (open(argv[1], O_WRONLY |
13     O_CREAT | O_TRUNC, 0644) == -1) {
14     perror(argv[1]);
15     exit(1);
16     }
17
18     close(2);
19     if (dup(1) == -1)
20     exit(2);
21
22     printf("first line to stdout (uses fd 1)\n");
23     fprintf(stderr,"first line to stderr (uses fd 2)\n
24     printf("second line to stdout\n");
25     fprintf(stderr,"second line to stderr\n");
26 }
```

Управление файловым дескриптором

Системный вызов fcntl(2) предназначен для управления открытыми файлами. Он позволяет копировать файловый дескриптор, получить/установить флаги файлового дескриптора, укоротить файл и/или освободить часть места, занимамого файлом, а также для установки захвата участков файла. Аргументы для fcntl(2):

fildes файловый дескриптор, получаемый обычно вызовом open(2).

cmd одна из команд, определенная символьными константами в файле <fcntl.h>. Они будут обсуждены вкратце.

arg fcntl(2) может иметь третий аргумент, тип и значение которого зависит от команды cmd. Тип может быть int или указатель на struct flock.

Команды fcntl(2)

Ниже приведены различные значения cmd:

. Команды, для которых не нужен arg:

F_GETFD возвращает состояние флага закрытия-по-exec для файла fildes. Если младший бит возвращаемого значения равен 0, то файл не будет закрыт при системном вызове exec(2).

F_GETFL возвращает флаги состояния файла. Это позволяет узнать, какие флаги были использованы при открытии файла. Пример этой команды приведен далее в этом разделе.

. Команды для аргумента arg типа int:

F_DUPFD копирует файловый дескриптор. Возвращает файловый дескриптор, значение которого больше или равно arg. Похож на системный вызов dup2(2).

F_SETFD устанавливает флаг закрытия-по-exec для файла fildes в соответствии с младшим битом в arg (0 или 1).

F_SETFL устанавливает флаги состояния файла в соответствии со значением arg. Можно установить только флаги O_NDELAY, O_NONBLOCK, O_APPEND и O_SYNC. Пример этой команды приведён ниже в данном разделе.

. Команды, использующие struct flock * arg:

F_FREESP освобождает место, занимаемое обычным файлом. Пример этой команды приведен ниже в данном разделе.

F_GETLK обсуждается в разделе, посвящённом захвату записей.

F_SETLK обсуждается в разделе, посвящённом захвату записей.

F_SETLKW обсуждается в разделе, посвящённом захвату записей.

Обратите внимание, что при помощи команды F_SETFL нельзя изменить флаги O_RDONLY и O_WRONLY, то есть нельзя «переоткрыть» для записи файл, который был открыт только для чтения.

Чтение с терминала в режиме опроса - Пример: флаг O_NDELAY

Рассмотрим ситуацию, когда ввод данных с терминала должен быть произведен в течении определенного периода времени. Например, автоматическая машина-инспектор требует ввода вашего идентификатора в течение пяти секунд. Программа работает следующим образом:

11-15 Открывается специальный файл, связанный с терминалом, с использованием флага O_NDELAY. Это повлияет на последующее чтение из файла. Если буфер терминала не содержит введенных данных, то функции read(2) вернется с нулем байтов. Если взамен был использован флаг O_NONBLOCK, то read(2) вернет -1 и установит errno в EAGAIN.

Замечание: /dev/tty — это псевдоустройство, которое всегда отображено на ваше настоящее терминальное устройство (управляющий терминал вашей сессии). Использование /dev/tty упрощает разработку программ, которые обязательно должны прочитать данные именно с терминала, даже если стандартные потоки ввода и вывода были переназначены. Это может быть необходимо, например, для ввода пароля.

16-21 Пользователь должен ввести ответ в течение пяти секунд. Программа "заснет" на пять секунд, в течение которых ответ должен быть введен. Ввод завершается нажатием <RETURN>.

Данные хранятся в буфере до тех пор, пока программа не их не прочитает. Если ввода не произошло, программа просыпается, ничего не может прочесть и завершается с сообщением.

В реальных программах чтение данных в режиме опроса использовать нежелательно, так как такая программа всегда будет спать 5 секунд, даже если пользователь введет данные раньше. Для ограничения времени ожидания ввода следует использовать select(3C), poll(2) или сигналы. select(3C) и poll(2) рассматриваются в разделе «Мультиплексирование ввода/вывода», сигналы — в разделе «Сигналы».

23-25 fcntl(2) используется для получения и модификации флагов состояния файла. Флаг O_NDELAY отключается, так что терминал будет ждать, пока ввод не будет завершен нажатием клавиши <RETURN>.

26-27 После печати сообщения программа ожидает ввода.

При использовании флага O_NDELAY нет возможности отличить состояние отсутствия ввода от конца ввода. В обоих случаях read(2) вернёт нуль. Если это нежелательно, можно использовать O_NONBLOCK и анализировать errno.

Файл: opentty.c

ОТКРЫТИЕ ТЕРМИНАЛЬНОГО ФАЙЛА - ПРИМЕР
ФЛАГ O_NDELAY

```
 1 #include <stdio.h>
 2 #include <fcntl.h>
 3 #include <unistd.h>
 4 #include <stdlib.h>
 5
 6 main()
 7 {
 8     int fd, flags;
 9     char line[BUFSIZ];
10
11     if ((fd = open("/dev/tty", O_RDONLY |
12     O_NDELAY)) == -1) {
13     perror("/dev/tty");
14     exit(2);
15     }
16     printf("Enter your PIN within five seconds:\n> ");
17     sleep(5);
18     if (read(fd, line, BUFSIZ) == 0) {
19     printf("\nSorry\n");
20     exit(1);
21     }
22
23     flags = fcntl(fd, F_GETFL);
24     flags &= ~O_NDELAY;      /* turn off delay flag */
25     fcntl(fd, F_SETFL, flags);
26     printf("Enter your bank account number:\n> ");
27     read(fd, line, BUFSIZ);
28
29     /*
30      * .... data processing is performed here ....
31      */
32 }
```

Освобождение пространства на диске

Команда F_FREESP вызова fcntl(2) освобождает место на диске, укорачивая файл или превращая файл в разреженный (понятие разреженного файла обсуждалось в этом разделе при описании системного вызова lseek(2)).

Следующая страница описывает структуру flock, используемую вызовом fcntl(2). Адрес такой структуры передается в качестве третьего аргумента при использовании команд F_FREESP, F_GETLK, F_SETLK или F_SETLKW. С командой F_FREESP используются только поля l_whence, l_start, l_len.

Если l_whence равно SEEK_SET (символьная константа 0), l_len байт освобождается, начиная с l_start.

Если l_whence равно SEEK_CUR (символьная константа 1), l_len байт освобождается, начиная с текущей позиции плюс l_start.

Если l_whence равно SEEK_END (символьная константа 0), l_len байт освобождается, начиная с конца файла плюс l_start.

Поле l_whence напоминает аргумент whence функции lseek(2). Также, l_start напоминает параметр offset. При l_whence, равном SEEK_CUR или SEEK_END, l_start может быть отрицательным.

Если l_len равен нулю, место освобождается от указанной точки до конца файла.

Если освобождаемый участок захватывает конец файла, то длина файла уменьшается. Если освобождаемый участок не доходит до конца файла, то образуется «дырка» в файле, то есть файл становится разреженным, но его логическая длина не изменяется.

Освобождение пространства на диске - Пример

Функция, урезающая файл, приведенная на следующей странице, напоминает библиотечную функцию truncate(3), описанную позже в этом курсе.

Программа работает следующим образом:

8-15 Для отладки функции урезания, компилируйте программу следующим

образом:

```
$ cc -DDEBUG truncate.c
```

При этом строки между #if и #endif останутся в программе.

17 Функция урезания получает два аргумента, path и len. Файл, на имя которого указывает path, укорачивается до len байт.

19 Объявляется структура flock. Адрес lck передается третьим аргументом вызову fcntl(2).

22-25 Открывается файл.

27-29 Полям структуры lck присваиваются соответствующие значения. Поскольку l_whence равно 0, l_start будет относительно начала файла. Пространство будет освобождено, начиная от l_start байт от начала файла и до конца файла (поскольку l_len равно 0).

31-35 fcntl(2) вызывается с командой F_FREESP. Третий аргумент — адрес структуры flock. Файл будет усечен до длины len. Если файл уже был короче, ничего не произойдет.

Вызов: команда wc -c x выводит количество байт в файле x. Программа truncate используется для урезания файла до 5 байт.

```
$ wc  -c x
  567   x
$ truncate x 5
$ wc  -c x
  5   x
```

Файл: truncate.c

ОСВОБОЖДЕНИЕ ПРОСТРАНСТВА НА НОСИТЕЛЕ - ПРИМЕР

```
 1 #include <sys/types.h>
 2 #include <stdio.h>
 3 #include <unistd.h>
 4 #include <stdlib.h>
 5 #include <fcntl.h>
 6 int trunc(char *, off_t);
 7
 8 #if DEBUG
 9 main(int argc, char *argv[])
10 {
11     off_t len;
12     len = atol(argv[2]);
13     trunc(argv[1], len);
14 }
15 #endif
16
17 int trunc(char *path, off_t len)
18 {
19     struct flock lck;
20     register fd;
21
22     if((fd = open(path, O_WRONLY)) == -1){
23     perror(path);
24     return(-1);
25     }
26
27     lck.l_whence = SEEK_SET; /* offset from beginning of file */
28     lck.l_start = len;
29     lck.l_len = 0L;     /* until the end of the file */
30
31     if(fcntl(fd, F_FREESP, &lck) == -1){
32     perror("fcntl");
33     close(fd);
34     return(-1);
35     }
36     close(fd);
37     return(0);
38 }
```

Отображение файлов на память

Современные версии Unix позволяют отображать ресурсы хранения данных (файлы или устройства) в адресное пространство процесса. Такое отображение осуществляется при помощи средств аппаратной виртуализации памяти (диспетчера памяти). Система устанавливает в дескрипторах всех страниц отображённой памяти бит отсутствия. При первом обращении к такой странице, диспетчер памяти генерирует страничный отказ (исключение отсутствия страницы). Ядро системы перехватывает это исключение , считывает страницу из файла или с устройства, снимает бит отсутствия в дескрипторе страницы и возвращает управление программе. Для пользовательской программы это выглядит так, как будто прочитанные с устройства данные всегда находились в странице, на которую они отображены. Таким образом можно отображать на память не только регулярные файлы, но и устройства, поддерживающие функцию lseek(2).

Некоторые устройства, например, видеоадаптеры, могут реализовать отображение в память без использования механизма страничных отказов. Типичный современный видеоадаптер имеет фрейм-буфер, отображенный в физическое адресное пространство компьютера. Запись в этот буфер приводит к изменению цвета и яркости точек («пикселов») на дисплее. При отображении фрейм-буфера в адресное пространство процесса, система отображает страницы виртуальной памяти на физические адреса, соответствующие адресам, на которые отображен фрейм-буфер. В результате, процесс получает прямой доступ к фрейм-буферу.

Поскольку данные считываются в память при первом обращении, отображение файлов на память может быть полезно, если программа заранее не знает, какие из участков файла ей понадобятся. При этом прочитаны будут только те участки файла, которые нужны. Кроме того, чтение данных происходит по мере работы, что может сократить задержки, наблюдаемые пользователем.

Главное использование отображения файлов на память в системах семейства Unix — это загрузка программ. В действительности, код и данные из исполняемых модулей и динамических библиотек не считываются в память при загрузке программы, а только отображаются на память. Участки кода программы или функции библиотек, которые не использовались при данном прогоне программы, могут быть не считаны в память. Это ускоряет время загрузки программ, но иногда может приводить к неожиданным задержкам в процессе исполнения.

Поскольку при отображении файла на память система обычно сама выбирает адрес для отображения, один и тот же файл может быть отображён в разных процессах на разные виртуальные адреса. Чтобы обеспечить разделение сегментов кода разделяемых библиотек, код этих библиотек рекомендуется собирать с ключом -fPIC. Этот ключ заставляет компилятор генерировать позиционно-независимый код.

Для отображения файла на память, файл должен по-прежнему открываться вызовом open(2) и закрываться вызовом close(2), однако read(2) и write(2) можно уже не использовать. После отображения функцией mmap(2), к содержимому файла можно обращаться, как к оперативной памяти.

Закрытие дескриптора файла при помощи close(2) и снятие отображения при помощи munmap(2) могут выполняться в любом порядке; закрытие дескриптора не мешает работать с отображенными данными. Разумеется, система должна сохранять системные структуры данных, необходимые для работы с файлом, все время, пока действует отображение, ведь иначе она не сможет считывать данные из файла. С этой точки зрения, mmap(2) аналогичен дополнительному дескриптору файла, созданному при помощи dup(2) и также учитывается в счетчике ссылок на системную структуру данных.

Отображение файла на память

Системный вызов mmap(2) можно использовать для установления отображения между адресным пространством процесса и файлом или запоминающим периферийным устройством. Это позволяет получать доступ к содержимому файла или устройства как к массиву байт в адресном пространстве процесса.

Для отображения не требуется и не следует предварительно выделять память функцией malloc(3C) или каким-либо другим способом. Вызов mmap(2) сам выделяет необходимое виртуальное адресное пространство. В действительности, функция malloc(3C), возможно, сама использует mmap(2) для того, чтобы запросить память у операционной системы. Поскольку память не может быть отображена одновременно на два разных файла, не следует пытаться отображать файлы на память, выделенную через malloc(3C).

mmap(2) возвращает начальный адрес отображённой области памяти в пределах адресного пространства вашего процесса. Далее этой памятью можно манипулировать, как любой другой памятью. mmap(2) позволяет процессу отобразить в память весь файл или его часть. Хотя mmap позволяет задавать начало и длину отображаемого участка с точностью до байта, в действительности отображение происходит страницами. Начало отображаемого участка файла должно быть кратно размеру страницы (или, что то же самое, выровнено на размер страницы). Длина отображаемого участка может быть не кратна размеру страницы, но mmap(2) округляет его вверх до значения, кратного этому размеру.

Размер страницы зависит от типа диспетчера памяти, а у некоторых диспетчеров также от настроек, определяемых ядром системы. Размер страницы или, точнее, то, что данная версия Unix в данном случае считает размером страницы, можно определить системным вызовом getpagesize(2) или вызовом sysconf(2) с параметром _SC_PAGESIZE.

Оставшаяся часть раздела обсуждает отображение обычных файлов. mmap(2) позволяет отображать и устройства, при условии, что драйвер устройства поддерживает отображение памяти. Например:

. Отображение псевдоустройства /dev/zero выделяет вызывающей программе заполненный нулями блок виртуальной памяти указанного размера. Это может быть альтернативой увеличению границы выделяемой памяти при помощи sbrk(2). Псевдоустройство /dev/zero представляет собой виртуальный файл бесконечной длины, заполненный нулями.

. Отображение фрейм-буфера графического устройства позволяет программе трактовать экран устройства как массив памяти.

В современных Unix-системах, например в Solaris и Linux, можно отображать «анонимные» участки памяти. Это достигается вызовом mmap(2) со значением -1 вместо дескриптора файла и флагом MAP_ANON. При первом обращении к такой странице, система выдает процессу новую страницу памяти, заполненную нулями, поэтому иногда такое отображение описывают как эквивалент отображения файла /dev/zero.

Параметры mmap(2)

addr используется для указания рекомендуемого адреса, по которому будет размещено отображение. Каким образом система располагает окончательный адрес отображения (pa) вблизи от addr, зависит от реализации. Нулевое значение addr дает системе полную свободу в выборе pa. В рамках нашего курса мы не изучаем сведений, необходимых для выбора addr, поэтому рекомендуется использовать нулевое значение.

len Длина отображаемого участка в байтах. Отображение будет размещено в диапазоне [pa, pa+len-1]. mmap(2) выделяет память страницами. То есть, при запросе отображения части страницы, будет отображена вся страница, покрывающая указанные байты.

prot Параметр prot определяет права доступа на чтение, запись, исполнение или их комбинацию с помощью побитового ИЛИ для отображаемых страниц. Соответствующие символьные константы определены в <sys/mman.h>:

PROT_READ страницу можно читать

PROT_WRITE страницу можно изменять

PROT_EXEC страницу можно исполнять.

mprotect(2) можно использовать для изменения прав доступа к отображаемой памяти

flags Символьные константы для этого параметра определены в <sys/mman.h>:

MAP_SHARED Если определен этот флаг, запись в память вызовет изменение отображенного объекта. Иными словами, если процесс изменяет память, отображенную с флагом MAP_SHARED, эти изменения будут сохранены в файле и доступны остальным процессам. Чтобы отобразить файл с PROT_WRITE в режиме MAP_SHARED, файл должен быть открыт на запись.

MAP_PRIVATE При указании этого флага, первое изменение отображенного объекта вызовет создание отдельной копии объекта и переназначит запись в эту копию. До первой операции записи эта копия не создается. Все изменения объекта, отображенного с флагом MAP_PRIVATE, производятся не над самим объектом, а над его копией. Измененные данные не сохраняются в файл, поэтому отображение файла с PROT_WRITE в режиме MAP_PRIVATE не требует ни открытия файла на запись, ни права записи в этот файл.

Либо MAP_SHARED, либо MAP_PRIVATE, но не оба, должны быть указаны.

MAP_ANON Отображение «анонимной» памяти, не привязанной ни к какому файлу. В соответствии с mmap(2), это эквивалентно отображению /dev/zero без флага MAP_ANON.

fd Файловый дескриптор отображаемого файла/устройства или -1 в сочетании с MAP_ANON.

off Отступ от начала файла, с которого начинается отображение.

Доступ к файлу

Эти примеры показывают два способа изменения байтов в начале файла, представляющих 32-битное целое значение.

. Традиционный подход

Открывается файл. open(2) возвращает файловый дескриптор. read(2) считывает байты, представляющие целое значение и сохраняет его в переменной count. Значение увеличивается на 10. Затем новое значение записывается в начало файла, для этого позиция чтения/записи сдвигается на величину, соответствующую размеру целого значения в байтах, с тем чтобы значение могло быть записано в то же самое место файла.

. Подход с отображением файла в память

Открывается файл. open(2) возвращает файловый дескриптор. Затем мы определяем длину файла при помощи вызова lseek(fd, 0, SEEK_END); такая форма вызова возвращает положение конца файла (для определения длины файла рекомендуется использовать вызовы stat(2) или fstat(2), но мы эти вызовы будем проходить далее). Весь файл отображается в память - off равен 0, sbuf.st_size равняется размеру файла в байтах. Система выбирает адрес отображения. Адрес, возвращаемый mmap(2), преобразуется в указатель на целое и присваивается ра. Затем содержимое файла изменяется прямой записью в память. Первое же целое значение, хранящееся в отображенной области памяти, увеличивается на 10.

ДОСТУП К ФАЙЛУ

. традиционный подход

```
fd = open("testfile", O_RDWR);
read(fd, &count, sizeof(count);
count += 10;
lseek(fd, -sizeof(count), SEEK_CUR);
write(fd, &count, sizeof(count));
```

. подход с отображением файла в память

```
fd = open("testfile", O_RDWR);
size = lseek(fd, 0, SEEK_END);
pa = (int *)mmap(0, size,
   PROT_READ|PROT_WRITE,
   MAP_SHARED, fd, 0);
*pa += 10;
```

Удаление отображения страниц памяти

Системный вызов munmap(2) удаляет отображение страниц в диапазоне [addr, addr+len-1]. Последующее использование этих страниц выразится в посылке процессу сигнала SIGSEGV. Границы освобождаемого сегмента не обязаны совпадать с границами ранее отображенного сегмента, но надо иметь в виду, что munmap(2) выравнивает границы освобождаемого сегмента на границы страниц.

Также, неявное удаление отображения для всех сегментов памяти процесса происходит при завершении процесса и при вызове exec(2).

Синхронизация памяти с физическим носителем

Память, отображённая с флагом MAP_SHARED, не сразу записывается на диск. Если вам нужно гарантировать, чтобы изменения оказались на диске, например, чтобы защититься от их потери при аварийном выключении компьютера, следует использовать функцию msync(3C). Также, если вы не модифицировали страницу памяти, но имеете основания предполагать, что она была изменена в файле, при помощи msync(3C) вы можете запросить считывание нового содержимого страницы из файла.

Библиотечная функция msync(3C) записывает все измененные страницы в диапазоне [addr, addr+len-1] на их постоянное место на физическом носителе. Флаги могут иметь следующие значения:

MS_ASYNC немедленно вернуться, как только спланированы все операции записи

MS_SYNC вернуться, только когда завершатся все операции записи

MS_INVALIDATE помечает страницы памяти как недействительные. После этого любое обращение к этим адресам вызывает чтение с постоянного физического носителя.

msync(3) похож на fsync(2) в том смысле, что он сам по себе не вносит никаких изменений в данные, но дожидается их физической записи на диск. Разница заключается в том, что fsync(2) ожидает завершения всех запланированных операций, в то время как msync(3) учитывает только операции записи данные в указанном диапазоне.

Вызов msync(addr, len, flags) эквивалентен memcntl(addr, len, MC_SYNC, flags, 0, 0)

Отображение файла - Пример

Пример на следующей странице ищет в файле записей о служащих необходимую запись и позволяет пользователю изменять значение зарплаты служащего. Пример использует отображение файла в память и работает следующим образом:

16-19 Открывается файл записей о служащих.

21 Используется вызов lseek(2) для получения длины файла.

22 Файл отображается в память. Адрес, возвращаемый mmap(2), преобразуется в struct employy * и присваивается переменной p.

26-27 Пользователь должен ввести номер записи. Нумерация начинается с 1.

28-34 Если пользователь вводит номер записи меньший или равный 0, цикл прекращается. Указание номера за пределами файла вызывает печать сообщения об ошибке и требование повторить ввод.

35-36 Печатаются поля записи.

38-39 Пользователь вводит новое значение зарплаты.

40 msync(2) возвращается только после записи в файл.

42-43 Удаляются отображения в память и файл закрывается.

Файл: update1.c

ОТОБРАЖЕНИЕ ФАЙЛА - ПРИМЕР

```
. . .
10 main(int argc, char *argv[])
11 {
12    off_t size;
13    struct employee *p;
. . .
16    if ((fd = open(argv[1], O_RDWR)) == -1) {
17     perror(argv[1]);
18     exit(1);
19    }
20
21    size = lseek(fd, 0, SEEK_END);
22    p = (struct employee *)mmap(0, size,
23       PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
24
25    for(;;) {
26        printf("\nEnter record number: ");
27        scanf("%d", &recnum);
28        recnum--;
29        if (recnum < 0)
30        break;
31        if (recnum * sizeof(struct employee) >= size) {
32        printf("record %d not found\n", recnum+1);
33        continue;
34        }
35        printf("Employee: %s, salary: %d\n",
36         p[recnum].name, p[recnum].salary);
37
38        printf("Enter new salary: ");
39        scanf("%d", &p[recnum].salary);
40        msync(p, size, MS_SYNC);
41   }
42   munmap(p, size);
43   close(fd);
44 }
```

Приложение - Стандартная библиотека ввода/вывода

Обзор стандартных библиотечных функций ввода/вывода

Эти функции библиотеки языка С автоматически подключаются при компиляции програм на C/C++. Не требуется никаких указания в командной строке. Следует включить <stdio.h> при использовании этих функций.

Функции ввода/вывода разделены на следующие категории:

. Доступ к файлам

. Состояние файла

. Ввод

. Вывод

Функции доступа к файлам

ФУНКЦИЯ	СТРАНИЦА РУКОВОДСТВА	КРАТКОЕ ОПИСАНИЕ
fclose	fclose(3S)	Закрывает открытый поток.
fdopen	fopen(3S)	Связывает поток с файлом, открытым при помощи open(2).
fileno	ferror(3S)	Выдает файловый дескриптор, связанный с открытым потоком.
fopen	fopen(3S)	Открывает файл с указанными правами доступа. fopen возвращает указатель на поток, который используется при последующих операциях с файлом.
freopen	fopen(3S)	Замещает указанным файлом открытый поток.
fseek	fseek(3S)	Перемещает указатель файла.
pclose	popen(3S)	Закрывает поток, открытый при помощи popen.
popen	popen(3S)	Создает программный канал, как поток между вызывающим процессом и командой.
rewind	fseek(3S)	Перемещает указатель файла на начало файла.
setbuf	setbuf(3S)	Назначает потоку буферизацию.
setvbuf	setbuf(3S)	То же, что и setbuf, но с более тонким управлением.

Функции состояния файла

ФУНКЦИЯ	СТРАНИЦА РУКОВОДСТВА	КРАТКОЕ ОПИСАНИЕ
clearerr	ferror(3S)	Сбрасывает состояние ошибки в потоке.
feof	ferror(3S)	Проверяет на конец файла в потоке.
ferror	ferror(3S)	Проверяет на состояние ошибки в потоке.
ftell	fseek(3S)	Выдает текущую позицию в файле.

Функции ввода

ФУНКЦИЯ	СТРАНИЦА РУКОВОДСТВА	КРАТКОЕ ОПИСАНИЕ
fgetc	getc(3S)	Чтение одиночного символа. В отличие от getc(3S), это функция а не препроцессорный макрос.
fgets	gets(3S)	Читает строку из потока.
fread	fread(3S)	Осуществляет ввод блока данных указанного размера.
fscanf	scanf(3S)	Осуществляет форматированный ввод из потока.
getc	getc(3S)	Читает символ из потока.
getchar	getc(3S)	Читает символ из стандартного ввода.
gets	gets(3S)	Читает строку из стандартного ввода. Не рекомендуется использовать, так как этой функции не передается размер буфера, поэтому велика опасность срыва буфера.
getw	getc(3S)	Читает слово из потока.
scanf	scanf(3S)	Осуществляет форматированный ввод из стандартного ввода.
sscanf	scanf(3S)	Осуществляет форматированный ввод из строки.
ungetc	ungetc(3S)	Возвращает символ в поток. Эта функция полезна при реализации лексических анализаторов с просмотром на один символ вперёд.
copylist	copylist(3G)	Копирует файл в память.

Функции вывода

ФУНКЦИЯ	СТРАНИЦА РУКОВОДСТВА	КРАТКОЕ ОПИСАНИЕ

fflush	fclose(3S)	Выводит все символы из буфера в файловый дескриптор.
fprintf	printf(3S)	Осуществляет форматированный вывод в поток.
fputc	putc(3S)	Подлинная функция для putc(3S).
fputs	puts(3S)	Осуществляет вывод строки.
fwrite	fread(3S)	Выводит в поток блок данных фиксированного размера.
printf	printf(3S)	Осуществляет форматированный вывод в стандартный вывод.
putc	putc(3S)	Выводит символ в стандартный вывод.
putchar	putc(3S)	Выводит символ в стандартный вывод.
puts	puts(3S)	Выводит строку в стандартный вывод.
putw	putc(3S)	Выводит слово в поток.
sprintf	printf(3S)	Осуществляет форматированный вывод в строку.
vprintf	vprintf(3C)	То же, что и printf(3C), но с использованием переменного числа аргументов varargs(5).
vfprintf	vprintf(3C)	То же, что и fprintf(3C), но с использованием переменного числа аргументов varargs(5).
vsprintf	vprintf(3C)	То же, что и sprintf(3C), но с использованием переменного числа аргументов varargs(5).

3. ЗАХВАТ ФАЙЛОВ И ЗАПИСЕЙ

Обзор

В этом разделе описываются средства захвата файлов и записей в ОС UNIX. Часто встречается ситуация, когда несколько процессов могут пытаться читать и изменять данные в одном и том же файле. Нужен метод, чтобы правильно управлять этими изменениями. Иначе данные в файле могут быть испорчены и программы, использующие эти данные, выдадут неправильный результат. Например, рассмотрим систему, которая управляет продажей акций. Такая система не должна допускать продажи одной и той же акции двумя разными продавцами.

Предположим, что первый продавец получил информацию об акции. Второй продавец также сможет получить информацию об этой акции, но он не сможет изменить её, пока первый продавец не закончит работу с этой информацией. Другой метод состоит в том, чтобы закрыть второму продавцу доступ к информации об акции, пока первый продавец не закончит работу с этой информацией.

В первом случае, запись, содержащая информацию об акции, захвачена по чтению (read lock). Это не даёт другим процессам возможности захватить эту запись по изменению. Захват по чтению предоставляет разделяемый доступ к записи. Во втором случае, запись захвачена по изменению (write lock). Это не дает другим процессам возможности захватить эту запись ни по чтению, ни по изменению. Первый продавец имеет эксклюзивный (exclusive) доступ к информации в записи.

В ОС семейства CP/M (OS/2, Win32, Win64) при открытии файла по умолчанию устанавливается блокировка на весь файл. Это нужно для обеспечения совместимости с программами для CP/M и MS/DR-DOS, которые разрабатывались в расчёте на однозадачную ОС и не предполагают, что в системе могут существовать другие процессы, кроме них.

В системах семейства Unix, предполагается, что программа с самого начала разрабатывается в расчёте на многозадачную среду. Поэтому файл при открытии автоматически не блокируется. Программист, который считает необходимым использование блокировок при работе с файлом, должен устанавливать эти блокировки явным образом. API для работы с блокировками и будет изучаться в этом разделе.

Что такое захват записи и файла?

Захват (блокировка) записей в ОС UNIX - это базовое средство синхронизации процессов, осуществляющих доступ к одной и той же записи в файле, а не средство обеспечения безопасности. Захват записи осуществляется системным вызовом fcntl(2).

При захвате записи используется следующая терминология:

. Запись (record) - это последовательный набор байтов в файле. У записи есть начальная позиция в файле и длина, отсчитываемая от этой позиции. Начало и длина указываются с точностью до байта. Ядро Unix не делает никаких предположений о том, как эта запись интерпретируется прикладной программой. Одна запись с точки зрения Unix, с точки зрения приложения может представлять собой одну или несколько записей базы данных или часть такой записи или текстовую строку или что-то ещё. Если захваченная запись начинается с начала файла и равна файлу по длине, иногда говорят о захвате всего файла.

. Захват записи по чтению (разделяемый доступ) не даёт другим процессам установить захват записи по изменению, но позволяет неограниченному количеству процессов устанавливать захваты по чтению.

. Захват записи по изменению (эксклюзивный доступ) не даёт другим процессам установить захват записи ни по чтению, ни по изменению, пока этот захват по изменению не будет снят.

. Конфликтующие захваты — пара захватов, несовместимых друг с другом (два захвата по изменению или один захват по чтению и один по изменению) и при этом устанавливаемых на перекрывающиеся участки файла. Перекрытия на один байт достаточно для возникновения конфликта.

. Рекомендательный (advisory) режим захвата не взаимодействует с подсистемой ввода/вывода. Это означает, что захват проверяется только при попытках захвата, но не при операциях чтения и записи. Как и семафоры и мутексы при работе с памятью, рекомендательный захват указывает другим процессам, что им не следует работать с захваченным участком, но не ставит физических препятствий.

. Обязательный (mandatory) режим захвата взаимодействует с подсистемой ввода/вывода. Это означает, что системные вызовы read(2) или write(2) будут приостанавливаться (неявный захват), пока не будет снят захват соответствующей записи.

Тип захвата (по чтению или по модификации) определяется параметрами fcntl(2), точнее, полями в структуре flock. Режим захвата (рекомендательный или обязательный) определяется атрибутами файла. По умолчанию, в Unix используется рекомендательный захват. Чтобы установить обязательный захват, файлу необходимо установить бит setgid, но при этом файл не должен быть доступен по исполнению для группы. Все захваты, устанавливаемые на такой файл, будут работать в обязательном режиме.

Внимание: NFSv3 поддерживает стандартный API для работы с рекомендательными блокировками, но обязательные блокировки через NFS не работают. В зависимости от версии сервера NFS, установка режима обязательной блокировки может либо игнорироваться, либо приводить к полной недоступности файла (все обращения через NFS выдают ошибку доступа).

Ядро обеспечивает определенную степень обнаружения и предотвращения взаимных блокировок (deadlock). При установке захвата осуществляется проверка, что не создается цикл процессов, каждый из которых ждет от других снятия захвата записи/файла. Если такая ситуация обнаружена, fcntl(2) немедленно возвращает неуспех и устанавливает errno равным EDEADLK.

Установка и снятие захвата

Системный вызов fcntl(2) используется для захвата файла и записи и для снятия захвата. Его аргументы:

fildes дескриптор файла, обычно получают с помощью системного вызова open(2).

cmd определяет одну из трех команд, используемых при захвате:

F_SETLK Эта команда используется, чтобы установить или снять захват записи. Структура захвата используется для задания расположения записи, ее длины и типа захвата. Если захват нельзя установить из-за конфликтующего захвата или по какой-то другой причине, fcntl вернет -1 и установит errno.

F_SETLKW Эта команда аналогична F_SETLK, кроме того что, если на указанный участок установлена конфликтующая блокировка, fcntl приостановится, пока запись не освободится.

F_GETLK Эта команда используется, чтобы получить информацию о захвате записи. Чтобы определить расположение записи и тип захвата используется структура захвата flock. F_GETLK выдает информацию о первом захвате записи, перекрывающейся с участком файла, описанным в структуре flock. Возвращаемая информация затирает информацию, переданную вызову fcntl через эту структуру. Если нет конфликтующих захватов, структура возвращается назад неизмененной, кроме типа захвата, который устанавливается равным F_UNLCK. Как сказано в системном руководстве, F_GETLK возвращает какой-то из захватов, который был установлен на некоторый момент времени. Это означает, что при интерпретации информации, полученной через F_GETLK надо учитывать, что с момента вызова F_GETLK на файл могли быть установлены новые блокировки или сняты какие-то из существовавших, в том числе и та блокировка, информацию о которой вернул этот вызов.

arg адрес структуры захвата struct flock. Поля этой структуры описаны на следующей странице.

Как при рекомендательном, так и при принудительном захвате записи, захват выполняется с использованием fcntl(2) или lockf(3C). Различие между двумя формами захвата состоит в том, когда он проверяется. При принудительном захватывании он проверяется перед каждой операцией ввода/вывода. При рекомендательном захватывании он проверяется, когда делается попытка захвата с помощью fcntl(2) или lockf(3). Захваты, которые не сняты явно с использованием fcntl(2), снимаются при завершении процесса с помощью exit(2) или при закрытии файла с помощью close(2).

Захват записи

Чтобы выполнить захват записи, ваша программа должна объявить переменную типа struct flock или выделить память под такую структуру другим способом (например, через malloc(3C)), присвоить значение полям этой структуры и передать её адрес системному вызову fcntl в качестве третьего аргумента.

Поля struct flock:

l_type указывает тип захвата. Его возможные значения:

F_RDLCK захват по чтению

F_WRLCK захват по изменению

F_UNLCK снятие захвата. Только процесс, который захватил запись, может освободить её. Освобождение сегмента из середины большой записи оставляет два захваченных сегмента с двух концов.

l_whence является признаком относительной стартовой позиции записи: SEEK_SET (0) - от начала файла, SEEK_CUR (1) - от текущей позиции в файле и SEEK_END (2) - от конца файла. Это поле аналогично аргументу whence в системном вызове lseek(2).

l_start определяет начальную позицию записи в зависимости от l_whence. Это поле аналогично аргументу offset вызова lseek(2). При l_whence, равном SEEK_CUR или SEEK_END, l_start может быть отрицательным.

l_len определяет длину захватываемой записи. Значение ноль захватывает/освобождает от начальной позиции до конца файла. Если после такого захвата длина файла увеличится, все новые данные, появляющиеся в файле, также будут захвачены.

l_pid устанавливается операционной системой равным идентификатору процесса, который захватил запись, когда выполняется запрос F_GETLK. При установке захвата не используется.

l_sysid устанавливается при запросе F_GETLK равным RFS-идентификатору удалённой системы, на которой размещён этот файл.

Печать отчёта - Пример

Этот и другие примеры в этом разделе читают и изменяют файл данных о служащих. Захваты по записи и изменению используются, чтобы запретить нескольким процессам изменять одну и ту же запись в одно и то же время.

Запись о служащем выглядит следующим образом:

employee.h:

```
 1 #define   NAMESIZE   24
 2
 3 struct employee {
 4     char name[NAMESIZE];
 5     int salary;
 6     unsigned short uid;
 7 };
```

Она содержит имя служащего, его оклад и идентификатор пользователя, который последним изменил эту запись.

Эта программа выводит содержание файла служащих. Весь файл захватывается по чтению, чтобы не допустить изменение любой записи при печатании отчёта. Это гарантирует, что отчёт будет точным отражением текущего состояния файла служащих. Захват всего файла не причинит неудобства, если отчёт печатается тогда, когда файл служащих не используется никем другим.

Код, показанный на следующей странице, содержит только захват и освобождение записи. Вся программа приведена в конце раздела. Программа работает следующим образом:

... Файл служащих, полученный как первый аргумент командной строки, открывается для чтения.

18-22 Полям структуры захвата присваиваются значения, соответствующие выполнению захвата по чтению.

23-33 Цикл пытается пять раз захватить файл. fcntl завершается неудачно, если где-либо в файле уже есть захват по записи. Если программа так и не смогла захватить файл, печатается сообщение об ошибке, рекомендующее пользователю исполнить программу позже.

... Отсутствующий код печатает записи о служащих, а также вычисляет и распечатывает суммарный оклад.

42-44 Перед завершением программы захват по чтению снимается. В данном случае это не обязательно, ведь захват будет неявно снят при exit(2). Но это может оказаться полезно, если затем в программу будет добавлена какая-то другая функциональность, исполняемая после завершения работы с базой данных.

Файл: report1.c

ПЕЧАТЬ ОТЧЕТА - ПРИМЕР
ЗАХВАТ ФАЙЛА ПО ЧТЕНИЮ

```
 1 #include <sys/types.h>
 2 #include <unistd.h>
 3 #include <stdlib.h>
 4 #include <stdio.h>
 5 #include <fcntl.h>
 6 #include <errno.h>
 7 #include "employee.h"
 8 #define   MAXTRIES   5
 9 main(int argc, char *argv[])
10 {
11     struct flock lock;
...
18     lock.l_type = F_RDLCK;
19     lock.l_whence = SEEK_SET;
20     lock.l_start = 0;
21     lock.l_len = 0; /* whole file address space */
22     while (fcntl(fd, F_SETLK, &lock) == -1) {
23         if ((errno == EACCES) || (errno == EAGAIN)) {
24             if (try++ < MAXTRIES) {
25                 sleep(1);
26                 continue;
27             }
28             printf("%s busy -- try later\n",
29                 argv[1]);
30             exit(2);
31         }
32         perror(argv[1]); exit(3);
33     }
...
42     lock.l_type = F_UNLCK;    /* unlock file */
43     fcntl(fd, F_SETLK, &lock);
44     close(fd);
...
```

Изменение записи - Пример

В этом примере изменяется запись о служащем. Перед чтением запись захватывается по изменению и освобождается после изменения. Программа работает следующим образом:

... Файл служащих открывается для чтения и записи. После входа в цикл for, программа просит пользователя ввести номер записи.

28-36 Запись захватывается по изменению. Это не дает возможности другим процессам захватить эту запись. fcntl(2) подвиснет, если запись уже захвачена, ожидая пока все другие процессы освободят ее.

38-44 Текущая позиция файла устанавливается на захватываемую запись. Затем запись читается. Замечание: lseek(2) используется явно, чтобы установить текущую позицию в файле. Использование fcntl(2) для захвата записи не приводит к изменению текущей позиции файла.

... Печатается имя и оклад служащего. Затем вводится новый оклад.

51-52 lseek(2) используется снова, чтобы установить позицию файла на захваченную запись. Измененная запись пишется в файл.

54-55 Захват по изменению снимается, чтобы все остальные процессы могли использовать измененную запись.

Эта схема захвата записи работает правильно, если все процессы используют эту программу для изменения файла служащих. Этого можно достигнуть соответствующей установкой прав доступа файла служащих и бита "установки идентификатора пользователя" этой программы. Кроме того, обеспечение такого доступа к файлу служащих делает программу уверенной в том, что запись доступна в каждый момент только одному процессу.

Файл: update1.c

ИЗМЕНЕНИЕ ЗАПИСИ - ПРИМЕР
ЗАХВАТ ЗАПИСИ ПО ИЗМЕНЕНИЮ

```
 1 #include <sys/types.h>
 2 #include <stdio.h>
 3 #include <unistd.h>
 4 #include <fcntl.h>
 5 #include <stdlib.h>
 6 #include <errno.h>
 7 #include "employee.h"
 8
 9 main(int argc, char *argv[])
10 {
11      struct flock lock;
...
28       position = (recnum-1) * sizeof(record);
29       lock.l_type = F_WRLCK;        /* lock record */
30       lock.l_whence = SEEK_SET;
31       lock.l_start = position;
32       lock.l_len = sizeof(record);
33       if (fcntl(fd, F_SETLKW, &lock) == -1) {
34            perror(argv[1]);
35            exit(2);
36       }
37
38       lseek(fd, position, SEEK_SET);      /* read record */
39       if (read(fd, &record, sizeof(record)) == 0) {
40            printf("record %d not found\n", recnum);
41            lock.l_type = F_UNLCK;
42            fcntl(fd, F_SETLK, &lock);
43            continue;
44       }
...
51       lseek(fd, position, SEEK_SET);
52       write(fd, &record, sizeof(record));
53
54       lock.l_type = F_UNLCK;        /* release record */
55       fcntl(fd, F_SETLK, &lock);
...
58 }
```

Изменение записи - Пример отображения на память и захвата записи

Пример с предыдущей страницы переписан здесь, чтобы использовать отображение файлов на память. Файл сначала отображается на память с помощью mmap(2). Каждая запись перед чтением захватывается по изменению и освобождается после изменения. Программа работает следующим образом:

... Файл служащих открывается для чтения и записи.

23-25 Файл служащих отображается на память.

... После входа в цикл for, программа запрашивает у пользователя номер записи.

34-38 Проверка номера записи.

40-47 Запись захватывается по изменению. Это не дает возможности другим процессам тоже захватить эту запись. fcntl(2) будет заблокирована, если запись уже захвачена, в ожидании пока все остальные процессы не освободят её.

... Печатается имя и оклад служащего. Потом вводится новый оклад.

55-59 Содержимое памяти синхронизуется с файлом вызовом msync(3C). Затем захват по изменению записи снимается, и она становится доступной для других процессов.

Обратите внимание, что в данном случае используется синхронизация с флагом MS_ASYNC, которая не дожидается физического завершения записи на диск. В момент снятия блокировки, модифицированные данные ещё могут находиться в очереди на запись в системном дисковом кэше. В данном случае это не проблема, так как другие процессы также работают с файлом через дисковый кэш, и система предоставляет им модифицированные данные, как если бы они уже лежали на диске. Использование флага MS_SYNC необходимо только если вас беспокоит возможность потери данных в случае общесистемного сбоя, когда содержимое дискового кэша будет потеряно.

Эта схема захвата записи работает правильно, если используется рекомендательный захват и все процессы используют эту программу или программу update1.c для изменения файла служащих.

Если в файле установлены атрибуты обязательного захвата, отображение на память работает следующим образом. Если запись отображена на память, установка захвата закончится неудачно (вернет -1). И наоборот, если запись захвачена, её отображение на память закончится неудачно. Таким образом, программа не будет работать, если файл данных имеет атрибуты обязательного захвата записи.

Файл: updatem.c
ИЗМЕНЕНИЕ ЗАПИСИ - ПРИМЕР ОТОБРАЖЕНИЯ НА ПАМЯТЬ И ЗАХВАТА ЗАПИСИ

```
 1 #include <sys/mman.h>
 2 #include <stdlib.h>
 3 #include <stdio.h>
 4 #include <fcntl.h>
 5 #include <sys/types.h>
 6 #include <unistd.h>
 7 #include <errno.h>
 8 #include "employee.h"
 9
10 main(int argc, char *argv[])
11 {
12    struct flock lock;
13    off_t size;
14    struct employee *p;
...
23    size = lseek(fd, 0, SEEK_END);
24    p = (struct employee *)mmap(0, size,
25        PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
...
34        position = recnum * sizeof(struct employee);
35        if (position >= size) {
36        printf("record %d not found\n", recnum);
37        continue;
38        }
39
40        lock.l_type = F_WRLCK;        /* lock record */
41        lock.l_whence = SEEK_SET;
42        lock.l_start = position;
43        lock.l_len = sizeof(struct employee);
44        if (fcntl(fd, F_SETLKW, &lock) == -1) {
45         perror(argv[1]);
46         exit(2);
47        }
...
55        msync(p, size, MS_ASYNC);
56
57        lock.l_type = F_UNLCK;        /* release record */
58        fcntl(fd, F_SETLK, &lock);

...

63 }
```

Изменение записи - Пример

Эта программа является развитием предыдущего примера. После захвата по чтению записи о служащем, информация из неё выводится для пользователя. Потом программа спрашивает пользователя, хочет ли он изменить запись. В случае положительного ответа, захват по чтению переводится в захват по изменению. Потом производится изменение, после чего запись освобождается. Эта программа работает следующим образом:

... Файл служащих открывается для чтения и записи. После входа в цикл for вводится номер записи.

28-36 Запись захватывается по чтению. Процесс подвешивается, если запись уже захвачена по изменению.

... Если попытка прочитать запись неуспешна, то захват по чтению снимается и программа идет на начало цикла. Иначе печатается имя служащего и его оклад.

50-54 Если пользователь не хочет изменять запись, захват по чтению снимается и программа продолжается с начала цикла. Предупреждение: Не забудьте освободить запись перед переходом на начало цикла.

55-59 Захват по чтению переводится в захват по изменению. fcntl(2) блокируется, если есть другие процессы, захватившие эту запись по чтению. После того как все процессы освободят эту запись, наш процесс просыпается и захватывает запись.

... Запись о служащем изменяется.

66-67 Захват по изменению снимается с записи.

Файл: update2.c

ИЗМЕНЕНИЕ ЗАПИСИ - ПРИМЕР
ЗАХВАТ ЗАПИСИ ПО ЧТЕНИЮ И ИЗМЕНЕНИЮ

```
 1 #include <sys/types.h>
 2 #include <stdlib.h>
 3 #include <stdio.h>
 4 #include <fcntl.h>
 5 #include <unistd.h>
 6 #include <errno.h>
 7 #include "employee.h"
 8
 9 main(int argc, char *argv[])
10 {
11      struct flock lock;
...
28       position = (recnum-1) * sizeof(record);
29       lock.l_type = F_RDLCK;          /* read lock */
30       lock.l_whence = SEEK_SET;
31       lock.l_start = position;
32       lock.l_len = sizeof(record);
33       if (fcntl(fd, F_SETLKW, &lock) == -1) {
34            perror(argv[1]);
35            exit(2);
36       }
...
50       if (ans[0] != 'y') {
51            lock.l_type = F_UNLCK;
52            fcntl(fd, F_SETLK, &lock);
53            continue;
54       }
55       lock.l_type = F_WRLCK;          /* write lock */
56       if (fcntl(fd, F_SETLKW, &lock) == -1) {
57            perror(argv[1]);
58            exit(3);
59       }
...
66       lock.l_type = F_UNLCK; /* release record */
67       fcntl(fd, F_SETLK, &lock);
...
70 }
```

Выдача информации о захватах записи - Пример

Так как несколько пользователей могут изменять различные записи в файле, может быть необходимо выдать отчет о захваченных записях. Например, полезно знать, что какие-то записи в файле захватываются слишком часто в одной и той же программе. Тогда эту программу можно исправить.

Программа, приведенная на следующей странице, печатает список всех захваченных записей, тип захвата и идентификатор процесса, захватившего запись. Программа проанализирует на захват все записи в файле. Для этого используется системный вызов fcntl(2) с запросом F_GETLK. Тип захвата должен быть равен F_WRLCK, потому что такой запрос конфликтует со всеми другими типами запросов и, таким образом выдаст информацию обо всех установленных захватах. Тип F_RDLCK конфликтует только с захватами на запись и выдаст информацию только о них.

Эта программа работает следующим образом:

12 Описание переменной типа struct flock.

... Файл открывается для чтения и определяется его размер при помощи lseek(2).

22 Записи проверяются с начала файла.

25-26 Этот цикл for проверяет каждую запись в файле.

27-30 Тип захвата запроса – это захват по изменению, но fcntl(2) при возврате изменяет поле l_type на тип захвата, установленного на проверяемой записи. Поле длины записи (l_len) и ее начальная позиция (l_start) также могут быть изменены. Именно поэтому l_start используется как переменная цикла в операторе for. Если запись, задаваемая l_start и l_len, свободна, цикл продолжается со следующей записи.

32-33 Печатается тип захвата.

34-35 Печатается номер записи. Заметим, что программа предполагает, что все записи одного и того же размера.

36-39 Если конец файла, начиная с l_start, захвачен, то l_len станет равным нулю. Тогда вычисляется число оставшихся в файле записей.

40-41 Печатается число захваченных записей.

Помните, что l_start и l_len могут измениться при вызове fcntl(2) с F_GETLK. Это происходит, когда расположение существующего захвата или не точно соответствует значениям, переданным fcntl(2) в структуре flock. Эта программа работает следующим образом:

```
$ getlocks data
File: data
Pid: 14585      Write lock      Record number: 1        Number of
records: 1
Pid: 14605      Read lock       Record number: 3        Number of
records: 1
```

Файл: getlocks.c

ВЫДАЧА ИНФОРМАЦИИ О ЗАХВАТАХ ЗАПИСИ - ПРИМЕР

```
 1 #include <sys/types.h>
 2 #include <stdlib.h>
 3 #include <stdio.h>
 4 #include <fcntl.h>
 5 #include <unistd.h>
 6 #include <errno.h>
 7 #include "employee.h"
 8
 9 main(int argc, char *argv[])
10 {
11     struct employee record;
12     struct flock lock;
13     off_t size;
...
22     lock.l_whence = SEEK_SET;
23     lock.l_len = sizeof(record);
24     printf("File: %s\n", argv[1]);
25     for (lock.l_start = 0; lock.l_start < size;
26      lock.l_start += lock.l_len) {
27      lock.l_type = F_WRLCK;
28      fcntl(fd, F_GETLK, &lock);
29      if (lock.l_type == F_UNLCK)     /* not locked */
30          continue;
31      printf("Pid: %ld", lock.l_pid);
32      printf("\t%s lock",
33          (lock.l_type == F_WRLCK)? "Write" : "Read");
34      printf("\tRecord number: %ld",
35          lock.l_start/sizeof(record));
36      if (lock.l_len == 0) {     /* rest of file */
37          lock.l_len = size - lock.l_start;
38          lock.l_start = size;  /* terminate for loop */
39          }
40      printf("\tNumber of records: %ld\n",
41          lock.l_len/sizeof(record));
42     }
43 }
```

Библиотечная функция lockf(3C)

Библиотечная функция lockf(3C) проще, чем fcntl(2). Однако, она не обеспечивает всех функциональных возможностей fcntl(2). В частности, она допускает только захват по изменению.

lockf(3C) не использует структуру flock. Началом записи считается указатель текущей позиции в файле. Поэтому перед использованием lockf(3C) необходимо установить позицию файла в нужное место.

Аргументы lockf(3C):

fildes дескриптор файла, обычно получают с помощью системного вызова open(2).

function функция захвата. Она аналогична команде в fcntl(2). Возможные значения функции обсуждаются на следующей странице.

size размер записи от текущей позиции файла. Отрицательный size говорит о том, что запись ограничивается текущей позицией сзади. Если size равен нулю, захватывается весь остаток файла, от текущей позиции и до конца (даже если конец и будет меняться в дальнейшем).

Функции захвата

Ниже приведены значения аргумента function:

F_UNLOCK освобождает ранее захваченную запись

F_LOCK устанавливает захват записи по изменению. Если запись уже захвачена другим процессом, функция блокируется, пока запись не освободится.

F_TLOCK тоже самое, что F_LOCK, кроме того что, если запись уже захвачена, функция возвращает неуспех и устанавливает errno.

F_TEST проверяет захват записи. Возвращает 0, если запись свободна, и -1, если запись захвачена.

Вызов lockf(3C) можно заменить вызовом fcntl(2) с подходящим значением аргумента cmd и типом захвата l_type равным F_WRLCK или F_UNLCK.

4. Мультиплексирование ввода/вывода и асинхронный ввод/вывод

Мы ждали его слишком долго
Что может быть глупее, чем ждать?
Б. Гребенщиков

Обзор

В ходе этой лекции вы изучите:

- Опрос нескольких устройств ввода-вывода при помощи системных вызовов select и poll
- Использование select/poll для ожидания ввода с тайм-аутом
- Стандартные средства асинхронного ввода/вывода

Мультиплексирование ввода-вывода

Если ваша программа главным образом занимается операциями ввода/вывода, вы можете получить наиболее важные из преимуществ многопоточности в однопоточной программе, используя системный вызов select(3C). В большинстве Unix-систем select является системным вызовом, или, во всяком случае, описывается в секции системного руководства 2 (системные вызовы), т.е. ссылка на него должна была бы выглядеть как select(2), но в Solaris 10 соответствующая страница системного руководства размещена в секции 3C (стандартная библиотека языка C).

Устройства ввода/вывода обычно работают гораздо медленнее центрального процессора, поэтому при выполнении операций с ними процессор обычно оказывается вынужден ждать их. Поэтому во всех ОС системные вызовы синхронного ввода/вывода представляют собой блокирующиеся операции.

Это относится и к сетевым коммуникациям – взаимодействие через Интернет сопряжено с большими задержками и, как правило, происходит через не очень широкий и/или перегруженный канал связи.

Если ваша программа работает с несколькими устройствами ввода/вывода и/или сетевыми соединениями, ей невыгодно блокироваться на операции, связанной с одним из этих устройств, ведь в таком состоянии она может пропустить возможность совершить ввод/вывод с другого устройства без блокировки. Эту проблему можно решать при помощи создания нитей, работающих с различными устройствами. В предыдущих лекциях мы изучили все необходимое для разработки таких программ. Однако для решения этой проблемы есть и другие средства.

Системный вызов select(3C)

В большинстве Unix-систем, select(3C) представляет собой системный вызов, но в Solaris 10 он реализован как библиотечная функция, использующая системный вызов poll(2), поэтому в Solaris руководство по select находится в секции руководства 3C.
select(3C) позволяет ожидать готовности нескольких устройств или сетевых соединений (в действительности, готовности объектов большинства типов, которые могут быть идентифицированы файловым дескриптором). Когда один или несколько из дескрипторов оказываются готовы передать данные, select(3C) возвращает управление программе и передает списки готовых дескрипторов в выходных параметрах.
В качестве параметров select(3C) использует множества (наборы) дескрипторов. В старых Unix-системах множества были реализованы в виде 1024-разрядных битовых масок. В современных Unix-системах и в других ОС, реализующих select, множества реализованы в виде непрозрачного типа fd_set, над которым определены некоторые теоретико-множественные операции, а именно – очистка множества, включение дескриптора в множество, исключение дескриптора из множества и проверка наличия дескриптора в множестве. Препроцессорные директивы для выполнения этих операций описаны на странице руководства select(3C).
В 32-разрядных версиях Unix SVR4, в том числе в Solaris, fd_set по прежнему представляет собой 1024-битовую маску; в 64-разрядных версиях SVR4 это маска разрядности 65536 бит. Размер маски определяет не только максимальное количество файловых дескрипторов в наборе, но и максимальный номер файлового дескриптора в наборе. Размер маски в вашей версии системы можно определить во время компиляции по значению препроцессорного символа FD_SETSIZE. Нумерация файловых дескрипторов в Unix начинается с 0, поэтому максимальный номер дескриптора равен FD_SETSIZE-1.
Таким образом, если вы используете select(3C), вам необходимо установить ограничения на количество дескрипторов вашего процесса. Это может быть сделано шелловской командой ulimit(1) перед запуском процесса или системным вызовом setrlimit(2) уже во время исполнения вашего процесса. Разумеется, setrlimit(2) необходимо вызвать до того, как вы начнете создавать файловые дескрипторы.
Если вам необходимо использовать более 1024 дескрипторов в 32-битной программе, Solaris 10 предоставляет переходный API. Для его использования необходимо определить препроцессорный символ FD_SETSIZE с числовым значением, превышающим 1024, перед включением файла <sys/time.h>. При этом в файле <sys/select.h> сработают необходимые препроцессорные директивы и тип fd_set будет определен как большая битовая маска, а select и другие системные вызовы этого семейства будут переопределены для использования масок такого размера.
В некоторых реализациях fd_set реализован другими средствами, без использования битовых масок. Например, Win32 предоставляет select в составе так называемого Winsock API. В Win32 fd_set реализован как динамический массив, содержащий значения файловых дескрипторов. Поэтому вам не следует полагаться на знание внутренней структуры типа fd_set. Так или иначе, изменения размера битовой маски fd_set или внутреннего представления этого типа требуют перекомпиляции всех программ, использующих select(3C). В будущем, когда архитектурный лимит в 65536 дескрипторов на процесс будет повышен, может потребоваться новая версия реализации fd_set и select и новая перекомпиляция программ. Чтобы избежать этого и упростить переход на новую версию ABI, компания Sun Microsystems рекомендует отказываться от использования select(3C) и использовать вместо него системный вызов poll(2). Системный вызов poll(2) рассматривается далее на этой лекции.

Select(3C)

Системный вызов select(3C) имеет пять параметров.

int nfds – число, на единицу большее, чем максимальный номер файлового дескриптора во всех множествах, переданных как параметры.

fd_set *readfds – Входной параметр, множество дескрипторов, которые следует проверять на готовность к чтению. Конец файла или закрытие сокета считается частным случаем готовности к чтению. Регулярные файлы всегда считаются готовыми к чтению. Также, если вы хотите проверить слушающий сокет TCP на готовность к выполнению accept(3SOCKET), его следует включить в это множество. Также, выходной параметр, множество дескрипторов, готовых к чтению.

fd_set *writefds – Входной параметр, множество дескрипторов, которые следует проверять на готовность к записи. Ошибка при отложенной записи считается частным случаем готовности к записи. Регулярные файлы всегда готовы к записи. Также, если вы хотите проверить завершение операции асинхронного connect(3SOCKET), сокет следует включить в это множество. Также, выходной параметр, множество дескрипторов, готовых к записи.

fd_set *errorfds – Входной параметр, множество дескрипторов, которые следует проверять на наличие исключительных состояний. Определение исключительного состояния зависит от типа файлового дескриптора. Для сокетов TCP исключительное состояние возникает при приходе внеполосных данных. Регулярные файлы всегда считаются находящимися в исключительном состоянии. Также, выходной параметр, множество дескрипторов, на которых возникли исключительные состояния.

struct timeval * timeout – тайм-аут, временной интервал, задаваемый с точностью до микросекунд. Если этот параметр равен NULL, то select(3C) будет ожидать неограниченное время; если в структуре задан нулевой интервал времени, select(3C) работает в режиме опроса, то есть возвращает управление немедленно, возможно с пустыми наборами дескрипторов.

Вместо любого из параметров типа fd_set * можно передать нулевой указатель. Это означает, что соответствующий класс событий нас не интересует. select(3C) возвращает общее количество готовых дескрипторов во всех множествах при нормальном завершении (в том числе при завершении по тайм-ауту), и -1 при ошибке.

Использование select(3C)

В примере 1 приводится использование select(3C) для копирования данных из сетевого соединения на терминал, а с терминала – в сетевое соединение. Эта программа упрощенная, она предполагает, что запись на терминал и в сетевое соединение никогда не будет заблокирована. Поскольку и терминал, и сетевое соединение имеют внутренние буферы, при небольших потоках данных это обычно так и есть.
Пример 1. Двустороннее копирование данных между терминалом и сетевым соединением. Пример взят из книги У.Р. Стивенс, Unix: разработка сетевых приложений. Вместо стандартных системных вызовов используются «обертки», описанные в файле “unp.h”

```
Использование select
#include        "unp.h"
void str_cli(FILE *fp, int sockfd) {
        int             maxfdp1, stdineof;
        fd_set          rset;
        char            sendline[MAXLINE], recvline[MAXLINE];

        stdineof = 0;
        FD_ZERO(&rset);
        for ( ; ; ) {
                if (stdineof == 0) FD_SET(fileno(fp), &rset);
                FD_SET(sockfd, &rset);
                maxfdp1 = max(fileno(fp), sockfd) + 1;
                Select(maxfdp1, &rset, NULL, NULL, NULL);
                if (FD_ISSET(sockfd, &rset)) {   /* socket is readable */
                        if (Readline(sockfd, recvline, MAXLINE) == 0) {
                                if (stdineof == 1) return; /* normal termination */
                                else    err_quit("str_cli: server terminated prematurely");
                        }

                        Fputs(recvline, stdout);
                }
                if (FD_ISSET(fileno(fp), &rset)) {  /* input is readable */
                        if (Fgets(sendline, MAXLINE, fp) == NULL) {
                                stdineof = 1;
                                Shutdown(sockfd, SHUT_WR); /* send FIN */
                                FD_CLR(fileno(fp), &rset);
                                continue;
                        }
                        Writen(sockfd, sendline, strlen(sendline));
                }
        }
}
```

Мультиплексирование ввода при помощи poll(2)

Системный вызов poll(2) выполняет приблизительно те же задачи, что и select(3C), но использует несколько более удобный способ передачи информации о том, какие дескрипторы его интересуют. poll(2) имеет три параметра:

struct pollfd fds[] – массив описателей дескрипторов. Структура pollfd обсуждается далее в этом разделе

nfds_t nfds – количество описателей в массиве fds

int timeout – тайм-аут в миллисекундах. Если параметр timeout равен 0, poll работает в режиме опроса (возвращает управление немедленно). Если он равен -1, poll ждет готовности дескрипторов неограниченное время.

poll(2) возвращает количество дескрипторов, с которыми произошли какие-то события, запрошенные программой либо представляющие интерес для нее. Если poll(2) возвращает управление по тайм-ауту, код возврата будет равен 0. При ошибке poll(2) возвращает -1 и устанавливает errno.

Структура pollfd имеет следующие поля:

int fd – дескриптор файла. Если это поле имеет отрицательное значение, запись игнорируется.

short events – события, связанные с fd, которые нас интересуют.

short revents – return events, события, связанные с fd, которые реально произошли.

При вызове poll пользователь должен заполнить поля fd и events; поле revents заполняется системным вызовом.

Поля events и revents представляют собой битовые маски, биты которых соответствуют типам событий. Вместо битов рекомендуется использовать символьные константы, определенные в <poll.h>

Основные используемые типы событий – POLLIN (проверять готовность к чтению), и POLLOUT (проверять готовность к записи). В действительности, эти типы композитные и представляют собой сочетания разных типов событий. Так, для сокетов TCP можно указывать проверку поступления внеполосных данных, для устройств STREAMS – проверку поступления приоритетных данных и т.д. В revents устанавливаются биты, соответствующие реально происшедшему событию, т.е. если вы заказывали ожидание POLLIN, не обязательно в revents будут установлены все биты, входящие в маску POLLIN. Это необходимо иметь в виду при проверке revents (см. пример 2).

Кроме POLLIN и POLLOUT, в revents также могут появляться биты POLLERR, POLLHUP и POLLNVAL. В events эти биты игнорируются, а в revents могут быть установлены при следующих условиях:

POLLERR – на устройстве возникла ошибка

POLLHUP – сокет, труба или терминальное устройство закрыты на другом конце

POLLNVAL – значение fd не соответствует валидному файловому дескриптору (скорее всего, дескриптор был закрыт на нашем конце).

Использование poll(2) (фрагмент программы)

```
#include <poll.h>
struct pollfd fds[3];
int ifd1, ifd2, ofd, count;

fds[0].fd = ifd1;
fds[0].events = POLLNORM;
fds[1].fd = ifd2;
fds[1].events = POLLNORM;
fds[2].fd = ofd;
fds[2].events = POLLOUT;
count = poll(fds, 3, 10000);
if (count == -1) {
        perror("poll failed");
        exit(1);
}
if (count==0)
        printf("No data for reading or writing\n");
if (fds[0].revents & POLLNORM)
        printf("There is data for reading fd %d\n", fds[0].fd);
if (fds[1].revents & POLLNORM)
        printf("There is data for reading fd %d\n", fds[1].fd);
if (fds[2].revents & POLLOUT)
        printf("There is room to write on fd %d\n", fds[2].fd);
```

Сравнение poll(2) и select(3C)

Преимущества poll(2) перед select(3C) достаточно очевидны:

1. интерфейс poll не накладывает ограничений на пространство номеров дескрипторов, во всяком случае пока эти номера входят в диапазон представления int.
2. при большом пространстве номеров дескрипторов (65536 в данном контексте следует считать большим пространством), poll часто требует передачи между пользовательским процессом и ядром меньшего объема данных, чем select.
3. poll сообщает больше информации о происшедших с дескриптором событиях, чем может сообщить select
4. У poll входные и выходные значения разнесены по разным полям структуры, так что не требуется полностью пересоздавать массив fds после каждого вызова.

Использование /dev/poll

Использование poll(2) с большим количеством файловых дескрипторов приводит к передаче больших объемов данных между пользовательским процессом и ядром. При этом, скорее всего, большая часть этих данных передается впустую – ведь если процесс действительно может работать с таким большим числом дескрипторов, это, скорее всего, означает, что большинство из них не готовы к работе.

В Solaris предоставляется нестандартный API, который может использоваться для решения этой проблемы. Этот API описывается на странице системного руководства poll(7D) и состоит в использовании специального псевдоустройства /dev/poll.

Это устройство открывается как обычный файл системным вызовом open(2). Затем в него следует записать одну или несколько структур pollfd (т.е. тех же самых структур, которые использует poll(2)). Запись осуществляется системным вызовом write(2) и может осуществляться в несколько приемов. При этом, если вы несколько раз записываете структуры, соответствующие одному и тому же дескриптору, с разными значениями поля events, это будет означать расширение списка опрашиваемых событий для вашего дескриптора. Т.е. если вы сначала запишете pollfd с events==POLLIN, а затем с events==POLLOUT, дескриптор будет опрашиваться в режиме POLLIN | POLLOUT.

Если вы хотите исключить дескриптор из множества опрашиваемых, вам следует записать структуру pollfd, в которой поле events содержит бит POLLREMOVE.

Многократное открытие /dev/poll одним процессом приводит к созданию нескольких независимых наборов дескрипторов.

Сам опрос осуществляется вызовом ioctl(2) с командой DP_POLL. Этот ioctl использует в качестве параметра значение struct dvpoll *. Тип struct dvpoll описан в <sys/devpoll.h> и содержит следующие поля:

struct pollfd* dp_fds – указатель на массив, в который следует положить описатели дескрипторов, с которым связаны события

int dp_nfds – размер массива dp_fds. Также, максимальное количество описателей дескрипторов, которые следует получить

int dp_timeout – тайм-аут в миллисекундах. Этот параметр соответствует параметру timeout poll(2).

Ioctl DP_POLL возвращает количество описателей файловых дескрипторов, записанных в dp_fds, 0 если ioctl был разблокирован по тайм-ауту и -1 в случае ошибки.

С практической точки зрения, важное отличие этого API от poll(2) состоит в том, что при использовании poll(2) описатели дескрипторов после опроса расположены на тех же местах в массиве, на которых вы сами их разместили. Напротив, ioctl DP_POLL возвращает вам массив, который включает только те дескрипторы, с которыми связаны события, причем эти дескрипторы лежат в массиве в том порядке, в котором их счел удобным разместить драйвер /dev/poll. Т.е. вы должны просматривать dp_fds в порядке увеличения индекса, используя затем значение поля fd как ключ поиска. Скорее всего, вам придется завести массив (возможно, ассоциативный), связывающий значение файлового дескриптора с метаинформацией о том, что это за дескриптор и что вы с ним хотели делать. При работе с этим массивом вам следует иметь в виду, что Unix при нормальной работе переиспользует номера файловых дескрипторов, т.е. вам надо обновлять данные в вашем массиве каждом закрытии файла.

Корректная реализация всей требуемой функциональности (особенно в многопоточной программе) требует большого объема кода, причем кода, довольно сложного при отладке.

Скорее всего, именно поэтому поддержка poll(7D) приложениями ограниченна и этот API не стал ни юридическим стандартом, ни даже стандартом де-факто. Большинство современных Unix-систем, за исключением Solaris, не поддерживают /dev/poll и не имеют планов реализации такой поддержки в обозримом будущем.

Однако с точки зрения производительности poll(7D) дает ощутимые преимущества даже при вполне реалистичных количествах файловых дескрипторов на процесс. По данным измере-

ний, опубликованных на сайте http://developers.sun.com/solaris/articles/polling_efficient.html, при 4000 тысячах файловых дескрипторов, poll(7D) требует в 16 раз меньше процессорного времени на исполнение заданного количества циклов опроса-чтения-записи, чем poll(2)!

Асинхронный ввод/вывод

Еще одна альтернатива многопоточности для приложений, ориентированных на ввод/вывод – это асинхронный ввод/вывод.

Традиционно, Unix использует синхронную модель ввода/вывода. Системные вызовы read(2), write(2) и их аналоги возвращают управление только того, как данные уже прочитаны или записаны. Часто это приводит к блокировке нити.

> **Примечание**
>
> В действительности, не все так просто. read(2) действительно должен ожидать физического прочтения данных с устройства, но write(2) по умолчанию работает в режиме отложенной записи: он возвращает управление после того, как данные переданы в системный буфер, но, вообще говоря, до того, как данные будут физически переданы устройству. Это, как правило, значительно повышает наблюдаемую производительность программы и позволяет использовать память из-под данных для других целей сразу после возврата write(2). Но отложенная запись имеет и существенные недостатки. Главный из них – вы узнаете о результате физической операции не сразу по коду возврата write(2), а лишь через некоторое время после возврата, обычно по коду возврата следующего вызова write(2). Для некоторых приложений – для мониторов транзакций, для многих программ реального времени и др. – это неприемлемо и они вынуждены выключать отложенную запись. Это делается флагом O_SYNC, который может устанавливаться при открытии файла и изменяться у открытого файла вызовом fcntl(2). Синхронизация отдельных операций записи может быть обеспечена вызовом fsync(2).

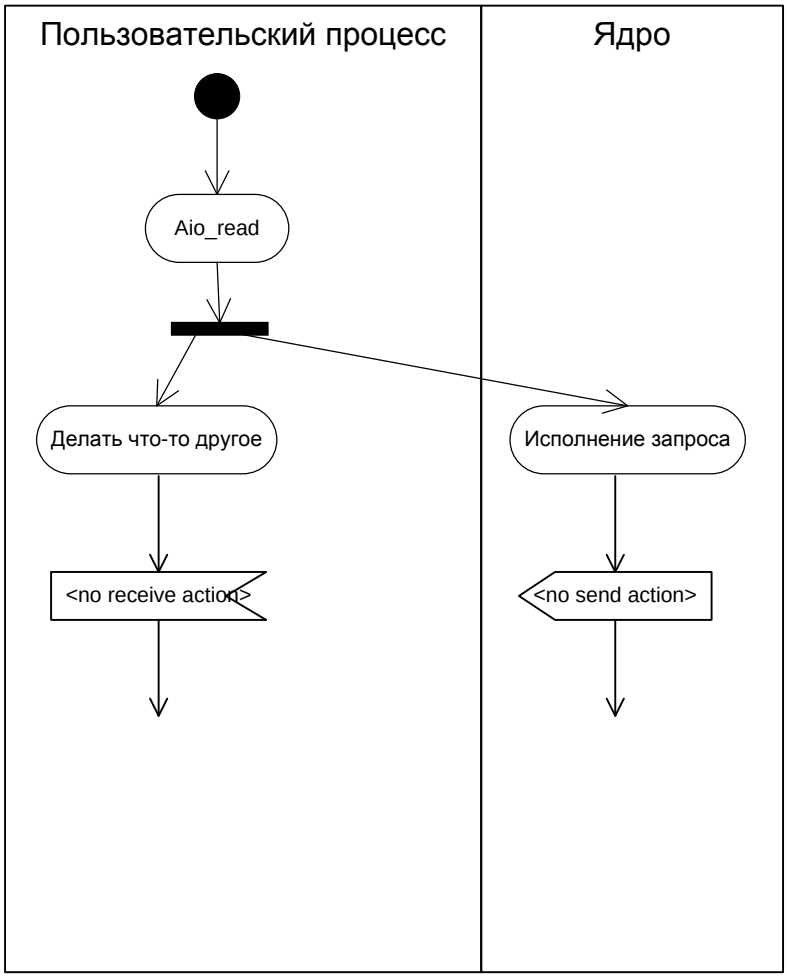
Для многих приложений, работающих с несколькими устройствами и/или сетевыми соединениями синхронная модель неудобна. Работа в режиме опроса тоже не всегда приемлема. Дело в том, что select(3C) и poll(2) считают дескриптор файла готовым для чтения только после того, как в его буфере физически появятся данные. Но некоторые устройства начинают отдавать данные только после того, как их об этом явно попросят.

Также, для некоторых приложений, особенно для приложений реального времени важно знать точный момент начала поступления данных. Для таких приложений также может быть неприемлемо то, что select(3C) и poll(2) считают регулярные файлы всегда готовыми для чтения и записи. Действительно, файловая система читается с диска и хотя она работает гораздо быстрее, чем большинство сетевых соединений, но все-таки обращения к ней сопряжены с некоторыми задержками. Для приложений жесткого реального времени эти задержки могут быть неприемлемы – но без явного запроса на чтение файловая система данные не отдает!

С точки зрения приложений жесткого реального времени может оказаться существенным еще один аспект проблемы ввода/вывода. Дело в том, что приложения жесткого РВ имеют более высокий приоритет, чем ядро, поэтому исполнение ими системных вызовов – даже неблокирующихся! – может привести к инверсии приоритета.

Решение этих проблем известно давно и называется асинхронный ввод/вывод. В этом режиме системные вызовы ввода/вывода возвращают управление сразу после формирования запроса к драйверу устройства, как правило, даже до того, как данные будут скопированы в системный буфер. Формирование запроса состоит в установке записи (IRP, Input/Output Request Packet, пакет запроса ввода вывода) в очередь. Для этого надо лишь ненадолго захватить мутекс, защищающий «хвост» очереди, поэтому проблема инверсии приоритета легко преодолима. Для того, чтобы выяснить, закончился ли вызов, и если закончился, то чем именно, и можно ли использовать память, в которой хранились данные, предоставляется специальный API (см. рис. 1)

Рис. 1. Асинхронная модель ввода/вывода



Асинхронная модель была основной моделью ввода/вывода в таких ОС, как DEC RT-11, DEC RSX-11, VAX/VMS, OpenVMS. Эту модель в той или иной форме поддерживают практически все ОС реального времени. В системах семейства Unix с конца 1980х использовалось несколько несовместимых API для асинхронного ввода/вывода. В 1993 году ANSI/IEEE был принят документ POSIX 1003.1b, описывающий стандартизованный API, который мы и изучим далее в этом разделе.

Функции aio_read(3AIO), aio_write(3AIO) и lio_listio(3AIO)

В Solaris 10 функции асинхронного ввода/вывода включены в библиотеку libaio.so. Для сборки программ, использующих эти функции, необходимо использовать ключ –laio.
Для формирования запросов на асинхронный ввод/вывод используются функции aio_read(3AIO), aio_write(3AIO) и lio_listio(3AIO).
Функции aio_read(3AIO) и aio_write(3AIO) имеют единственный параметр, struct aiocb *aiocbp. Структура aiocb определена в файле <aio.h> и содержит следующие поля:

int aio_fildes – дескриптор файла

off_t aio_offset – смещение в файле, начиная с которого будет идти запись или чтение

volatile void* aio_buf – буфер, в который следует прочитать данные или в котором лежат данные для записи.

size_t aio_nbytes – размер буфера. Как и традиционный read(2), aio_read(3AIO) может прочитать меньше данных, чем было запрошено, но никогда не прочитает больше.

int aio_reqprio – приоритет запроса

struct sigevent aio_sigevent – способ оповещения о завершении запроса (рассматривается далее в этом разделе)

int aio_lio_opcode – при aio_read(3AIO) и aio_write(3AIO) не используется, используется только функцией lio_listio.

Функция lio_listio(3AIO) позволяет одним системным вызовом сформировать несколько запросов на ввод/вывод. Эта функция имеет четыре параметра:

int mode – может принимать значения LIO_WAIT (функция ждет завершения всех запросов) и LIO_NOWAIT (функция возвращает управление сразу после формирования всех запросов).

struct aiocb *list[] – список указателей на структуры aiocb с описаниями запросов. Запросы могут быть как на чтение, так и на запись, это определяется полем aio_lio_opcode. Запросы к одному дескриптору исполняются в том порядке, в каком они указаны в массиве list.

int nent – количество записей в массиве list.

struct sigevent *sig – способ оповещения о завершении всех запросов. Если mode==LIO_WAIT, этот параметр игнорируется.

Библиотека POSIX AIO предусматривает два способа оповещения программы о завершении запроса, синхронный и асинхронный. Сначала рассмотрим синхронный способ.

Проверка статуса асинхронного запроса

Функция aio_return(3AIO) возвращает статус запроса. Если запрос уже завершился и завершился успешно, она возвращает размер прочитанных или записанных данных в байтах. Как и у традиционного read(2), в случае конца файла aio_return(3AIO) возвращает 0 байт. Если запрос завершился ошибкой или еще не завершился, возвращается -1 и устанавливается errno. Если запрос еще не завершился, код ошибки равен EINPROGRESS. Функция aio_return(3AIO) разрушающая; если ее вызвать для завершенного запроса, то она уничтожит системный объект, хранящий информацию о статусе запроса. Многократный вызов aio_return(3AIO) по поводу одного и того же запроса, таким образом, невозможен.
Функция aio_error(3AIO) возвращает код ошибки, связанной с запросом. При успешном завершении запроса возвращается 0, при ошибке – код ошибки, для незавершенных запросов – EINPROGRESS.
Функция aio_suspend(3AIO) блокирует нить до завершения одного из указанных ей запросов асинхронного ввода/вывода либо на указанный интервал времени. Эта функция имеет три параметра:

const struct aiocb *const list[] – массив указателей на описатели запросов.
int nent – количество элементов в массиве list.
const struct timespec *timeout – тайм-аут с точностью до наносекунд (в действительности, с точностью до разрешения системного таймера).

Функция возвращает 0, если хотя бы одна из операций, перечисленных в списке, завершилась. Если функция завершилась с ошибкой, она возвращает -1 и устанавливает errno. Если функция завершилась по тайм-ауту, она также возвращает -1 и errno==EINPROGRESS.
Пример использования асинхронного ввода/вывода с синхронной проверкой статуса запроса приводится в примере 3.

Асинхронный ввод/вывод с синхронной проверкой статуса запроса.

Код сокращен, из него исключены открытие сокета и обработка ошибок.

```
const char req[]="GET / HTTP/1.0\r\n\r\n";
int main() {
	int s;
	static struct aiocb readrq;
	static const struct aiocb *readrqv[2]={&readrq, NULL};
/* Открыть сокет […] */
	memset(&readrq, 0, sizeof readrq);
	readrq.aio_fildes=s;
	readrq.aio_buf=buf;
	readrq.aio_nbytes=sizeof buf;
	if (aio_read(&readrq)) {
		/* … */
	}
	write(s, req, (sizeof req)-1);

	while(1) {
		aio_suspend(readrqv, 1, NULL);
		size=aio_return(&readrq);
		if (size>0) {
			write(1, buf, size);
			aio_read(&readrq);
		} else if (size==0) {
			break;
		} else if (errno!=EINPROGRESS) {
			perror("reading from socket");
		}
	}
}
```

Асинхронное оповещение о завершении операции

Асинхронное оповещение приложения о завершении операций состоит в генерации сигнала при завершении операции. Чтобы это сделать, необходимо внести соответствующие настройки в поле aio_sigevent описателя запроса.

Поле aio_sigevent имеет тип struct sigevent. Эта структура определена в <signal.h> и содержит следующие поля:

int sigev_notify – режим нотификации. Допустимые значения – SIGEV_NONE (не посылать подтверждения), SIGEV_SIGNAL (генерировать сигнал при завершении запроса) и SIGEV_THREAD (при завершении запроса запускать указанную функцию в отдельной нити). Solaris 10 также поддерживает тип оповещения SIGEV_PORT, который рассматривается в приложении к этой лекции.

int sigev_signo – номер сигнала, который будет сгенерирован при использовании SIGEV_SIGNAL.

union sigval sigev_value – параметр, который будет передан обработчику сигнала или функции обработки. При использовании для асинхронного ввода/вывода это обычно указатель на запрос. При использовании SIGEV_PORT это должен быть указатель на структуру port_event_t, содержащую номер порта и, возможно, дополнительные данные.

void (*sigev_notify_function)(union sigval) – функция, которая будет вызвана при использовании SIGEV_THREAD.

pthread_attr_t *sigev_notify_attributes – атрибуты нити, в которой будет запущена sigev_notify_function при использовании SIGEV_THREAD.

Далеко не все реализации libaio поддерживают оповещение SIGEV_THREAD. Некоторые Unix-системы используют вместо него нестандартное оповещение SIGEV_CALLBACK. Далее в этой лекции мы будем обсуждать только оповещение сигналом.

В качестве номера сигнала некоторые приложения используют SIGIO или SIGPOLL (в Unix SVR4 это один и тот же сигнал). Часто используют также SIGUSR1 или SIGUSR2; это удобно потому, что гарантирует, что аналогичный сигнал не возникнет по другой причине. В приложениях реального времени используются также номера сигналов в диапазоне от SIGRTMIN до SIGRTMAX. Некоторые реализации выделяют для этой цели специальный номер сигнала SIGAIO или SIGASYNCIO, но в Solaris 10 такого сигнала нет.

Разумеется, перед тем, как исполнять асинхронные запросы с оповещением сигналом, следует установить обработчик этого сигнала. Для оповещения необходимо использовать сигналы, обрабатываемые в режиме SA_SIGINFO. Установить такой обработчик при помощи системных вызовов signal(2) и sigset(2) невозможно, необходимо использовать sigaction(2). Установка обработчиков при помощи sigaction рассматривается в приложении 2 к этой лекции.

Обработка сигнала в режиме SA_SIGINFO имеет два свойства, каждое из которых полезно для наших целей. Во первых, обработчики таких сигналов имеют три параметра, в отличие от единственного параметра у традиционных обработчиков. Первый параметр имеет тип int и соответствует номеру сигнала, второй параметр имеет тип siginfo_t *, генерируется системой и содержит ряд интересных полей. Нас в данном случае больше всего интересует поле этой структуры si_value. Как раз в этом поле нам передается значение aio_sigevent.sigev_value, которое мы создали при настройке структуры aiocb.

Приложение 1. Порты Solaris

В Solaris 10 введен новый API, который может быть полезен и в качестве замены select/poll, и для управления запросами асинхронного ввода/вывода. В действительности, select и poll в Solaris 10 реализуются с использованием портов.

Порт представляет собой объект, идентифицируемый файловым дескриптором. Порт создается функцией port_create(3C) и уничтожается системным вызовом close(2). С уже созданным портом можно связывать объекты, способные генерировать события. В настоящее время поддерживаются следующие типы источников событий:

PORT_SOURCE_AIO – запросы асинхронного ввода-вывода. Событием считается завершение запроса.

PORT_SOURCE_FD – файловые дескрипторы. При ассоциации файлового дескриптора с портом необходимо указать флаги в формате поля events структуры pollfd; событиеэти флаги и будут задавать события, которые нас интересуют.

PORT_SOURCE_TIMER – таймеры.

PORT_SOURCE_USER – события, генерируемые пользователем при помощи функции port_send(3C).

PORT_SOURCE_ALERT – события, генерируемые пользователем при помощи функции port_send(3C).

Связывание асинхронного запроса с портом происходит в момент создания запроса, установкой поля aio_sigevent.sigev_notify=SIGEV_PORT.

Связывание файлового дескриптора с портом производится функцией port_associate(3C). Судя по количеству и типам параметров, эта функция должна допускать связывание с портом различных объектов, но в Solaris 10 поддерживается единственный тип источника - PORT_SOURCE_FD.

С одним портом можно связывать разнородные источники событий.

После того, как источники связаны с портом, можно получать информацию о возникших событиях функциями port_get(3C) и port_getn(3C). После того, как порт вернул информацию о событии, источник события отсоединяется от порта. Если от этого источника ожидаются какие-то еще события, источник следует связать с портом заново. В случае запросов асинхронного ввода/вывода и таймера смысл отсоединения очевиден. В случае с файлами смысл тоже достаточно легко понять – это защищает нас от неприятных сценариев использования select/poll в многопоточной программе, которые мы рассматривали в ходе этой лекции. При использовании традиционного select/poll существует возможность, что несколько нитей получат один и тот же файловый дескриптор и попытаются что-то с ним сделать. В лучшем случае это приведет к блокировке некоторых нитей, в худшем – к потере данных. Использование портов защищает нас от такого развития событий - после того, как одна из нитей получит файловый дескриптор у порта, он будет отсоединен и остальные нити не смогут получить его, пока он не будет присоединен снова явным образом.

Порты представляют собой новый API. В страницах системного руководства Solaris 10 функции этого API отмечены как evolving (развивающиеся).

В настоящее время важным недостатком портов является то, что примитивы синхронизации POSIX Thread API нельзя использовать в качестве источников событий наравне с файловыми дескрипторами и таймерами.

Приложение 2. Установка обработчиков сигналов при помощи sigaction(2)

Системный вызов sigaction(2) рассматривается в разделе «Сигналы. Он предоставляет возможность регистрации обработчиков сигналов с дополнительными параметрами. Для этого необходимо использовать поле sa_sigaction. В качестве параметра, sigaction(2) получает struct sigaction со следующими полями:

void (*sa_handler)(int); - Адрес традиционного обработчика сигнала, SIG_IGN или SIG_DFL

void (*sa_sigaction)(int, siginfo_t *, void *) – Адрес обработчика сигнала в режиме SA_SIGINFO. Реализации могут совмещать это поле с полем sa_handler, поэтому не следует пытаться установить sa_handler и sa_sigaction в одной структуре.

sigset_t sa_mask - Маска сигналов, которые должны быть заблокированы, когда вызывается функция обработки сигнала.

int sa_flags - Флаги, управляющие доставкой сигнала.

Если аргумент act ненулевой, он указывает на структуру, определяющую новые действия, которые должны быть предприняты при получении сигнала sig. Если аргумент oact ненулевой, он указывает на структуру, где сохраняются ранее установленные действия для этого сигнала.

Поле sa_flags в struct sigaction формируется побитовым ИЛИ следующих значений:

SA_ONSTACK - Используется для бработки сигналов на альтернативном сигнальном стеке.

SA_RESETHAND - Во время исполнения функции обработки сбрасывает реакцию на сигнал к SIG_DFL; обрабатываемый сигнал при этом не блокируется.

SA_NODEFER - Во время обработки сигнала сигнал не блокируется.

SA_RESTART - Системные вызовы, которые будут прерваны исполнением функции обработки, автоматически перезапускаются.

SA_SIGINFO - Указывает на необходимость использовать значение sa_sigaction в качестве функции-обработчика сигнала. Также используется для доступа к подробной информации о процессе, исполняющем сигнальный обработчик, такой как причина возникновения сигнала и контекст процесса в момент доставки сигнала.

5. СОЗДАНИЕ ПРОЦЕССОВ И ИСПОЛНЕНИЕ ПРОГРАММ

Обзор

Этот раздел объясняет, что такое процесс и чем он отличается от программы. Вы изучите взаимодействие между родительским процессом и одним или более порожденными процессами (подпроцессами). Также обсуждается создание подпроцесса, исполнение другой программы в том же процессе, заверение процесса и ожидание завершения порожденного процесса.

Что такое процесс? - Обзор

Процесс представляет собой исполняющуюся программу вместе с необходимым ей окружением. Окружение процесса состоит из:

. информации о процессе, содержащейся в различных системных структурах данных

. содержимого регистров процессора (контекста процесса)

. пользовательского стека процесса и системного стека, используемого при обработке системных вызовов

. пользовательской области, которая содержит информацию об открытых файлах, текущей директории, обработке сигналов и т.д.

Образ процесса есть размещение процесса в памяти в заданный момент времени. Образ процесса представляет собой набор отображённых в виртуальную память сегментов, представляющих собой код основной программы, сегмент данных, сегменты кода и данных разделяемых библиотек, отображённые на память файлы, сегменты разделяемой памяти System V IPC и др. Обратите внимание, что программа является частью образа процесса.

Благодаря тому, что отображённые в память сегменты не обязаны быть загружены полностью и подкачиваются с диска по мере необходимости, а также благодаря тому, что сегменты кода и неизмененные части сегментов данных у различных процессов могут размещаться в разделяемой памяти, общий объем образов всех процессов в системе может превосходить объем физического ОЗУ.

Каждый процесс имеет собственное виртуальное адресное пространство и, таким образом, защищён от ошибок и злонамеренных действий кода, запущенного другими пользователями.

В современных системах семейства Unix, в рамках процесса может быть создано несколько нитей или потоков исполнения. В рамках данного раздела мы не изучаем многопоточное исполнение и рассматриваем только традиционные процессы с единственной нитью.

Процесс представляет собой изменяющийся со временем динамический объект. Процесс может создать один или более порождённых процессов, используя системный вызов fork(2). Кроме того, он может изменить свою программу, используя системный вызов exec(2). Процесс может приостановить исполнение, используя системные вызовы wait(2) или waitid(2). Он может также завершить своё исполнение системным вызовом exit(2).

Многие прикладные программы реализованы не в виде единой монолитной программы, а в виде нескольких или даже множества взаимодействующих процессов. Это позволяет

повысить отказоустойчивость: аварийное завершение одного процесса не обязательно приводит к нарушению работы всей программы,

реализовать принцип минимума привилегий: каждый процесс исполняется с теми правами, которые необходимы ему для выполнения его функций, но не более,

обойти ограничение на размер образа процесса, что было очень актуально на 16-разрядных компьютерах, а в последние годы становится актуально на 32-разрядных машинах, а также некоторые другие ограничения, например, ограничение на количество одновременно открытых файлов,

распределить вычислительную нагрузку по нескольким процессорным ядрам или виртуальным процессорам. Эту задачу можно также решать с использованием многопоточности, но многие современные прикладные программы были разработаны до того, как были стандартизованы API и средства для многопоточного программирования.

Создание процесса

Системный вызов fork(2) создаёт новый процесс, исполняющий копию исходного процесса. В основном, новый процесс (порождённый или дочерний) идентичен исходному (родителю). В описании fork(2) перечислены атрибуты, которые порождённый процесс наследует от родителя, и различия между ними.

Дочерний процесс наследует все отображённые на память файлы и вообще все сегменты адресного пространства, все открытые файлы, идентификаторы группы процессов и сессии, реальный и эффективный идентификаторы пользователя и группы, ограничения rlimit, текущий каталог, а также ряд других параметров, которые будут обсуждаться в следующих разделах.

Дочерний процесс НЕ наследует: идентификатор процесса, идентификатор родительского процесса, а также захваченные участки файлов. В большинстве Unix-систем, дочерний процесс не наследует нити исполнения, кроме той, из которой был вызван fork(2). Однако в Solaris предоставляется системный вызов forkall(2), который воспроизводит в дочернем процессе все нити родительского. Этот системный вызов не имеет аналогов в стандарте POSIX и его использование приведёт к проблемам при переносе вашей программы на другие платформы.

После возврата из fork(2), оба процесса продолжают исполнение с той точки, где fork(2) был вызван. Процессы могут узнать, кто из них является родителем, а кто порождённым, на основании значения, возвращённого fork(2).

Родительский процесс получает идентификатор порождённого процесса, положительное число. Порождённый процесс получает нулевое значение. Как правило, за fork(2) следует оператор if или switch, который определяет, какой код исполнять для родительского и какой для порождённого процесса.

Системный вызов fork(2) может завершиться неудачей, если вы пытаетесь превысить разрешённое количество процессов для каждого пользователя или общее количество процессов в системе. Эти два ограничения устанавливаются при конфигурации операционной системы. Если fork(2) завершается неуспехом, он возвращает значение -1. Рекомендуется проверять код возврата этого и остальных системные вызовы на предмет неудачного завершения.

Системный вызов fork(2)

Эта иллюстрация показывает родительский процесс до вызова fork(2) и после того, как этот вызов возвратил управление. После fork(2) исполняются два процесса с различными идентификаторами. Сегменты текста, данных и стека у родительского и порождённого процессов идентичны. Для программы с разделяемым сегментом TEXT (компилятор по умолчанию создаёт именно такие программы), сегмент кода, в действительности, будет одним и тем же физическим сегментом. После fork(2) только увеличится счётчик ссылок на него.

Оба процесса имеют почти одинаковые пользовательские области. Так как пользовательская область содержит таблицу дескрипторов файлов, все перенаправления ввода/вывода, сделанные в родительском процессе, наследуются потомком. Захваты файлов являются собственностью процесса, поэтому подпроцесс не унаследует захватов файлов, сделанных родителем. Отображение файлов на память при fork(2) сохраняется.

При этом, если сегмент отображался в режиме MAP_SHARED, родительский и порождённый процессы используют одну и ту же физическую память; если такой сегмент доступен для модификации, то родитель и потомок будут видеть вносимые в эту память изменения. Большинство разделяемых сегментов в Unix-системах — это сегменты кода, недоступные для модификации.

Если же сегмент отображался как MAP_PRIVATE или MAP_ANON (по умолчанию, сегменты данных и стека отображаются именно так), процессы получают собственные копии соответствующих страниц. В действительности, копирование происходит при первой записи в страницу. Непосредственно после fork(2), приватные сегменты остаются разделяемыми, но система устанавливает на страницы этих сегментов защиту от записи. Чтение таких страниц происходит без изменений, но при попытке модификации такой страницы, диспетчер памяти генерирует исключение защиты памяти. Ядро перехватывает это исключение, но вместо завершения процесса по SIGSEGV (как это происходит при обычных ошибках защиты памяти), создаёт копию приватной страницы и отображает эту копию в адресное пространство того процесса, который пытался произвести запись. Такое поведение называется копированием при записи (copy-on-write).

Поскольку большая часть памяти родительского и порождённого процессов является разделяемой, fork(2) представляет собой относительно дешёвую операцию, во всяком случае, существенно более дешёвую, чем создание процесса в Win32, но существенно более дорогую, чем создание нити в рамках процесса.

Системный вызов fork(2) - Пример

Этот пример иллюстрирует создание подпроцесса. После fork(2) два процесса будут исполнять одну и ту же программу. Они оба распечатают свой идентификатор процесса и идентификатор родителя. Эта программа работает следующим образом:

8-9 Процесс распечатывает идентификатор своего родительского процесса; при работе в терминальной сессии это обычно идентификатор процесса командного процессора.

11 Создается новый процесс. Новый (порожденный) процесс является [почти] точной копией вызывающего процесса (родителя).

13-14 Оба процесса исполняют этот оператор.

Файл: fork0.c

Замечание: Вызов getppid(2), исполняемый вновь порожденным процессом, может возвратить идентификатор процесса 1, то есть процесса init. После fork(2) родительский и порождённый процесс исполняются независимо. Родитель может завершиться раньше, чем порождённый, особенно если он выполняет меньше действий. Если родительский процесс завершается раньше порождённого, то последний "усыновляется" процессом init, то есть система устанавливает процесс с идентификатором 1 в качестве родителя подпроцесса.

СИСТЕМНЫЙ ВЫЗОВ fork(2) - ПРИМЕР

```
 1 #include <sys/types.h>
 2 #include <unistd.h>
 3 #include <stdio.h>
 4
 5 main()
 6 {
 7
 8         printf("[%ld] parent process id: %ld\n",
 9         getpid(), getppid());
10
11         fork();
12
13         printf("\n\t[%ld] parent process id: %ld\n",
14         getpid(), getppid());
15 }
```

```
$ fork0
[18607] parent process id: 18606

        [18608] parent process id: 18607

        [18607] parent process id: 18606
```

Системный вызов fork(2) - Пример

Эта программа создает два процесса: родитель распечатывает заглавные буквы, а порожденный - строчные.

13-16 Значение, возвращенное fork(2), присваивается переменной pid. Положительное значение pid означает родительский процесс.

17-20 Нулевое значение pid означает порожденный процесс.

21-24 Если значение, возвращаемое fork(2), равно -1, то произошла ошибка. В этом случае вызывается функция perror(3), которая распечатывает сообщение о причине неуспеха. Затем программа завершается.

21-24 Как родительский, так и порожденный процессы распечатывают буквы этим оператором for. Внутренний цикл здесь просто потребляет время процессора, чтобы происходило переключение процессов. Это приводит к тому, что большие и маленькие буквы на выводе перемешиваются. Иначе оба процесса быстро бы завершались в течении одного кванта времени.

Оба процесса имеют одинаковую возможность получить управление. Поэтому любой из них может начать исполнение первым.

Файл: fork1.c

СИСТЕМНЫЙ ВЫЗОВ fork(2) - ПРИМЕР

```
 1 #include <sys/types.h>
 2 #include <unistd.h>
 3 #include <stdlib.h>
 4 #include <stdio.h>
 5 static const int Bignumber = 10000;
 6
 7 main(int argc, char *argv[ ])   /* demonstrate fork(2) */
 8 {
 9     char ch, first, last;
10         pid_t pid;
11     int i;
12
13     if ((pid = fork()) > 0) {   /* parent */
14                 first = 'A';
15         last = 'Z';
16     }
17         else if (pid == 0) {            /* child */
18         first = 'a';
19         last = 'z';
20     }
21         else {                  /* cannot fork(2) */
22         perror(argv[0]);
23         exit(1);
24     }
25         for (ch = first; ch <= last; ch++) {
26         /* delay loop */
27         for (i = 0; i < Bignumber; i++)
28             ;   /* null */
29         write(1, &ch, 1);
30     }
31
32     exit(0);
33 }
```

Системный вызов fork(2) - Пример

Наблюдая за выводом программы-примера, можно заметить следующие факты:

. каждый из процессов выводит свой текст в правильном порядке, то есть, как заглавные, так и строчные буквы идут в алфавитном порядке.

. время исполнения каждого процесса непредсказуемо.

. невозможно предсказать, какой из процессов закончится первым.

Как правило, существует несколько процессов, поочерёдно использующих центральный процессор. Каждому процессу выделяется определённое количество времени процессора (квант). Когда процесс израсходовал свой квант, процессор может быть передан другому процессу. Этот механизм предотвращает захваты процессора одним процессом.

Обратите внимание, что при первых двух вызовах приглашение shell появилось в середине строки вывода. Это случилось, потому что родительский процесс завершился раньше порождённого. shell выдает приглашение при завершении родительского процесса, не ожидая завершения его подпроцессов.

СИСТЕМНЫЙ ВЫЗОВ fork(2) - ПРИМЕР (ВЫВОД)

```
$ fork1
abcABdeCDfEFgGhijklHIJKmnopLMNOPQRqrstSTUuvwxyVWXYZ$ z

$ fork1
aAbBCDEFGHIJcdefghijkKLMNOPQRSlmnTUVopqrstWXYZ$ uvwxyz

$ fork1
abABCcdefgDEFGhijklmnoHIJKLMpNqOPrQsRtuvSTUwxVWyXzYZ$

$ fork1
abcAdeBCfDghEFGHIijkJKlLMNOmnopqPQRrsSTtuUVvWwxXyzYZ$
```

Исполнение программы

Процесс может заменить текущую программу на новую, исполнив системный вызов exec(2). Этот вызов заменяет текст, данные, стек и остальные сегменты виртуального адресного пространства текущей программы на соответствующие сегменты новой программы. Однако пользовательская область при этом вызове сохраняется.

Существует шесть вариантов системного вызова exec(2). Обратите внимание, что за exec идет одна или несколько букв:

l (список аргументов),

v (вектор аргументов),

e (изменение среды) или

p (использование переменной PATH).

Формат вызова exec(2) определяет, какие данные передаются новой программе. Ниже приведены параметры различных версий exec(2):

path указывает на строку, которая содержит абсолютное или относительное имя загрузочного модуля.

file указывает на строку, содержащую имя загружаемого файла, который находится в одной из директорий, перечисленных в переменной PATH.

arg0,...,argn указывают на строки - значения параметров, которые надо передать новой программе. Эти значения помещаются в вектор argv[] - параметр функции main() новой программы. Количество параметров помещается в параметр argc функции main(). Список параметров должен завершаться нулевым указателем.

argv[] вектор указателей на строки, содержащие параметры, которые нужно передать новой программе. Преимущество использования argv состоит в том, что список параметров может быть построен динамически. Последний элемент вектора должен содержать нулевой адрес.

envp[] вектор указателей на строки, представляющие новые переменные среды для новой программы. Значения элементов этого массива копируются в параметр envp[] функции main() новой программы. Аналогично, environ новой программы указывает на envp[0] новой программы. Последний элемент envp[] должен содержать нулевой адрес.

cnup указатель на вектор указателей на строки, представляющие новые переменные среды новой программы; в действительности то же что и envp[].

arg0 или argv[0], следует устанавливать равным последней компоненте path или параметру file, то есть равным имени загружаемой программы.

Использование argv[0]

Некоторые программы воспринимают нулевой аргумент как значимый параметр.

Так, в Solaris 10, утилиты mv(1) и cp(1) представляют собой жесткие линки на один и тот же файл. В этом можно убедиться, набрав команду:

```
$ ls -li `which mv` `which cp`
    261 -r-xr-xr-x   3 root     bin        26768 янв.  9  2007 /usr/bin/cp
    261 -r-xr-xr-x   3 root     bin        26768 янв.  9  2007 /usr/bin/mv
```

Команда ls –li выводит номер инода (уникального идентификатора) файла. Видно, что оба файла имеют одинаковый номер инода. Это означает, что и /usr/bin/cp, и /usr/bin/mv – жесткие связи одного и того же файла. Таким образом, программа cp(1) определяет, удалять ли старый файл после копирования, именно на основании argv[0].

Аналогично, в стандартной поставке, программы gzip(1) и gunzip(1) (потоковые архиватор и деархиватор) обычно представляют собой один и тот же бинарный файл, который определяет, что ему делать (упаковывать или распаковывать) по имени команды, которой он был запущен, то есть по argv[0].

В некоторых дистрибутивах Linux используется утилита busybox (http://www.busybox.net/), которая, в зависимости от имени, под которым она была запущена, может имитировать большинство стандартных утилит Unix, таких, как ls(1), mv(1), cp(1), rm(1) а также ряд системных сервисов, таких, как crond(1M), telnetd(1M), tftpd(1M), всего более трёхсот разных программ.

Таким образом, неправильное задание arg0 может привести к тому, что запускаемая программа поведёт себя совершенно неожиданным образом.

Исполнение программы (продолжение)

Если exec(2) исполняется успешно, то новая программа не возвращает управление в исходную (исходного образа процесса больше не существует). Если exec(2) возвратил управление, значит вызов завершился неудачей. Например, exec(2) будет неудачным, если требуемая программа не найдена или у вас нет прав на её исполнение, а также если система не может исполнять файлы такого формата.

Современные системы, в том числе Solaris, используют формат загружаемых файлов ELF (Executable and Linking Format). В заголовке файлов этого формата, помимо прочего, указана используемая система команд (x86, x64, SPARC, MIPS, ARM и др.). Разумеется, компьютер с процессором SPARC не может исполнять загрузочные модули x86/x64, а 32-битная версия ОС для x86 не может исполнять загрузочные модули x64.

Кроме бинарных модулей, современные Unix-системы могут исполнять текстовые файлы, если такие файлы начинаются с «магической последовательности» - строки вида #!pathname [arg], например, #!/bin/sh или #!/usr/bin/perl. Если файл начинается с такой строки, система интерпретирует pathname как имя программы-интерпретатора, запускает эту программу, передаёт ей аргументы [arg] (если они были указаны), затем имя файла и затем остальные аргументы exec(2). В результате, если файл с именем pathname действительно является программой-интерпретатором, он рассматривает запускаемый файл как программу на соответствующем языке, например, командный файл shell или программу на языке Perl.

Важно отметить, что анализ «магической последовательности» и запуск интерпретатора осуществляется именно ядром системы, поэтому, если текстовый файл имел атрибут setuid, то ядро запустит интерпретатор с соответствующим значением эффективного идентификатора пользователя. Если бы анализ «магической последовательности» выполнялся библиотечной функцией, смена euid при запуске интерпретатора была бы невозможна.

Любой файл, открытый перед вызовом exec(2), остается открытым, если при помощи fcntl(2) для его дескриптора не был установлен флаг закрыть-по-exec. Это обсуждалось в разделе, посвящённом системным вызовам ввода/вывода.

Для версий exec(2), не передающих среду исполнения в качестве параметра, в качестве новой среды используется массив указателей, на который указывает внешняя переменная environ.

Запуск программ из shell

Командные интерпретаторы или командные оболочки Unix, такие, как sh(1), ksh(1), bash(1) и некоторые другие, часто объединяют под общим названием shell, так как их командные языки очень похожи. Командные языки shell описаны на соответствующих страницах руководства, а также во многих учебных пособиях для пользователей Unix. В действительности, командный язык shell представляет собой полнофункциональный (turing-complete) процедурный интерпретируемый язык программирования с переменными, разрушающим присваиванием, условными операторами, циклами и т. д. Полное изучение языка shell выходит за пределы нашего курса, но знакомство с этим языком полезно для выполнения многих упражнений и правильного понимания поведения системы.

Shell, как и другие программы на языке C, использует exec(2) для исполнения программ.

Shell читает командную строку с терминала или из командного файла, разделяет её на аргументы, затем создаёт дочерний процесс и в этом процессе вызывает exec(2) с соответствующими параметрами (основной процесс shell при этом ждёт завершения дочернего процесса). Первое слово командной строки — это имя программы, которую нужно исполнить, последующие аргументы — это значения argv[1] и последующих элементов вектора argv[]. Если имя программы содержит один или несколько символов /, оно интерпретируется как абсолютное или относительное путевое имя. Если же имя программы не содержит /, то исполняемый файл с таким именем ищется в списке каталогов, заданных в переменной среды PATH, как при использовании execvp(2). В действительности, некоторые командные оболочки (например, bash(1)) не используют execvp(2), а сами выполняют поиск файла по PATH и кэшируют результаты поиска во внутренних хэш-таблицах, что может ускорить исполнение длинных последовательностей команд.

Если один из аргументов команды содержит символы *, ? или [, shell интерпретирует такой аргумент как шаблон имени файла (точный формат шаблона описан на страницах руководства fnmatch(5) и sh(1)). Shell находит все файлы, соответствующие шаблону (если шаблон содержит также символы /, поиск может вестись в других каталогах; так, шаблон */* соответствует всем файлам во всех подкаталогах текущего каталога) и заменяет шаблон на список аргументов, каждый из которых соответствует одному из имён найденных файлов. Если файлов, соответствующих шаблону, не найдено, шаблон передаётся команде без изменений. Если вам нужно передать команде сам шаблон (например, команда find(1) или некоторые архиваторы ожидают шаблон имени файла, который следует найти), соответствующий аргумент необходимо экранировать одиночными или двойными кавычками, например find . -name '*.c' -print.

Важно отметить, что замена шаблонов осуществляется shell'ом, но не системными вызовами exec(2). Для замены шаблонов имён файлов в вашей программе следует использовать библиотечную функцию glob(3C).

Shell имеет встроенные переменные, значения которых представляют собой текстовые строки. Команда VAR=value присваивает переменной VAR значение value. Если переменная VAR не была определена, она будет создана. По умолчанию, встроенные переменные shell не передаются запускаемым программам. Чтобы переменная VAR стала частью среды запускаемых программ, необходимо выполнить команду export VAR. Некоторые оболочки, например, bash(1), допускают гибридный синтаксис команды export VAR=value, т. е. одновременное присваивание значения переменной и её экспорт.

Если вам нужно запустить команду с определённым значением переменной среды, не меняя значение соответствующей переменной shell, это можно сделать с использованием синтаксиса VAR=value cmd [arg] . Например, команда TZ=Asia/Tokyo date выдаст вам время и дату в Токио, не меняя значение переменной TZ.

Исполняемая программа - Пример

Эта программа будет использоваться для демонстрации системных вызовов семейства exec(2) в последующих примерах. Эта программа распечатывает свои аргументы командной строки и переменные среды исполнения.

12-13 Этот цикл распечатывает аргументы командной строки.

16-17 Этот цикл распечатывает переменные среды исполнения.

Файл: newpgm.c

ИСПОЛНЯЕМАЯ ПРОГРАММА - ПРИМЕР

```
 1 #include <stdio.h>
 2 extern char **environ;
 3
 4 /* program to be exec(2)'ed */
 5
 6 main(int argc, char *argv[ ])
 7 {
 8     char **p;
 9         int n;
10
11     printf("My input parameters(argv) are:\n");
12         for (n = 0; n < argc; n++)
13         printf("    %2d: '%s'\n", n, argv[n]);
14
15     printf("\nMy environment variables are:\n");
16         for (p = environ; *p != 0; p++)
17         printf("    %s\n", *p);
18 }
```

```
$ newpgm parm1 parm2 parm3
My input parameters(argv) are:
     0: 'newpgm'
     1: 'parm1'
     2: 'parm2'
     3: 'parm3'

My environment variables are:
    HOME=/uxm2/jrs
    LOGNAME=jrs
    MAIL=/var/mail/jrs
    PATH=/usr/bin:/usr/lbin/:/uxm2/jrs/bin:.
    TERM=5420
    TZ=EST5EDT
```

Использование execl(2) - Пример

Эта программа демонстрирует использование системного вызова execl(2).

9-10 Системный вызов execl(2) использует список параметров.

newpgm имя программы, которую следует исполнить

newpgm значение argv[0] новой программы

parm1 значение argv[1] новой программы

parm2 значение argv[2] новой программы

parm3 значение argv[3] новой программы

(char *)0 конец списка параметров. Он необходим, потому что в C/C++ функция с переменным количеством аргументов (а execl(2) является именно такой функцией) не имеет встроенных в язык средств, чтобы определить, сколько ей было передано аргументов (см. varargs(3EXT)).

12 Исходная программа exec1 распечатает сообщение об ошибке, только если execl(2) завершится неудачно. Это может произойти, например, если требуемая программа не может быть исполнена или был передан неправильный параметр, такой как недопустимый указатель на один из аргументов.

Когда исполняется newpgm, из параметров execl(2) создается новый список argv[]. Переменные среды исполнения наследуются от вызывающего процесса.

Файл: exec1.c

ИСПОЛЬЗОВАНИЕ execl(2) - ПРИМЕР

```
 1 #include <unistd.h>
 2 #include <stdio.h>
 3
 4 main()
 5 {
 6
 7     printf("this is the original program\n");
 8
 9     execl("newpgm", "newpgm", "parm1", "parm2",
10             "parm3", (char *) 0);
11
12      perror("This line should never get printed\n");
13 }
```

```
$ exec1
this is the original program
My input parameters(argv) are:
     0: 'newpgm'
     1: 'parm1'
     2: 'parm2'
     3: 'parm3'

My environment variables are:
    HOME=/uxm2/jrs
    LOGNAME=jrs
    MAIL=/var/mail/jrs
    PATH=/usr/bin:/usr/lbin/:/uxm2/jrs/bin:.
    TERM=5420
    TZ=EST5EDT
```

Использование execv(2) - Пример

Эта программа исполняет новую программу, используя execv(2).

6-8 nargv[] - массив указателей на строки, представляющие собой аргументы новой программы. Последним элементом nargv[] должен быть нулевой адрес, отмечающий конец списка. В этом примере аргументы таковы: "diffnm", "parm1", "parm2" и "parm3".

Замечание: nargv[0] отличается от первого параметра execv - имени запускаемой программы.

13 Второй аргумент execv(2) - адрес массива, содержащего адреса параметров новой программы. Использование списка позволяет динамически формировать этот список в программе.

Переменные среды наследуются от вызывающего процесса.

Файл: exec2.c

ИСПОЛЬЗОВАНИЕ execv(2) - ПРИМЕР

```
 1 #include <unistd.h>
 2 #include <stdio.h>
 3
 4 main()
 5 {
 6     char *nargv[ ] = {
 7                 "diffnm", "parm1", "parm2", "parm3",
 8         (char *) 0 };
 9
10     printf("this is the original program\n");
11
12     execv("newpgm", nargv);
13
14     perror("This line should never get printed\n");
15 }
```

```
$ exec2
this is the original program
My input parameters(argv) are:
     0: 'diffnm'
     1: 'parm1'
     2: 'parm2'
     3: 'parm3'

My environment variables are:
    HOME=/uxm2/jrs
    LOGNAME=jrs
    MAIL=/var/mail/jrs
    PATH=/usr/bin:/usr/lbin/:/uxm2/jrs/bin:.
    TERM=5420
    TZ=EST5EDT
```

Использование execve(2) - Пример

Эта программа исполняет новую программу с помощью execve(2)

9-13 nenv[] - список указателей на новые значения переменных среды исполнения. Он имеет такую же форму, как и nargv[]. Каждый элемент nenv[] указывает на строку, похожую на оператор присваивания shell: "имя=значение"

17 Третий параметр execve(2) - это адрес списка новых переменных среды.

18-19 Закомментирован вызов execle(2). Он приведёт к тому же результату.

Файл: exec3.c

ИСПОЛЬЗОВАНИЕ execve(2) - ПРИМЕР

```
 1 #include <unistd.h>
 2 #include <stdio.h>
 3
 4 main()
 5 {
 6     char *nargv[ ] = {
 7                 "newpgm", "parm1", "parm2", "parm3",
 8           (char *) 0 };
 9     char *nenv[ ]  = {
10                  "NAME=value",
11           "nextname=nextvalue",
12           "HOME=/xyz",
13           (char *) 0 };
14
15     printf("this is the original program\n");
16
17     execve("newpgm", nargv, nenv);
18          /* execle("newpgm", "newpgm", "parm1", "parm2",
19          "parm3", (char *) 0, nenv); */
20
21     perror("This line should never get printed\n");
22 }
```

```
$ exec3
this is the original program
My input parameters(argv) are:
     0: 'newpgm'
     1: 'parm1'
     2: 'parm2'
     3: 'parm3'

My environment variables are:
    NAME=value
    nextname=nextvalue
    HOME=/xyz
```

Использование execvp(2) - Пример

Этот пример использует execvp(2). execvp(2) и execlp(2) осуществляют поиск загружаемого файла программы в соответствии с переменной среды PATH. Вспомните, что PATH - это список директорий, разделённых двоеточием, в которых система должна искать загружаемые файлы.

Файл: exec4.c

ИСПОЛЬЗОВАНИЕ execvp(2) - ПРИМЕР

```
 1 #include <unistd.h>
 2 #include <stdio.h>
 3
 4 main()
 5 {
 6     char *nargv[ ] = {
 7                 "newpgm", "parm1", "parm2", "parm3",
 8         (char *) 0 };
 9
10     printf("this is the original program\n");
11
12     execvp("newpgm", nargv);
13         /* execlp("newpgm", "newpgm", "parm1", "parm2",
14         "parm3", (char *) 0); */
15
16     perror("This line should never get printed\n");
17 }
```

```
$ exec4
this is the original program
My input parameters(argv) are:
     0: 'newpgm'
     1: 'parm1'
     2: 'parm2'
     3: 'parm3'

My environment variables are:
    HOME=/uxm2/jrs
    LOGNAME=jrs
    MAIL=/var/mail/jrs
    PATH=/usr/bin:/usr/lbin/:/uxm2/jrs/bin:.
    TERM=5420
    TZ=EST5EDT
```

Использование fork(2) и exec(2) - Пример

Этот пример представляет программу, которая порождает три процесса, каждый из которых запускает программу echo(1), используя системный вызов execl(2). Обратите внимание, что за каждым вызовом execl(2) следует сообщение об ошибке и завершение процесса. Сообщение будет распечатано, только если вызов execl(2) завершится неудачей.

Важно проверять успешность системных вызовов семейства exec(2), иначе может начаться исполнение нескольких копий исходной программы. В этом примере, если все вызовы exec(2) будут неудачными, может возникнуть восемь копий исходной программы.

Если все вызовы execl(2) были успешными, после последнего fork(2) будет существовать четыре процесса. Порядок, в котором они будут исполняться, невозможно предсказать.

Эта программа демонстрируется так:

```
$ forkexec1
Parent program ending
this is message three
this is message two
this is message one
```

Файл: forkexec1.c

ИСПОЛЬЗОВАНИЕ fork(2) И exec(2) - ПРИМЕР

```
 1 #include <sys/types.h>
 2 #include <unistd.h>
 3 #include <stdlib.h>
 4 #include <stdio.h>
 5
 6 main()
 7 {
 8
 9     if (fork() == 0) {
10                 execl("/bin/echo", "echo", "this is",
11             "message one", (char *) 0);
12         perror("exec one failed");
13         exit(1);
14     }
15         if (fork() == 0) {
16         execl("/bin/echo", "echo", "this is",
17             "message two", (char *) 0);
18         perror("exec two failed");
19         exit(2);
20     }
21         if (fork() == 0) {
22         execl("/bin/echo", "echo", "this is",
23             "message three", (char *) 0);
24         perror("exec three failed");
25         exit(3);
26     }
27
28     printf("Parent program ending\n");
29 }
```

Завершение процесса

Системный вызов exit(2) предназначен для завершения процесса. Он прекращает исполнение вашей программы. В качестве параметра exit(2) передаётся код завершения в диапазоне от 0 до 255. По соглашению, значение 0 означает, что программа завершилась нормально. Значения от 1 до 255 означают, что программа завершилась из-за какой-либо ошибки.

Полезно использовать для каждого типа ошибки свой код завершения. Код завершения может быть получен родительским процессом через системный вызов wait(2). Это будет обсуждаться далее. Код завершения команды, исполненной из shell, доступен как переменная shell ${?}. Это полезно при написании командных файлов, выполняющих ветвление в зависимости от кода завершения, возвращённого командой.

exit(2) осуществляет действия по очистке, такие как закрытие всех открытых файлов и исполнение деструкторов статических переменных C++. Системный вызов _exit(2) сокращает эти действия по очистке. Например, _exit(2) не очищает буфера стандартной библиотеки ввода-вывода.

Вызов _exit(2) необходимо использовать в аварийных ситуациях, например, когда вы подозреваете повреждение памяти вашего процесса и имеете основания предполагать, что сброс буферов стандартной библиотеки может привести к записи в файлы некорректных данных.

В языках C/C++, возврат управления из функции main оператором return эквивалентен вызову exit(2). В действительности, такой возврат приводит к вызову exit(2). В этом можно убедиться, просмотрев ассемблерные исходники стартового файла среды исполнения языка C (crt1.o) или пройдя соответствующий оператор в отладчике (отладчик необходимо переключить из режима показа исходного текста в режим показа деассемблированного кода).

Сигналы

Еще одна возможная причина завершения процесса в Unix — это сигналы. Сигнал — это предоставляемое Unix средство обработки ошибок и исключительных ситуаций, иногда используемое для других целей, например для межпроцессной коммуникации. Подробнее сигналы и их обработка будут рассматриваться в разделе «Сигналы». Сигналы могут возникать:

При ошибках программирования: деление на ноль (SIGFPE), ошибки защиты памяти (SIGSEGV), ошибки обращения к памяти (SIGBUS).

В ответ на действия пользователя: нажатие некоторых клавиш на терминале приводит к посылке сигналов процессам соответствующей терминальной сессии.

В ответ на различные события: разрыв терминальной сессии (SIGHUP), разрыв трубы или сокета (SIGPIPE), завершение операции асинхронного ввода-вывода (настраивается при формировании запроса на ввод-вывод), срабатывание будильника (SIGALRM).

Также сигналы могут программно посылаться одними процессами другим процессам при помощи системных вызовов kill(2) и sigsend(2).

Каждый тип сигнала идентифицируется номером. Стандарт POSIX описывает 32 различных сигнала (нумерация начинается с 1); в ОС Solaris предусмотрено 64 типа сигналов.

Большинство необработанных сигналов приводит к завершению процесса, получившего этот сигнал. Это считается аварийным завершением процесса и отличается от завершения процесса по exit(2) или _exit(2).

Некоторые сигналы, например, SIGTSTP, SIGTTIN, SIGTTOUT, приводят не к завершению процесса, а к его приостановке. Приостановленный процесс находится в специальном состоянии, которое отличается от ожидания в блокирующемся системном вызове или на примитиве синхронизации. Приостановленный процесс может быть продолжен сигналом SIGCONT. Приостановка процессов используется при реализации управления заданиями, которое рассматриваются в разделе «Терминальный ввод-вывод», и отладчиками, которые в нашем курсе не рассматриваются.

Ожидание порожденного процесса

После завершения по exit(2) или по сигналу, процесс переходит в состояние, известное как «зомби». В этом состоянии процесс не исполняется, не имеет пользовательской области, адресного пространства и открытых файлов и не использует большинство других системных ресурсов. Однако «зомби» занимает запись в таблице процессов и сохраняет идентификатор процесса и идентификатор родительского процесса. Эта запись используется для хранения слова состояния процесса, в котором хранится код завершения процесса (параметр exit(2)), если процесс завершился по exit(2) или номер сигнала, если процесс завершился по сигналу.

На самом деле, главным назначением «зомби» является защита идентификатора процесса (pid) от переиспользования. Дело в том, что родительские процессы идентифицируют своих потомков на основе их pid, а ядро может использовать свободные pid для вновь создаваемых процессов. Поэтому, если бы не существовало записей-«зомби», была бы возможна ситуация, когда потомок с pid=21285 завершается, а родитель, не получив код возврата этого потомка, создает новый подпроцесс и система выделяет ему тот же pid. После этого родитель уже не сможет объяснить системе, какой из потомков с pid-21285 его интересует, и не сможет понять, к какому из потомков относится полученное слово состояния.

Также, если «зомби» является последним из группы процессов или сессии, то соответствующая группа процессов или сессия продолжают существовать, пока существует зомби.

В выводе команды ps(1), процессы-зомби отмечаются надписью <defunct> в колонке, где для нормальных процессов выводится имя программы. Также, в форматах вывода, в которых показывается статус исполнения процесса, процессы-зомби имеют статус Z.

Название «зомби» связано с тем, что обычные процессы можно «убить» сигналом, но процессы-зомби на сигналы не реагируют, то есть «убить» их невозможно. Для уничтожения «зомби», родитель такого процесса должен считать его слово состояния системным вызовом wait(2), waitpid(2) или waitid(2). Эти системные вызовы рассматриваются далее. В системах семейства Unix код завершения процесса может быть считан только родителем этого процесса.

Идентификаторы процессов и записи в таблице процессов представляют собой ценные системные ресурсы, поэтому хорошим тоном считается как можно быстрее освобождать ненужные записи. Если ваша программа создает подпроцессы, она должна позаботиться о своевременном сборе их кодов завершения.

Если родительский процесс завершается, все его потомки, как продолжающие исполнение, так и «зомби», усыновляются процессом с pid=1. При нормальной работе системы этот процесс исполняет программу /bin/init, которая большую часть времени проводит в системном вызове wait(2), отслеживая состояние запущенных ею системных процессов. Поэтому коды завершения усыновленных «зомби» быстро считываются и соответствующие записи в таблице процессов освобождаются.

Ожидание порожденного процесса - wait(2)

Процесс может синхронизоваться с завершением порожденного процесса с помощью системного вызова wait(2). Если вызывается wait(2), а у процесса нет ни одного незавершенного подпроцесса, wait(2) немедленно возвращает -1 и устанавливает errno равной ECHILD. Иначе вызывающий процесс:

1. засыпает, если существует незавершившийся подпроцесс.

2. возвращает управление немедленно, если существует подпроцесс, который уже завершился, но к нему не применялся wait(2).

В обоих вышеперечисленных случаях, wait(2) возвращает идентификатор завершившегося подпроцесса. Кроме того, слово состояния подпроцесса сохраняется в параметре wait(2).
3. возвращает значение -1 и устанавливает errno в EINTR, если wait(2) был прерван сигналом. Если это произошло, а вы по-прежнему хотите дождаться завершения подпроцесса, вы должны еще раз вызвать wait(2).

Параметр wait(2) - указатель на целое число, по которому размещается слово состояния подпроцесса. Если вас не интересует слово состояния подпроцесса, вы можете использовать нулевой указатель.

Подпроцесс может завершиться штатно (вызовом exit(2)), или он может быть убит необработанным сигналом. Эти два варианта можно различить анализом содержимого младших 16 бит слова состояния, которое формирует wait(2). Младший байт слова состояния содержит номер сигнала или ноль, если процесс был завершен по exit(2). Вспомним, что нумерация сигналов начинается с 1, то есть сигнала с номером ноль не существует. Второй по старшинству байт содержит параметр exit(2) или 0, если процесс завершился по сигналу. Содержимое старших двух байтов целочисленного значения не определено, на практике оно обычно 0. В заголовочном файле <wait.h> определено несколько макросов, используемых для анализа содержимого слова состояния. Эти макросы описаны на странице руководства wstat(2) и будут обсуждаться далее.

Если процесс ожидает завершения нескольких подпроцессов, то порядок, в котором они завершатся, неизвестен, а порядок получения слов состояния может не соответствовать порядку их завершения. Поэтому ваша программа не должна зависеть от предполагаемого порядка завершения подпроцессов.

Слово состояния wait(2)

Когда процесс ожидает получения слова состояния своих подпроцессов с использованием wait(2) или waitpid(3C), то это слово может быть проанализировано при помощи макросов, определенных в <sys/wait.h>. Эти макросы обсуждаются на странице руководства wstat(5).

WIFEXITED(stat) Ненулевое значение, если это слово состояния получено от подпроцесса, завершившегося по exit(2).

WEXITSTATUS(stat) Если значение WIFEXITED(stat) ненулевое, этот макрос возвращает код завершения, переданный подпроцессом вызову exit(2), или возвращенный его функцией main(), иначе код возврата не определен.

WIFSIGNALED(stat) Возвращает ненулевое значение, если это слово состояния получено от подпроцесса, который был принудительно завершен сигналом.

WTERMSIG(stat) Если значение WIFSIGNALED(stat) ненулевое, этот макрос возвращает номер сигнала, который вызвал завершение подпроцесса, иначе код возврата не определен.

WIFSTOPPED(stat) Возвращает ненулевое значение, если слово состояния получено от приостановленного подпроцесса (wait(2) не реагирует на приостановленные подпроцессы, такое слово состояния может быть получено только вызовом waitpid(2)).

WSTOPSIG(stat) Если значение WIFSTOPPED(stat) ненулевое, этот макрос возвращает номер сигнала, который вызвал приостановку подпроцесса, иначе код возврата не определен.

WIFCONTINUED(stat) Возвращает ненулевое значение, если слово состояния получено от процесса, продолжившего исполнение (wait(2) не реагирует на приостановленные подпроцессы, такое слово состояния может быть получено только вызовом waitpid(2)).

WCOREDUMP(stat) Если значение WIFSIGNALED(stat) ненулевое, этот макрос возвратит ненулевое значение, если был создан посмертный дамп памяти (core-файл) завершившегося подпроцесса. Факт создания дампа памяти определяется по номеру сигнала; завершение по некоторым сигналам, таким, как SIGSEGV и SIGFPE, всегда приводит к созданию дампа памяти, завершение по остальным сигналам никогда не создает такой дамп.

Ожидание одного процесса - Пример

Эта программа показывает, как ожидать завершения одного подпроцесса, и работает следующим образом:

13-17 Создается подпроцесс, который распечатывает свой идентификатор, идентификатор родительского процесса и код завершения, который будет передан exit(2) в следующем операторе. Родительский процесс запоминает идентификатор подпроцесса в переменной pid.

Замечание: Проверка успешности исполнения fork(2) для краткости опущена, но на практике всегда надо проверять значение, возвращенное системным вызовом.

19 Родительский процесс объявляет, что он ожидает завершения своего подпроцесса.

21 Родитель ожидает завершения порожденного подпроцесса. Может возникнуть одна из двух ситуаций. В одном случае, порожденный процесс может завершиться раньше, чем родитель вызовет wait(2). Тогда wait(2) возвращает управление немедленно, сохранив слово состояния в своем параметре. В другом случае, подпроцесс может начать исполнение длинной и медленной программы. Тогда родитель прекращает исполнение (засыпает), пока

подпроцесс не завершится. Так же, как и в первом случае, формируется слово состояния. Возвращаемое значение, идентификатор завершившегося подпроцесса, сохраняется в переменной ret.

23 Родитель распечатывает значение, возвращенное wait(2). Оно должно соответствовать значению, распечатанному в строке 19.

Замечание: Если код возврата отрицательный, может возникнуть ситуация потери знака: система сохраняет в слове состояния только младший байт кода возврата, а макрос WEXITSTATUS интерпретирует его как беззнаковое число, поэтому отрицательные коды возврата будут преобразованы в положительные числа в диапазоне от 127 до 255.

24-26 WIFEXITED возвращает ненулевое значение, если подпроцесс завершился нормально. Затем макросом WEXITSTATUS вычисляется код завершения подпроцесса.

28-31 WIFSIGNALED возвращает ненулевое значение, если подпроцесс был прерван сигналом. WTERMSIG используется для вычисления номера сигнала.

Пример демонстрируется следующим образом:

```
$ wait1
child: pid=10701 ppid=10700 exit_code=1
parent: waiting for child=10701
parent: return value=10701
child's exit status is: 1
```

Файл: wait1.c

ОЖИДАНИЕ ОДНОГО ПРОЦЕССА - ПРИМЕР

```
 1 #include <stdio.h>
 2 #include <sys/types.h>
 3 #include <unistd.h>
 4 #include <stdlib.h>
 5 #include <wait.h>
 6 #define EXIT_CODE 1
 7
 8 main()
 9 {
10     pid_t pid, ret;
11     int status;
12
13     if ((pid = fork()) == 0){    /* child */
14     printf("child: pid=%ld ppid=%ld exit_code=%d\n",
15         getpid(), getppid(), EXIT_CODE);
16     exit(EXIT_CODE);
17     }
18
19     printf("parent: waiting for child=%ld\n", pid);
20
21     ret = wait(&status);
22
23     printf("parent: return value=%ld\n", ret);
24     if (WIFEXITED(status))
25     printf("child's exit status is: %d\n",
26         WEXITSTATUS(status));
27     else
28     if (WIFSIGNALED(status))
29         printf("signal is: %d\n",
30         WTERMSIG(status));
31
32     exit(0);
33 }
```

Ожидание нескольких процессов - Пример

В этом примере родительский процесс порождает два процесса, каждый из которых запускает команду echo(1). Затем родитель ждет завершения своих потомков, прежде чем продолжить свое исполнение.

Строки 17 и 18 показывают, как использовать wait(2) в этой ситуации. wait(2) вызывается из цикла while. Он вызывается три раза. Первые два вызова ожидают завершения процессов-потомков. Последний вызов возвращает неуспех, так как некого больше ожидать. Заметьте, что код завершения потомков здесь игнорируется.

Ниже эта программа исполняется два раза. Порядок исполнения трех процессов непредсказуем.

```
$ wait2
this is message one
parent: waiting for children
this is message two
parent: all children terminated

$ wait2
this is message two
this is message one
parent: waiting for children
parent: all children terminated
```

Файл: wait2.c

ОЖИДАНИЕ НЕСКОЛЬКИХ ПРОЦЕССОВ - ПРИМЕР

```
 1 #include <sys/types.h>
 2 #include <unistd.h>
 3 #include <stdlib.h>
 4 #include <wait.h>
 5 #include <stdio.h>
 6
 7 main()
 8 {
 9     if (fork() == 0)    /* child */
10                 execl("/bin/echo", "echo", "this is",
11               "message one", (char *) 0);
12      if (fork() == 0)    /* child */
13                 execl("/bin/echo", "echo", "this is",
14               "message two", (char *) 0);
15     printf("parent: waiting for children\n");
16
17     while (wait(0) != -1)
18                 ;       /* null */
19
20     printf("parent: all children terminated\n");
21         exit(0);
22 }
```

Ожидание нескольких процессов - Пример (Улучшенный)

В этом примере, как и в предыдущем, родитель также ждет завершения нескольких потомков. Кроме того, родитель предпринимает специальные действия для каждого из потомков и распечатывает код завершения каждого из них.

12-13 Первый подпроцесс исполняет команду date(1).

14-15 Второй подпроцесс исполняет date(1) с неправильной опцией.

17-24 Цикл while ожидает завершения обоих подпроцессов. Заметьте, как идентификатор завершившегося подпроцесса присваивается переменной pid. Внутри цикла выбирается оператор печати, соответствующий этому идентификатору. Заметьте также, что эта программа не зависит от порядка завершения подпроцессов.

Этот пример демонстрируется следующим образом:

```
$ wait3
Sun Oct  6 10:25:39 EDT 1990
parent: waiting for children
date: bad conversion
parent: first child: 0
parent: second child: 1
parent: all children terminated
```

Файл: wait3.c

ОЖИДАНИЕ НЕСКОЛЬКИХ ПРОЦЕССОВ - ПРИМЕР (УЛУЧШЕННЫЙ)

```
 1 #include <sys/types.h>
 2 #include <unistd.h>
 3 #include <stdlib.h>
 4 #include <wait.h>
 5 #include <stdio.h>
 6
 7 main()
 8 {
 9     pid_t child1, child2, pid;
10         int status;
11
12     if ((child1 = fork()) == 0)
13                 execl("/bin/date", "date", (char *) 0);
14     if ((child2 = fork()) == 0)
15                 execl("/bin/date", "date", "-x", (char *) 0);
16     printf("parent: waiting for children\n");
17         while ((pid = wait(&status)) != -1) {
18         if (child1 == pid)
19             printf("parent: first child: %d\n",
20                 WEXITSTATUS(status));
21         else if (child2 == pid)
22             printf("parent: second child: %d\n",
23                 WEXITSTATUS(status));
24     }
25         printf("parent: all children terminated\n");
26
27     exit(0);
28 }
```

Вызов команды shell из программы на языке C - Пример

Этот пример демонстрирует исполнение команды shell из программы на C. Он показывает функцию общего назначения, которая принимает в качестве аргумента произвольную команду shell. Функция создает подпроцесс и исполняет shell, передав ему свой параметр в качестве команды. Этот пример похож на библиотечную функцию system(3C).

15-18 Подпроцесс исполняет shell. Флаг -c, переданный shell'у означает, что следующий аргумент - это команда.

19-20 Цикл while ожидает завершения определенного подпроцесса, а именно запущенного shell'а. Причина использования цикла состоит в том, что может существовать несколько завершившихся подпроцессов. Функция command() может быть использована в большой программе, которая создает другие подпроцессы. Кроме того, цикл прекращается, если системный вызов wait(2) завершается неуспехом. Например, wait(2) возвращает -1 и устанавливает errno в EINTR, если он был прерван перехваченным сигналом. Кроме того, он может возвратить -1, если fork(2) в строке 15 завершился неуспехом.

24 Код завершения возвращается в вызвавшую функцию.

27-33 Эта функция main() является тестовым драйвером для command(). main() исполняет некоторые команды shell и распечатывает код возврата функции command(). Эта программа компилируется командой

```
cc -DDEBUG -o command command.c
```

Эта техника используется для включения драйверной функции main() для тестирования и отладки функции.

Этот пример демонстрируется так:

```
$ command
Sun Oct  6 12:04:04 EDT 1990
0
date: bad conversion
1
```

Файл: command.c

ВЫЗОВ КОМАНДЫ ИЗ СИ-ПРОГРАММЫ - ПРИМЕР

```
 1 #include <sys/types.h>
 2 #include <unistd.h>
 3 #include <stdlib.h>
 4 #include <wait.h>
 5 #include <stdio.h>
 6 int command(char *);
 7
 8 /* run a shell command from C program */
 9
10 int command(char *cmd)
11 {
12     pid_t chpid, w;
13         int status;
14
15     if ((chpid = fork()) == 0) {
16                 execlp("sh", "sh", "-c", cmd, (char *) 0);
17         exit(127);
18     }
19         while ((w = wait(&status)) != chpid && w != -1)
20         ;       /* null */
21     if (w == -1)
22                 return(-1);
23     else
24                 return(WEXITSTATUS(status));
25 }
26
27 #if DEBUG
28 main()          /* test command() function */
29 {
30     printf("%d\n", command("date > Date; cat Date"));
31         printf("%d\n", command("date -x"));
32 }
33 #endif
```

Ожидание изменения состояния подпроцесса

waitid(2) приостанавливает вызывающий процесс, пока один из его подпроцессов не изменит состояние. Изменения состояния включают в себя не только завершение, но также остановку по сигналу и продолжение работы после такой остановки. Если изменения происходили до вызова waitid(2), он возвращает управление немедленно. Еще одно отличие waitid(2) от wait(2) состоит в том, что waitid(2) позволяет указывать, каких потомков следует ожидать, а wait(2) всегда ждет всех потомков. Параметры idtype и id определяют, какие из подпроцессов должны обрабатываться:

idtype	какие подпроцессы обрабатывать
P_PID	подпроцесс с идентификатором, равным id
P_PGID	подпроцесс с идентификатором группы процессов, равным id
P_ALL	любой подпроцесс; id игнорируется

Выходной параметр infop содержит информацию о причине изменения состояния подпроцесса. <wait.h> включает файл <sys/siginfo.h>, который содержит описание структуры siginfo_t. Страница руководства SIGINFO(5) описывает поля siginfo_t, относящиеся к waitid. Это:

```
int si_signo    /* always equals SIGCLD for waitid */
int si_code     /* contains a code identifying the cause of signal */
int si_status   /* equals exit value or signal */
pid_t si_pid    /* child process id */
```

Параметр options используется для задания того, какие изменения состояния следует принимать во внимание. Он формируется побитовым ИЛИ следующих значений:

WEXITED ожидать нормального завершения подпроцесса (по exit(2)).

WTRAPPED ожидать прерывания трассировки или точки останова в отлаживаемом процессе (ptrace(2)).

WSTOPPED ожидать, пока подпроцесс будет остановлен получением сигнала.

WCONTINUED ожидать, пока остановленный подпроцесс не начнет исполнение.

WNOHANG немедленно возвращать управление, если нет немедленно доступного слова состояния (ни один из подпроцессов не менял свое состояние). Использование этого флага приводит к "ожиданию" без блокировки, в режиме опроса. Это может быть использовано для динамического наблюдения за изменением состояния подпроцессов.

WNOWAIT сохранить подпроцесс в таком состоянии, что его слово состояния может быть получено повторным вызовом wait. Этот флаг означает неразрушающее использование waitid(2). Например, этот флаг позволяет процессу опросить состояние своего потомка, но не уничтожает процесс- «зомби», если потомок уже был таковым, так что группа процессов этого потомка по прежнему будет существовать. Поэтому другие процессы по-прежнему могут присоединяться к этой группе.

Ожидание изменения состояния подпроцесса - Пример 1

Следующая страница показывает пример использования waitid(2), который работает следующим образом:

10 Объявляется структура siginfo_t.

12-15 Запускается подпроцесс. Этот процесс спит случайное число секунд, а затем завершается с кодом 5.

sleeper.c:

```
 1 #include <sys/types.h>
 2 #include <unistd.h>
 3 #include <stdlib.h>
 4
 5 main()
 6 {
 7 srand(getpid());
 8 sleep(rand()%10 +1);
 9 exit(5);
10 }
```

17 Родитель ожидает завершения всех своих потомков, используя опции P_ALL и WEXITED.

18 Для значения, сформированного waitid(2), поле si_signo всегда будет равно SIGCLD (значение 18).

19 Значения si_code, связанные с SIGCLD, определены в <sys/siginfo.h>. Эти значения таковы:

```
#define CLD_EXITED      1       /* child has exited */
#define CLD_KILLED      2       /* child was killed */
#define CLD_DUMPED      3       /* child has coredumped */
#define CLD_TRAPPED     4       /* traced child has stopped
*/
#define CLD_STOPPED     5       /* child has stopped on
signal */
#define CLD_CONTINUED   6       /* stopped child has
continued */
```

20 Если si_signo равно SIGCLD и si_code равно CLD_EXITED, то si_status будет равно коду завершения подпроцесса. Иначе, если si_signo равно SIGCLD, а si_code не равно CLD_EXITED, то si_status будет равно номеру сигнала, который вызвал изменение состояния

процесса. В этом примере подпроцесс нормально завершается с кодом 5, так что si_status имеет значение 5.

```
$ waitid parent: waiting for child : 8370  child  signal  no:  18
child signal code: 1 child status: 5 parent: child completed $
```

Файл: waitid1.c

ОЖИДАНИЕ ИЗМЕНЕНИЯ СОСТОЯНИЯ ПОДПРОЦЕССА - ПРИМЕР 1

```
 1 #include <sys/types.h>
 2 #include <unistd.h>
 3 #include <stdlib.h>
 4 #include <wait.h>
 5 #include <stdio.h>
 6
 7 main()
 8 {
 9     pid_t child1;
10     siginfo_t status;
11
12     if ((child1 = fork()) == 0) {
13     execl("sleeper", "sleeper", (char *) 0);
14     exit(1);
15     }
16     printf("parent: waiting for child : %ld\n",child1);
17     waitid(P_ALL, 0, &status, WEXITED);
18     printf("child signal no: %d\n", status.si_signo);
19     printf("child signal code: %d\n", status.si_code);
20     printf("child status: %d\n", status.si_status);
21
22     printf("parent: child completed\n");
23 }
```

Ожидание изменения состояния подпроцесса - Пример 2

На следующей странице приведен другой пример использования waitid(2), который работает так:

11 Объявляется структура siginfo_t. Она объявляется как static, поэтому она будет проинициализирована нулями.

15-18 Запускается подпроцесс. Этот процесс исполняет программу sleeper из предыдущего примера, которая спит случайное число секунд и завершается с кодом 5. Его идентификатор сохраняется в переменной child1.

20-21 Родительский процесс ожидает завершения определенного подпроцесса, используя опцию WEXITED. Опция WNOHANG заставляет waitid не приостанавливать исполнение вызвавшего процесса, если статус child1 не доступен немедленно. Эта опция используется для опроса завершения подпроцесса. waitid возвращает ноль, если он дождался подпроцесса или из-за опции WNOHANG.

22 Если si_pid остается нулевым, waitid возвратил управление из-за WHOHANG. Если si_pid равен идентификатору подпроцесса, то waitid возвратил статус этого подпроцесса.

23-27 Делается MAXTRIES попыток получить состояние завершения подпроцесса.

28-32 После MAXTRIES попыток подпроцессу посылается сигнал SIGKILL.

34 Распечатывается количество попыток получить статус.

35 Поле si_signo для waitid всегда будет равно SIGCLD (значение 18).

36 Поле si_code будет равно CLD_EXITED (значение 1), если подпроцесс нормально завершился. Оно будет равно CLD_KILLED (значение 2), если подпроцесс получил сигнал SIGKILL.

37-40 Если подпроцесс нормально завершился, si_status равен его коду завершения. Если он был убит сигналом, si_status будет равен номеру сигнала, вызвавшего завершение процесса. В этом случае, номер сигнала будет SIGKILL (значение 9).

```
$ waitid2 parent: waiting for child 8291 sending signal to  child
tries  =  8 child signal no: 18 child signal code: 2 child signal is:
9 parent: child completed $
```

Файл: waitid2.c

ОЖИДАНИЕ ИЗМЕНЕНИЯ СОСТОЯНИЯ ПОДПРОЦЕССА - ПРИМЕР 2

```
...
 9 main()
10 {
11     static siginfo_t stat;
...
15     if ((child1 = fork()) == 0) {
16    execl("sleeper", "sleeper", (char *) 0);
17    exit(1);
18     }
19     printf("parent: waiting for child %ld\n", child1);
20     while (waitid(P_PID, child1, &stat,
21    WNOHANG|WEXITED) != -1) {
22    if (stat.si_pid == 0) {
23        if (try < MAXTRIES) {
24       try++;
25       sleep(1);
26       continue;
27        }
28        else {
29       printf("sending signal to child\n");
30       kill(child1, SIGKILL);
31       continue;
32        }
33    }
34    printf("tries = %d\n", try);
35    printf("child signal no: %d\n", stat.si_signo);
36    printf("child signal code: %d\n", stat.si_code);
37    if (stat.si_code == CLD_EXITED)
38        printf("exit status is: %d\n", stat.si_status);
39    else
40        printf("child signal is: %d\n", stat.si_status);
41     }
42     printf("parent: child completed\n");
43 }
```

Ожидание изменения состояния подпроцесса

waitpid(2) приостанавливает исполнение вызывающего процесса, пока один из его потомков не изменит состояние. Если такое изменение произошло до вызова waitpid(2), он возвращает управление немедленно. pid задает набор подпроцессов, для которых запрашивается состояние. Вызов waitpid(2) требует меньше дополнительного кода для использования, чем waitid(2) (в частности, не требуется создавать структуру siginfo_t), но не позволяет делать некоторые вещи, которые можно сделать при помощи waitid(2)

pid	состояние запрашивается
-1	для всех подпроцессов
>0	для подпроцесса с идентификатором pid
0	для любого подпроцесса с тем же идентификатором группы, что у вызывающего процесса
<-1	для любого подпроцесса, чей идентификатор группы процессов равен -pid

Параметр options формируется побитовым ИЛИ следующих значений, которые описаны в <sys/wait.h>

WNOHANG то же значение, что и для waitid(2).

WUNTRACED то же, что WSTOPPED для waitid(2).

WCONTINUED то же, что и для waitid(2).

WNOWAIT то же, что и для waitid(2).

Функция waitpid(2) эквивалентна вызову waitid(2) с добавлением опций WEXITED | WTRAPPED к значению соответствующего параметра waitpid(2). Если передан ненулевой указатель stat_loc, то слово состояния подпроцесса будет сохранено по этому указателю. Затем полученное значение может быть проанализировано макросами из wstat(5).

Подпрограмма, исполняемая при завершении

atexit(3C) используется для определения функции, которая должна быть вызвана при нормальном завершении программы. Нормальным завершением в данном случае считается вызов функции exit(2) или возврат из функции main(), ненормальным — завершение по сигналу или по _exit(2).

atexit(3C) гарантируют пользователю возможность зарегистрировать по крайней мере 32 таких функции, которые будут вызываться в порядке, обратом их регистрации.

Стандартная библиотека языка C сама использует atexit(3C) для выполнения действий при завершении программы. Наиболее важным из таких действий является сброс буферов потоков буферизованного ввода-вывода. Обработчики, используемые библиотекой для собственных нужд, не занимают 32 обработчка, гарантированные пользователю.

Деструкторы глобальных и статических переменных C++ вызываются через тот же механизм, что и atexit(3C).

Подпрограмма, вызываемая при завершении - Пример

Этот пример иллюстрирует использование библиотечной функции atexit(3C).

13 Библиотечная функция atexit(3C) вызывается, чтобы зарегистрировать функцию finish для исполнения при выходе из программы.

20-27 Родительский процесс ожидает завершения своих потомков.

28 Родитель завершается системным вызовом exit(2). При этом вызывается функция finish.

31-35 Функция finish. Она автоматически вызывается при завершении программы.

Файл: atexit.c

ПОДПРОГРАММА, ВЫЗЫВАЕМАЯ ПРИ ЗАВЕРШЕНИИ - ПРИМЕР

```
 1 #include <sys/types.h>
 2 #include <stdlib.h>
 3 #include <unistd.h>
 4 #include <wait.h>
 5 #include <stdio.h>
 6 static void finish(void);
 7
 8 main()
 9 {
10     pid_t child1, child2, pid;
11         int status;
12
13     atexit(finish);
14         if ((child1 = fork()) == 0)
15         execl("/bin/date", "date", (char *) 0);
16     if ((child2 = fork()) == 0)
17                 execl("/bin/date", "date", "-x", (char *) 0);
18
19     printf("parent: waiting for children\n");
20         while ((pid = wait(&status)) != -1) {
21         if (child1 == pid)
22             printf("parent: first child: %d\n",
23                 WEXITSTATUS(status));
24         else if (child2 == pid)
25             printf("parent: second child: %d\n",
26                 WEXITSTATUS(status));
27     }
28         exit(0);
29 }
30
31 static void finish(void)
32 {
33     /* additional processing */
34         printf("parent: all children terminated\n");
35 }
```

6. УПРАВЛЕНИЕ ФАЙЛАМИ

Обзор

В разделе, посвященном вводу/выводу, вы узнали как открывать файлы, читать из них и писать в них. В этом разделе вы узнаете, как получить доступ к управляющей информации о файле и к информации о состоянии файла. Вы также узнаете, как можно изменить часть этой информации. Управляющая информация и информация о состоянии файла хранится в отдельной структуре данных, называемой инод (inode). Каждый инод идентифицируется номером, уникальным в пределах файловой системы. Любой файл единственным образом определяется в системе своим inode-номером и номером устройства файловой системы, содержащей этот файл. Инод содержит такую информацию, как размер файла, права доступа, владелец, тип файла, различные временные отметки и т. д. Кроме того, файловая система хранит в иноде информацию о размещении блоков файла на носителе, но эта информация пользователю напрямую не доступна.

В этом разделе описываются системные вызовы, которые получают и изменяют информацию в inode. Эти системные вызовы не работают с данными, содержащимися файле. Кроме того вы изучите системные вызовы, которые устанавливают максимальный размер файла и изменяют права доступа существующего файла.

Доступность файла - access(2)

Системный вызов access(2) определяет, доступен ли файл для чтения, модификации, или исполнения и просто его существование. В отличие от других системных вызовов, которые проверяют права доступа файла, access(2) использует реальные идентификаторы пользователя и группы вызывающего процесса, чтобы проверить права доступа файла. Для проверки прав доступа от имени эффективного идентификатора, необходимо использовать аналогичный вызов eaccess(2)

Аргументы access(2):

path Абсолютное или относительное путевое имя файла. Как для всех остальных системных вызовов, каждая директория из путевого имени должна быть доступна на поиск. Иначе системный вызов будет неудачным.

amode Этот аргумент конструируется с помощью побитового ИЛИ из следующих символьных констант, описанных в <unistd.h>:

режим	проверяет
R_OK	чтение
W_OK	изменение
X_OK	исполнение (поиск)
F_OK	существование

Вы можете проверить существование файла, даже если у вас нет прав на доступ к этому файлу. Для директорий право на исполнение означает право на поиск в этой директории.

Проверка прав доступа при помощи access(2) гораздо дешевле, чем аналогичная проверка при помощи open(2).

Возвращаемое значение access(2) указывает, разрешен ли запрашиваемый доступ.

Доступность файла - Пример

Эта программа определяет, доступен ли файл для чтения, изменения, исполнения и просто его существование. Она работает следующим образом:

3 Этот препроцессорный макрос определяет количество элементов в массиве table[]. Поскольку препроцессорный макрос — это просто текстовая подстановка, его определение может стоять перед определением самого массива.

8-16 Эта таблица используется для перевода режимов доступа на английский.

18-24 Этот цикл проверяет четыре режима доступа файла.

19-21 Если вызов access(2) успешен для данного режима, печатается соответствующее сообщение.

22-23 Если access(2) неуспешен, то сообщение об ошибке выводится с помощью perror(3).

Это пример демонстрируется следующим образом:

```
$ access access
exists - ok
execute - ok
write: Text file busy
read - ok
$ access /etc/passwd
exists - ok
execute: Permission denied
write: Permission denied
read - ok
```

В первом случае программа используется для проверки прав доступа исполняемого программного файла. Система не допускает доступа на изменение к файлу, который исполняется в данный момент, даже если у вас есть права на запись в него. Во втором случае программа используется для проверки прав доступа файла /etc/passwd.

Файл: access.c

ДОСТУПНОСТЬ ФАЙЛА - ПРИМЕР

```
 1 #include <stdio.h>
 2 #include <unistd.h>
 3 #define   TABLESIZ   (sizeof(table)/sizeof(*table))
 4
 5 main(int argc, char *argv[])
 6 {
 7      int i;
 8      static struct {
 9       int mode;
10        char *text;
11      } table[ ] = {
12       { F_OK, "exists" },
13       { X_OK, "execute" },
14       { W_OK, "write" },
15       { R_OK, "read" }
16      };
17
18      for (i = 0; i < TABLESIZ; i++) {
19       if (access(argv[1], table[i].mode) != -1)
20           printf("%s - ok\n",
21                table[i].text);
22       else
23            perror(table[i].text);
24      }
25 }
```

Получение и установка ограничений для пользователя

Системный вызов ulimit(2) используется, чтобы определять и устанавливать некоторые ограничения. Этот вызов аналогичен системным вызовам getrlimit(2)/setrlimit(2), но появился гораздо раньше и поддерживается для совместимости со старыми программами. Рекомендуется использовать getrlimit(2)/setrlimit(2). Встроенная команда shell ulimit(1) также может использоваться для установки этих ограничений.

Аргумент cmd может принять значение одной из следующих символьных констант:

UL_GETFSIZE Возвращает текущее ограничение процесса на максимальный размер файла. Размер измеряется в блоках по 512 байт.

UL_SETFSIZE Устанавливает ограничение на размер файла. Только суперпользователь может увеличить это ограничение. Остальные могут только уменьшить его. Может быть полезно ограничить размер файла при отладке программы, которая создает файлы или удлиняет существующие файлы.

UL_GMEMLIM Возвращает максимально допустимое значение границы выделяемой памяти. Это значение можно использовать при проверке того, что программа пытается получить память с помощью brk(2) или sbrk(2) больше, чем допустимо. Изменить соответствующий параметр при помощи ulimit(2) невозможно.

UL_GDESLIM Возвращает ограничение, устанавливаемое программно при конфигурации системы, на число файлов, которые процесс может одновременно держать открытыми.

newlimit используется при cmd равном UL_SETFSIZE. Это новый размер файла в блоках.

Получение и установка маски создания файла

Системный вызов umask(2) используется, чтобы установить параметр cmask (в некоторых документах этот параметр также называется umask): маску ограничения прав доступа к создаваемым файлам или маску создания файла. Параметр cmask является частью окружения процесса и упоминался в разделе «Файловый ввод-вывод». Shell использует umask(2) при исполнении встроенного оператора umask(1).

Аргумент cmask - это новое значение маски создания файла. Аргумент, задающий права доступа для open(2), модифицированный с помощью cmask, используется для получения прав доступа файла при его создании. Права вычисляются по форме mode & (^cmask), где mode — третий параметр open(2). Таким образом, биты, установленные в cmask, не будут присутствовать в правах доступа.

Например, если вы хотите разрешить всем в вашей группе и всем остальным читать, но не писать в ваши файлы, тогда вам следует установить cmask равным 022 (в языке C числовые константы, начинающиеся с 0, интерпретируются как восьмеричные).

Установка маски создания файла - Пример

Этот пример демонстрирует, как использовать umask(2). Первый аргумент этой программы - новая маска создания. Остальные аргументы — команда, которая будет исполняться как подпроцесс.

12 Значение новой маски (представленное как восьмеричное число) преобразуется в long с помощью библиотечной функции strtol(3).

13 Устанавливается новое значение маски создания. umask(2) возвращает старое значение.

14-17 Печатается новое и старое значение маски создания файла.

19-23 Создается порожденный процесс. Он наследует родительскую маску. Подпроцесс исполняет программу, заданную в arg[2].

Программа демонстрируется следующим образом:

```
$ umask
0022
$ who > file1
$ setumask 027 sort -o file2 < file1
Old filemode creation mask: 0022
New filemode creation mask: 0027
$ ls -l file?
-rw-r--r--   1 imr     ustg        1258 Apr  3 14:59 file1
-rw-r-----   1 imr     ustg        1258 Apr  3 15:00 file2
$ umask
0022
```

Опция -o функции sort(1) указывает на файл вывода. Значение маски, установленное с помощью umask(2), действует только во время исполнения программы и не меняет cmask родительского процесса shell, что демонстрируется повторным вызовом команды umask(1).

Файл: setumask.c

УСТАНОВКА МАСКИ СОЗДАНИЯ ФАЙЛА - ПРИМЕР

```
 1 #include <sys/types.h>
 2 #include <stdio.h>
 3 #include <fcntl.h>
 4 #include <stdlib.h>
 5 #include <sys/stat.h>
 6 #include <unistd.h>
 7
 8 main(int argc, char *argv[])
 9 {
10     mode_t newmask, oldmask;
11
12     newmask = strtol(argv[1], (char **) NULL, 8);
13         oldmask = umask(newmask);
14     printf("Old filemode creation mask: %04o\n",
15                 oldmask);
16     printf("New filemode creation mask: %04o\n",
17                 newmask);
18
19     if (fork() == 0) {
20                 execvp(argv[2], &argv[2]);
21         perror(argv[2]);
22         exit(127);
23     }
24 }
```

Определение атрибутов файла

Системный вызов stat(2) используется для получения информации об атрибутах файла. Для просмотра атрибутов не нужно иметь права доступа на чтение, изменение и исполнение для исследуемого файла. Однако, все директории в путевом имени файла должны быть доступны на поиск.

Системный вызов fstat(2) используется для уже открытых файлов. Системный вызов lstat(2) используется для символических связей (понятие символической связи рассматривается в разделе «Управление директориями»). Если файл не является символической связью, lstat(2) ведет себя так же, как stat(2).

У stat(2), fstat(2) и lstat(2) следующие аргументы:

path Путевое имя файла или символической связи. Используется для системных вызовов stat(2) и lstat(2).

fildes Дескриптор открытого файла, используется для системного вызова fstat(2).

buf Указатель на структуру stat. Системный вызов stat(2) заполняет структуру stat информацией о файле. Список полей этой структуры приводится на следующей странице

Атрибуты файла

Структура stat используется для получения информации об атрибутах файла. Поля структуры заполняются системными вызовами stat(2), fstat(2) и lstat(2).

У структуры stat следующие поля:

st_dev Это поле идентифицирует файловую систему, которая содержит заданный файл. Это значение можно использовать как аргумент ustat(2), чтобы получить больше информации о файловой системе.

st_ino Inode-номер заданного файла. Файл однозначно определяется с помощью st_dev и st_ino.

st_mode Это поле содержит биты прав доступа к файлу, тип файла и специальные биты. Это поле обсуждается более подробно на следующих страницах раздела. Биты доступа и специальные биты могут быть изменены с помощью системного вызова chmod(2).

st_nlink Число жестких связей заданного файла. Другими словами, это число ссылающихся на заданный файл записей в директориях. Это понятие подробнее рассматривается в разделе «Управление директориями». Поле изменяется с помощью системных вызовов link(2) и ulink(2).

st_uid Пользовательский идентификатор владельца файла. Он изменяется с помощью системных вызовов chown(2) и lchown(2) (для символических связей).

st_gid Идентификатор группы заданного файла. Он также изменяется с помощью системных вызовов chown(2) и lchown(2) (для символических связей).

st_rdev Это поле применяется только для байт- и блок-ориетированных специальных файлов и является идентификатором устройства, на которое этот файл ссылается.

st_size Размер файла в байтах. Он изменяется при записи в файл. Для специальных файлов этот размер всегда равен нулю.

st_atime Время последнего чтения файла. Для директории это время не изменяется при ее поиске (с помощью cd), но изменяется при просмотре содержания директории (с помощью ls), так как при этом читается файл директории. Это время изменяется следующими системными вызовами: creat, mknod, utime и read.

st_mtime Время последней записи в файл. Это время изменяется следующими системными вызовами: creat, mknod, utime и write.

st_ctime Время изменения состояния файла. Это время не изменяется при чтении и модификации содержимого файла. Это время изменяется следующими системными вызовами: chmod, chown, creat, link, mknod, pipe, unlink, utime и write.

Атрибуты файла - st_mode

Поле режима файла, st_mode, разделено на три набора битов: тип файла, специальные биты (setiud/setgid и «липкий» бит) и биты прав доступа. Для интерпретации st_mode используются символьные константы, определенные в <stat.h>

S_IFMT используется для выделения типа файла из поля режима файла. Примеры на следующей странице показывают, как использовать структуру stat и перечисленные символьные константы.

```
<sys/stat.h>
#define S_IFMT          0xF000  /* type of file */
#define S_IAMB          0x1FF   /* access mode bits */
#define S_IFIFO         0x1000  /* fifo */
#define S_IFCHR          0x2000  /* character special */
#define S_IFDIR         0x4000  /* directory */
/* XENIX definitions are not relevant to Solaris */
#define S_IFNAM         0x5000  /* XENIX special named file */
#define S_INSEM         0x1     /* XENIX semaphore subtype of IFNAM */
#define S_INSHD         0x2     /* XENIX shared data subtype of IFNAM */
#define S_IFBLK         0x6000  /* block special */
#define S_IFREG         0x8000  /* regular */
#define S_IFLNK         0xA000  /* symbolic link */
#define S_IFSOCK        0xC000  /* socket */
#define S_IFDOOR        0xD000  /* door */
#define S_IFPORT        0xE000  /* event port */
#define S_ISUID         0x800   /* set user id on execution */
#define S_ISGID         0x400   /* set group id on execution */
#define S_ISVTX         0x200   /* save swapped text even after use */
#define S_IREAD         00400   /* read permission, owner */
#define S_IWRITE        00200   /* write permission, owner */
#define S_IEXEC         00100   /* execute/search permission, owner */
#define S_ENFMT         S_ISGID /* record locking enforcement flag */
#define S_IRWXU         00700   /* read, write, execute: owner */
#define S_IRUSR         00400   /* read permission: owner */
#define S_IWUSR         00200   /* write permission: owner */
#define S_IXUSR         00100   /* execute permission: owner */
#define S_IRWXG         00070   /* read, write, execute: group */
#define S_IRGRP         00040   /* read permission: group */
#define S_IWGRP         00020   /* write permission: group */
#define S_IXGRP         00010   /* execute permission: group */
#define S_IRWXO         00007   /* read, write, execute: other */
#define S_IROTH         00004   /* read permission: other */

#define S_ISFIFO(mode)  (((mode)&0xF000) == 0x1000)
#define S_ISCHR(mode)   (((mode)&0xF000) == 0x2000)
#define S_ISDIR(mode)   (((mode)&0xF000) == 0x4000)
#define S_ISBLK(mode)   (((mode)&0xF000) == 0x6000)
#define S_ISREG(mode)   (((mode)&0xF000) == 0x8000)
#define S_ISLNK(mode)   (((mode)&0xF000) == 0xa000)
#define S_ISSOCK(mode)  (((mode)&0xF000) == 0xc000)
#define S_ISDOOR(mode)  (((mode)&0xF000) == 0xd000)

#define S_ISPORT(mode)  (((mode)&0xF000) == 0xe000)
```

Печать состояния файла - Пример

Эта программа печатает информацию о состоянии файла. Распечатываются все поля структуры stat, полученной с использованием stat(2). Программа работает следующим образом:

13 Объявление структуры stat.

15-18 Первый аргумент - имя файла. Второй аргумент — адрес структуры stat. Система заполняет структуру информацией о файле.

19-20 Печатаются имя файла и inode-номер.

21 Функция prntmode() печатает информацию из моды файла. Текст этой функции приведен дальше в этом разделе.

22-23 Печатаются число жестких связей и размер файла.

24-25 Эти функции печатают идентификатор пользователя и различные временные отметки файла. Тексты этих функций приведены дальше в этом разделе.

Эта программа демонстрируется на выполняемом файле и директории следующим образом:

```
$ stat stat # executable file
file name: stat         i-number: 6028
regular file    permissions: 755
links: 1        file size: 24899
user ID: 49026 name: jrs        group ID: 46014 group: ustg
last access:            Tue Dec 17 11:49:42 1989
last modification:      Tue Dec 17 10:19:45 1989
last status change:     Tue Dec 17 10:19:45 1989
$ stat /etc # directory
file name: /etc         i-number: 1258
directory       permissions: 775
links: 6        file size: 3024
user ID: 0 name: root   group ID: 3 group: sys
last access:            Tue Dec 17 02:30:16 1989
last modification:      Tue Dec 17 10:25:33 1989
last status change:     Tue Dec 17 10:25:33 1989
```

Файл: stat.c

ПЕЧАТЬ СОСТОЯНИЯ ФАЙЛА - ПРИМЕР

```
 1 #include <stdio.h>
 2 #include <sys/types.h>
 3 #include <stdlib.h>
 4 #include <sys/stat.h>
 5 static void prntuser(struct stat *);
 6 static void prntimes(struct stat *);
 7 static void prntmode(struct stat *);
 8
 9 /* print the status of a file */
10
11 main(int argc, char *argv[])
12 {
13     struct stat stbuf;
14
15     if (stat(argv[1], &stbuf) == -1) {
16         perror(argv[1]);
17         exit(1);
18     }
19     printf("file name: %s\t\t", argv[1]);
20     printf("i-number: %lu\n", stbuf.st_ino);
21     prntmode(&stbuf);
22     printf("links: %lu\t", stbuf.st_nlink);
23     printf("file size: %ld\n", stbuf.st_size);
24     prntuser(&stbuf);
25     prntimes(&stbuf);
26
27     exit(0);
28 }
29
```

Печать состояния файла - Пример (Продолжение)

Пример на следующей странице распечатывает права доступа файла. Он работает следующим образом:

35-53 Оператор switch печатает тип файла. Обратите внимание, как в строке 36 тип файла выделяется из моды файла. Оператор switch здесь уместен, так как файл может принадлежать только к одному типу.

39,43 Идентификатор специального устройства определен только для байт- и блок-ориетированных специальных файлов.

54-59 Если установлен любой из битов установки идентификатора пользователя, установки идентификатора группы или "липкий" бит, тогда печатается соответствующее сообщение. Обратите внимание, что одновременно могут быть установлены несколько битов.

60 Печатаются права доступа файла. Обратите внимание, как последние девять битов выделяются для печати.

Эта программа демонстрируется на байт- и блок-ориентированном специальном файле следующим образом:

```
$ stat /dev/tty38          # character special file
file name: /dev/tty38          i-number: 1722
character special file  special device: 5926    permissions: 622
links: 1        file size: 0
user ID: 49026 name: jrs        group ID: 46014 group: ustg
last access:            Tue Dec 17 11:54:57 1989
last modification:      Tue Dec 17 11:54:57 1989
last status change:     Tue Dec 17 11:54:57 1989
$ stat /dev/dsk/2s4 # block special file
file name: /dev/dsk/2s4         i-number: 146
block special file      special device: 36      permissions: 600
links: 1        file size: 0
user ID: 0 name: root   group ID: 0 group: root
last access:            Tue Dec 17 11:16:23 1989
last modification:      Tue Oct  9 10:43:45 1989
last status change:     Tue Oct  9 10:43:45 1989
```

Файл: stat.c

ПЕЧАТЬ СОСТОЯНИЯ ФАЙЛА - ПРИМЕР (ПРОДОЛЖЕНИЕ)

```
32
33 static void prntmode(struct stat *stbuf)
34 {
35     switch(stbuf->st_mode & S_IFMT) {
36     case S_IFDIR:
37         printf("directory\t");
38         break;
39     case S_IFCHR:
40         printf("character special file\t");
41         printf("special device: %lu\t", stbuf->st_rdev);
42         break;
43     case S_IFBLK:
44         printf("block special file\t");
45         printf("special device: %lu\t", stbuf->st_rdev);
46         break;
47     case S_IFREG:
48         printf("regular file\t");
49         break;
50     case S_IFIFO:
51         printf("named pipe\t");
52         break;
53     }
54     if (stbuf->st_mode & S_ISUID)
55         printf("setuid\t");
56     if (stbuf->st_mode & S_ISGID)
57         printf("setgid\t");
58     if (stbuf->st_mode & S_ISVTX)
59         printf("sticky\t");
60     printf("permissions: %o\n", stbuf->st_mode & 0777);
61 }
62
```

Доступ к БД учетных записей

В структуре stat, информация о пользователе и группе файла хранится в виде числовых идентификаторов uid и gid. Для распечатки информации о файле, удобно было бы перевести эту информацию в имена пользователя и группы. Для этого необходимо обратиться к базе данных учетных записей. В традиционных Unix-системах эта база хранилась в текстовых файлах /etc/passwd и /etc/group (в более современных также в файле /etc/shadow). Современные системы могут также использовать распределенные сетевые базы, такие, как NIS, NIS+ и LDAP. Если ваша программа самостоятельно анализирует файл /etc/passwd, она потребует адаптации для работы на системах, использующих LDAP. Поэтому рекомендуется использовать стандартные библиотечные функции, которые поддерживают все типы и форматы БД учетных записей, поддерживаемые текущей версией системы, и используют именно ту БД, из которой настроены брать информацию стандартные утилиты, такие, как login(1) и su(1).

Стандартные библиотечные функции getpwent(3C), getpwuid(3C) и getpwnam(3C) возвращают указатель на структуру, которая содержит разбитую на поля строку из файла /etc/password или другой БД учетных записей, в зависимости от конфигурации системы. Каждая строка в файле представлена в формате структуры password, определенной следующим образом:

```
struct passwd {
      char        *pw_name;
      char        *pw_passwd;
      uid_t   pw_uid;
      gid_t   pw_gid;
      char        *pw_age;
      char        *pw_comment;
      char        *pw_gecos;
      char        *pw_dir;
      char        *pw_shell;
};
```

Замечание: Поле pw_comment не используется. Другие поля имеют значения, соответствующие описанным в password(4). При первом вызове getpwent(3C) возвращает указатель на первую структуру password в файле. В следующий раз getpwent(3C) вернет указатель на следующую структуру password. Последовательные вызовы могут использоваться для поиска во всем файле.

getpwuid(3C) ищет от начала файла, пока не найдет структуру с полем идентификатора пользователя равным uid. Возвращает указатель на найденную структуру.

getpwnam(3C) ищет от начала файла, пока не найдет структуру с полем регистрационного имени пользователя равным name. Возвращает указатель на найденную структуру.

Вызов setpwent(3C) предоставляет возможность вести последующий поиск с помощью функции getpwent(3C) с начала файла.

endpwent(3C) может быть вызвана, чтобы закрыть файл паролей после завершения обработки.

fgetpwent(3C) возвращает указатель на следующую структуру pasword в потоке f, формат которого соответствует формату /etc/password.

Замечание: эти библиотечные функции возвращают указатель на структуру, которая расположена в сегменте данных. Следовательно, значения в структуре должны быть скопированы перед последующими вызовами этих функций. Если достигнут конец файла или возникнет ошибка чтения, то функция вернет NULL.

В традиционных Unix-системах в поле pw_passwd хранился хэш пароля. В SVR4

информация о паролях не хранится больше в /etc/password. Информацию о паролях содержит файл /etc/shadow, который доступен для чтения только root. Это защищает от словарной атаки, позволяющей определить пароли пользователей путем подбора значений с совпадающей хэш-функцией.

Получение доступа к файлу групп

Функции getgrent(3C), getgrgid(3C) и getgrnam(3C) возвращают указатель на структуру, которая содержит разбитую на поля строку из файла /etc/group. Каждая строка представлена в формате структуры group, определенной следующим образом:

```
grp.h:
struct group {
        char    *gr_name;
        char    *gr_passwd;
        gid_t   gr_gid;
        char    **gr_mem;
};
```

При первом вызове getgrent(3C) возвращает указатель на первую структуру group в файле. В следующий раз getgrent(3C) вернет указатель на следующую структуру group. Последовательные вызовы могут использоваться для поиска во всем файле.

getgrgid(3C) ищет от начала файла, пока не найдет структуру с полем идентификатора группы равным gid. Возвращает указатель на найденную структуру.

getpgram(3C) ищет от начала файла, пока не найдет структуру с полем имени группы равным name. Возвращает указатель на найденную структуру.

Вызов setgrent(3C) предоставляет возможность вести последующий поиск с помощью функции getgrent(3C) с начала файла.

endgrent(3C) может быть вызвана, чтобы закрыть файл групп после завершения обработки.

fgetgrent(3C) возвращает указатель на следующую структуру group в потоке f, формат которого соответствует формату /etc/group.

Замечание: эти библиотечные функции возвращают указатель на структуру, которая расположена в сегменте данных. Следовательно, значения в структуре должны быть скопированы перед последующими вызовами этих функций. Если достигнут конец файла или возникнет ошибка чтения, то функция вернет NULL.

Печать имени пользователя - Пример

Конец файла stat.c приведен на следующей странице. Функция prntuser() выводит на печать идентификаторы пользователя и группы, вместе с соответствующими именами пользователя и группы. Функция prntimes() печатает временные отметки файла в понятном формате.

Эти функции работают следующим образом:

71-72 Объявляются указатели на структуры password и group.

74-76 Печатается имя пользователя, полученное по идентификатору пользователя.

77-79 Печатается имя группы, полученное по идентификатору группы.

88-93 Печатаются три временные характеристики файла в понятном формате.

Файл: stat.c

ПЕЧАТЬ ИМЕНИ ПОЛЬЗОВАТЕЛЯ - ПРИМЕР

```
64 #include <pwd.h>
65 #include <grp.h>
66
67 /* print user and group name */
68
69 static void prntuser(struct stat *stbuf)
70 {
71     struct passwd *pw;
72     struct group *grp;
73
74     pw = getpwuid(stbuf->st_uid);
75     printf("user ID: %ld name: %s\t",
76         stbuf->st_uid, pw->pw_name);
77     grp = getgrgid(stbuf->st_gid);
78     printf("group ID: %ld group: %s\n",
79         stbuf->st_gid, grp->gr_name);
80 }
81
82 #include <time.h>
83
84 /* print the three time stamps */
85
86 static void prntimes(struct stat *stbuf)
87 {
88     printf("last access: \t\t%s",
89         ctime(&stbuf->st_atime));
90     printf("last modification: \t%s",
91         ctime(&stbuf->st_mtime));
92     printf("last status change: \t%s",
93         ctime(&stbuf->st_ctime));
94 }
```

Изменение прав доступа файла

Системный вызов chmod(2) может изменить специальные биты и биты прав доступа файла. Этот системный вызов также используется одноименной командой chmod(1).

Аргументы chmod(2):

path Путевое имя файла или устройства. Используется в системном вызове chmod(2).

fildes Дескриптор открытого файла. Используется в системном вызове fchmod(2).

mode Этот двенадцатибитовый аргумент используется для изменения специальных битов и битов прав доступа. В отличие от open(2), параметр chmod(2) не подвергается преобразованию в соответствии с cmask.

Чтобы изменить права доступа файла, эффективный пользовательский идентификатор процесса, вызвавшего chmod(2), должен совпадать с пользовательским идентификатором владельца файла. Также, пользователь root (в Solaris также обладатель привилегии PRIV_FILE_OWNER, см privileges(5)) может менять права доступа всех файлов.

Воздействие chmod(2) можно наблюдать, если вызвать stat(2) с данным файлом и проверить поле st_mode структуры stat.

Кроме традиционной маски прав доступа, многие современные Unix-системы поддерживают списки управления доступом произвольного вида (discretionary ACL). Такой ACL состоит из последовательности записей, каждая из которых содержит идентификатор, тип (флаг, указывающий, содержит ли запись идентификатор пользователя или группы) и биты прав на чтение, запись и исполнение. Отдельного права изменять права не предусмотрено, для изменения ACL необходимо, чтобы эффективный идентификатор процесса совпадал с идентификатором хозяина файла или имел привилегию менять права доступа для произвольных файлов. Для просмотра и модификации ACL файла следует использовать системные вызовы acl(2) и facl(2) или команды setfacl(1) и getfacl(1). Эти вызовы в нашем курсе подробно не рассматриваются. Поддержка ACL требуется как на уровне ядра ОС, так и на уровне файловой системы; во многих случаях, поддержка ACL может включаться или выключаться при монтировании файловой системы.

Изменение прав доступа файла - Пример

Эта программа демонстрирует, как использовать системный вызов chmod(2). Это упрощенная версия команды chmod(1). Она работает следующим образом:

10 Новые права доступа получены из первого аргумента и преобразуются к восьмеричному виду.

12-15 Изменяются права доступа файла, заданного вторым аргументом.

Эта программа демонстрируется следующим образом:

```
$ >file
$ ls -l file
-rw-r--r--   1 wjj     ustg          0 Dec 17 13:47 file
$ setmode 4755 file
$ ls -l file
-rwsr-xr-x   1 wjj     ustg          0 Dec 17 13:47 file
```

Файл: setmode.c

ИЗМЕНЕНИЕ ПРАВ ДОСТУПА ФАЙЛА - ПРИМЕР

```
 1 #include <sys/types.h>
 2 #include <sys/stat.h>
 3 #include <stdio.h>
 4 #include <stdlib.h>
 5
 6 main(int argc, char *argv[])
 7 {
 8     mode_t newmode;
 9
10     newmode = strtol(argv[1], (char **) NULL, 8);
11
12     if (chmod(argv[2], newmode) == -1) {
13                 perror(argv[2]);
14         exit(1);
15     }
16
17     exit(0);
18 }
```

Изменение владельца и группы файла

Системный вызов chown(2) изменяет владельца файла. lchown(2) изменяет владельца файла символической связи. fchown(2) изменяет владельца открытого файла, используя дескриптор файла. Идентификаторы пользователя и группы могут быть изменены одновременно. Команды chown(1) и chgrp(1) реализованы с использованием этого системного вызова.

Большинство современных Unix-систем ограничивают возможность изменения владельца файла, потому что владение пользователя файлом не только кодирует права доступа, но и означает, что файл учитывается дисковой квотой этого пользователя. В Solaris, это ограничение регулируется директивой set rstchown в файле /etc/system либо может устанавливаться параметрами монтирования файловой системы. Определить, действуют ли ограничения на chown в заданной файловой системе можно вызовом pathconf(2) с параметром _PC_CHOWN_RESTRICTED.

Если chown ограничен, менять хозяина файла могут обладатели привилегий PRIV_FILE_CHOWN (по умолчанию это пользователь root) или PRIV_FILE_CHOWN_SELF (privileges(5)). Если chown не ограничен, владелец файла (точнее, процесс с эффективным идентификатором равным идентификатору владельца) также может изменять идентификатор владельца файла. Изменение параметра rstchown требует перезагрузки системы.

Для изменения идентификатора группы достаточно быть владельцем файла.

Аргументы системного вызова:

path Путевое имя файла, используется chown(2) и lchown(2).

fildes Дескриптор открытого файла, используется fchown(2).

owner Новый пользовательский идентификатор файла.

group Новый идентификатор группы файла.

Если owner или group равно -1, соответствующий идентификатор не изменяется.

Воздействие chown(2) можно наблюдать, если вызвать stat(2) с данным файлом и проверить поле st_gid структуры stat.

Установка времени доступа и изменения файла

Системный вызов utime(2) используется для изменения времени последнего доступа и последнего изменения файла. Это может быть полезно при копировании файлов или их распаковке из архива, чтобы время модификации копии соответствовало времени модификации оригинала, а не моменту копирования.

Аргументы utime(2):

path Путевое имя файла

times Адрес структуры utimbuf, содержащей новые временные отметки.

```
struct utimbuf {
        time_t actime;          /* access time */
        time_t modtime;         /* modification time */
};
```

Если times равно нулю, время доступа и изменения файла устанавливаются равными текущему времени. Чтобы использовать utime(2) таким образом, процесс должен иметь эффективный идентификатор пользователя равный владельцу данного файла или иметь право на запись в файл.

Если times не равно нулю, оно интерпретируется как указатель на struct itimbuf, и времена доступа и изменения устанавливаются равными значениям, содержащимися в структуре. Только владелец файла или суперпользователь может так использовать utime(2).

В обоих случаях, время создания файла устанавливается равным текущему времени.

Команда touch(1) использует utime(2).

Изменение временных отметок файла - Пример

Эта программа демонстрирует, как работает системный вызов utime(2). Она устанавливает временные отметки вашего файла равными соответствующим временным отметкам другого файла или равными текущему времени. Этот пример является упрощенной версией команды touch(1).

Эта программа работает следующим образом:

16-23 Если первый аргумент - путевое имя файла, то получаем его времена последнего доступа и изменения. Отсутствие первого аргумента указывается с помощью "-".

24-25 Если первый аргумент равен "-", указатель на times устанавливаем равным NULL, для задания текущего времени.

26-29 Изменяем временные характеристики второго файла.

Эта программа демонстрируется следующим образом:

```
$ ls -ld /uxm2/tmm
drw-r--r--   4 tmm     ustg         240 Dec 15 10:38 /uxm2/tmm
$ date
Wed Dec 17 18:17:15 EST 1986
$ ls -l file
-rw-r--r--   1 tmm     ustg           0 Dec 11 15:35 file
$ settime - file
$ ls -l file
-rw-r--r--   1 tmm     ustg           0 Dec 17 18:18 file
$ settime /uxm2/tmm file
$ ls -l file
-rw-r--r--   1 tmm     ustg           0 Dec 15 10:38 file
```

Файл: settime.c

ИЗМЕНЕНИЕ ВРЕМЕННЫХ ОТМЕТОК ФАЙЛА - ПРИМЕР

```
 1 #include <stdio.h>
 2 #include <sys/types.h>
 3 #include <stdlib.h>
 4 #include <sys/stat.h>
 5 #include <utime.h>
 6
 7 main(int argc, char *argv[])
 8 {
 9     struct stat stbuf;
10     struct utimbuf timestamp, *times = &timestamp;
11
12     if (argc < 3) {
13         printf("usage: \t%s otherfile yourfile\n", arg
14         exit(1);
15     }
16     if (argv[1][0] != '-') {
17         if (stat(argv[1], &stbuf) == -1) {
18             perror(argv[1]);
19             exit(2);
20         }
21         times->actime = stbuf.st_atime;
22         times->modtime = stbuf.st_mtime;
23     }
24     else
25         times = NULL;
26     if (utime(argv[2], times) == -1) {
27         perror(argv[2]);
28         exit(3);
29     }
30     exit(0);
31 }
```

Установка длины файла

Функция truncate(3C) используется для установки заданной длины у файла. truncate(3C) требует, чтобы пользователь с эффективным пользовательским идентификатором процесса имел право на запись в данный файл. ftruncate(3C) требует, чтобы файл был открыт на запись.

Аргументы truncate(3C) и ftruncate(3C):

path Путевое имя файла.

fildes Дескриптор файла, открытого на запись.

length Если файл был длиннее чем length, байты за length станут недоступны и дисковое пространство будет освобождено. Если файл был короче чем length, размер файла будет установлен равным length. В этом случае, байты между старым и новым концами файла будут считываться как нули.

Поиск файла

Библиотечная функция pathfind(3G) используется для поиска файла в списке директорий. Аргументы pathfind(3G):

path Список разделенных двоеточием (:) директорий, в которых ведется поиск. Пустой path трактуется, как текущая директория.

name Имя файла, который ищется в заданном списке директорий.

mode Задает некоторые характеристики искомого файла. Это строка букв опций из набора rwxfbcdpugks, где буквы задают, соответственно, доступность по чтению, доступность по изменению, доступность по исполнению, нормальный файл, специальный блок-ориентированный, специальный байт-ориентированный, директорию, FIFO, установку бита идентификатора пользователя, установку бита идентификатора группы, "липкий бит" и ненулевой размер. Опции чтения, изменения и исполнения проверяются для реального идентификатора пользователя и группы. Если mode - пустая строка, не задается никаких характеристик для поиска файла.

Если файл name со всеми характеристиками, заданными с помощью mode, найден в какой-либо из директорий, перечисленных в path, тогда pathfind(3G) возвращает указатель на строку, содержащую имя директории из path, за которым идет косая черта(/), за которой идет name.

Замечание: Программы, использующие библиотечные функции из секции (3G), должны быть скомпилированы с опцией -lgen, чтобы компилятор подключил библиотеку libgen.a.

Поиск файла - Пример

На следующей странице приведен пример использования pathfind(3G). Пример работает следующим образом:

9 Массивы объявляются как static, так чтобы они инициализировались нулями.

17-18 Пользователь пытается ввести mode, которая задает характеристики отыскиваемого файла.

20-21 Пользователь пытается ввести имена директорий для поиска. Точка (.) интерпретируется как текущая директория. Двоеточие (:) также интерпретируется как текущая директория, потому что ее ввод добавляет пустой элемент в список директорий.

23-24 Директории читаются в массив temp.

25-26 Производится проверка, что размер массива, отведенного под хранение списка директорий, не превысил предельного значения.

29 Новая директория вставляется в конец списка директорий поиска.

30 В конец списка директорий добавляется двоеточие (:), приготавливая вставку следующей директории.

32 Последнее двоеточие стирается из списка директорий.

33-36 Осуществляется поиск файла, заданного в командной строке, в списке директорий, введенных пользователем. Если файл найден в одной из директорий, то печатается имя этой директории.

Программа демонстрируется следующим образом:

```
$ pathfind ls
Enter mode (CTRL<D> for no mode): rx

Enter directories to be searched
End input with <CTRL/D>
/etc
/bin
/usr/bin
.
/instr/jrs/bin

ls is found in /bin/ls
```

Файл: pathfind.c

ПОИСК ФАЙЛА - ПРИМЕР

```
 1 #include <stdlib.h>
 2 #include <stdio.h>
 3 #include <libgen.h>
 4 #include <string.h>
 5 #define MAXLEN 256
 6
 7 main(int argc, char *argv[])
 8 {
 9     static char temp[50], dir[MAXLEN], mode[10];
10     char *ptr;
...
17     printf("Enter mode (<CRTL/D> for no mode): ");
18     scanf("%s", mode);
19
20     printf("\nEnter directories to be searched\n");
21     printf("End input with <CTRL/D>\n");
22     for(;;) {
23     if (scanf("%s",temp) == -1)
24         break;
25     if (strlen(dir) + strlen(temp) > MAXLEN) {
26         printf("last input not allowed\n");
27         break;
28     }
29     strcat(dir,temp);
30     dir[strlen(dir)] = ':';
31     }
32     dir[strlen(dir) - 1] = '\0';
33     if ((ptr = pathfind(dir,argv[1],mode)) == NULL)
34     printf("\n%s not found\n",argv[1]);
35     else
36     printf("\n%s found in %s\n",argv[1],ptr);
37     exit(0);
38 }
```

Генерация имени для временного файла

Библиотечная функция mktemp(3C) используется для получения уникального имени, которое обычно используется для временных файлов. Аргументом mktemp(3C) является шаблон имени файла, заканчивающийся шестью буквами X, то есть подстрокой XXXXXX. mktemp(3C) заменяет XXXXXX на уникальные символы. При генерации этих символов используются pid процесса и внутренний счетчик для каждого процесса, так что последовательный вызов mktemp(3C) в одном процессе будет порождать разные имена.

mktemp(3C) не проверяет уникальности сгенерированного имени файла. Файл рекомендуется создавать с комбинацией флагов O_CREATE | O_EXCL, и при обнаружении, что такой файл уже существует, повторно вызывать mktemp(3C).

Библиотечные функции tmpfile(3S) и tmpname(3S) также используются для создания имени временного файла.

Создание временного файла - Пример

Пример, приведенный на следующей странице, показывает как использовать mktemp(3C) для создания имени временного файла. Пример работает следующим образом:

9 Объявляется массив tempfile. Он будет содержать имя временного файла.

12-17 Имя временного файла формируется из первых восьми символов имени программы, за которыми следуют подставляемые вместо XXXXXX функцией mktemp(3C) уникальные символы.

19 Печатается имя временного файла.

21-25 Временный файл создается и открывается на чтение и запись.

27 Чтение и запись временного файла опущены.

29 Временный файл уничтожается.

Программа демонстрируется следующим образом:

```
$ mktemp
temporary file name is: mktempa001SL
$
```

Файл: mktemp.c

СОЗДАНИЕ ВРЕМЕННОГО ФАЙЛА - ПРИМЕР

```
 1 #include <stdlib.h>
 2 #include <stdio.h>
 3 #include <string.h>
 4 #include <unistd.h>
 5 #include <fcntl.h>
 6
 7 main(int argc, char *argv[])
 8 {
 9     static char tempfile[15];
10     int fd;
11
12     strncpy(tempfile, argv[0], 8);
13     strcat(tempfile, "XXXXXX");
14     if (mktemp(tempfile) == NULL) {
15     printf("cannot create temporary file name\n");
16     exit(1);
17     }
18
19     printf("temporary file name is: %s\n",tempfile);
20
21     if ((fd = open(tempfile, O_RDWR | O_CREAT,
22     0640)) == -1) {
23     perror(argv[0]);
24     exit(0);
25     }
26
27     /* use temporary file */
28
29     unlink(tempfile);
30     exit(0);
31 }
```

7. УПРАВЛЕНИЕ ДИРЕКТОРИЯМИ

Обзор

В предыдущем разделе вы научились получать доступ к управляющей информации и информации о состоянии файла, хранящейся в его иноде, и изменять эту информацию. В этом разделе вы научитесь выполнять действия над директориями, а также узнаете, как организованы директории и файлы.

В частности, вы научитесь просматривать содержимое директории. Затем вы изучите системные вызовы для добавления и удаления записей о файле. Кроме того, вы научитесь переходить из одной текущей рабочей директории в другую.

Свойства директории

Директории (каталоги) используются для иерархической организации других файлов, включая и другие директории, а также для организации файловых систем. При монтировании файловой системы необходимо указать существующий каталог, так что после монтирования содержимое файловой системы выглядит как подкаталог или иерархия подкаталогов. Пользовательский процесс может открывать директорию только для чтения. Модификация директории осуществляется неявно, системными вызовами link(2), unlink(2), open(2) и mknod(2).

Формат записи в директории выглядит так:

```
/*
* File-system independent directory entry.
*/
struct dirent {
	ino_t		d_ino;  /* "inode number" of entry */
	off_t		d_off;  /* offset of disk directory entry */
	unsigned short  d_reclen;	/* length of this record */
	char		d_name[1];	/* name of file */
};
```

Первое поле, d_ino, представляет собой номер инода файла. Этот номер определяет инод, который содержит всю управляющую информацию о файле и информацию о его состоянии. Поле d_name[] содержит имя файла. Максимальное количество символов, которое может содержаться в имени файла, зависит от типа используемой файловой системы и может быть определено системным вызовом pathconf(2) с параметром _PC_NAME_MAX. Поскольку каталог может быть точкой монтирования, максимальную длину имени следует определять для каждого каталога заново.

Обратите внимание, что длина имени файла не учитывается в размере структуры dirent, поэтому при размещении памяти под эту структуру недостаточно выделить sizeof(struct dirent) байт. При размещении памяти для копии уже считанной записи директории можно определять объем требуемой памяти как sizeof(struct dirent)+strlen(d.d_name).

Заметьте, что имя файла не содержится в иноде. Это дает дополнительную гибкость, позволяя иметь несколько записей об одном файле в одной или в различных директориях; имена в различных директориях также могут быть различными. Запись в директории называется жесткой связью.

Директории могут только увеличиваться в размере, так как место, занятое стертой записью, остается в директории. Поэтому могут существовать большие директории с маленьким числом записей. Новые записи используют пространство из-под стертых записей, когда это возможно.

Свойства директории

Права доступа для директорий ведут себя не так, как для файлов:

на чтение: Разрешает считывать содержимое директории. Например, вы можете использовать команду ls(1) без опций для просмотра директории.

на изменение: Позволяет добавлять и удалять файлы в директории. Если на директории установлен «липкий» бит, для удаления файла необходимо быть владельцем файла; иначе, для удаления файла достаточно иметь право записи в каталог.

на исполнение: Позволяет осуществлять поиск в директории, то есть право открывать файлы в этом каталоге и его подкаталогах, а также право проверять существование и доступность таких файлов. Право на исполнение требуется, если вы хотите установить директорию как текущую. Кроме того все компоненты-директории из путевого имени должны быть исполняемыми, хотя и не обязаны иметь право на чтение, когда вы читаете файл или исполняете команду. Требование доступа на исполнение для компонентов-директорий путевого имени распространяется на все системные вызовы, которые получают путевое имя в качестве параметра.

Хотя вы можете не иметь право на чтение из директории, если вы имеете право на исполнение для нее и знаете имена файлов и поддиректорий в этой директории, то вы можете читать или исполнять файлы из нее, в зависимости от прав доступа к файлам.

ls -ld распечатывает информацию о директории (по умолчанию - о текущей директории). Эта команда не выдает информацию о файлах в директории. Такую информацию выдает команда ls -l.

Директории и файлы

Иллюстрация на следующей странице показывает взаимоотношения между записью в директории, инодом файла и блоками данных. Уникальный номер файла, например, 36 указывает на инод номер 36 в таблице инодов

Таблица инодов — это статическая или динамическая таблица на диске, в которой собраны все иноды файловой системы. Формат этой таблицы и формат самого инода зависит от файловой системы; при монтировании файловой системы, ядро инициализирует соответствующий модуль (драйвер файловой системы), который преобразует формат метаданных, хранящихся на диске, во внутренние форматы ядра ОС. В случае, если файловая система не использует иноды, как FAT 16/32 или CDFS, драйвер файловой системы должен сочинять структуры инода «на ходу».

В inode хранятся списки указателей на блоки данных файла. Формат этих списков также зависит от файловой системы. Некоторые ФС хранят отдельный указатель на каждый блок. Для длинных файлов используются косвенные блоки, то есть блоки, хранящие указатели на другие блоки. Некоторые другие ФС хранят списки так называемых экстентов, то есть фрагментов файла, каждый из которых занимает непрерывное место на диске. Каждый экстент описывается своим логическим смещением от начала файла, указателем на первый блок на диске и длиной.

Кроме дисковых файловых систем, Unix поддерживает сетевые файловые системы, а также псевдофайловые системы. Аналогично псевдоустройствам, псевдофайловые системы обслуживаются специальными драйверами, которые во всех наблюдаемых отношениях ведут себя как драйверы обычных ФС, но вместо хранения файлов и каталогов на диске, генерируют содержимое этих файлов и каталогов программно. Примером псевдофайловой системы является /proc, в которой ядро системы создает по каталогу для каждого процесса в системе. Этот каталог, в свою очередь, содержит образ адресного пространства процесса, ссылки на открытые этим процессом файлы и другую информацию.

Изменение текущей директории

Системные вызовы chdir(2) и fchdir(2) изменяют текущую рабочую директорию вашего процесса. Текущая рабочая директория является частью среды исполнения вашего процесса. Текущая директория неявно используется большинством системных вызовов, которые допускают передачу относительного путевого имени: open(2), stat(2), chmod(2) и т. д. Относительное путевое имя — это имя, которое начинается с любого символа, кроме /. Система ищет файл с таким именем в поддиректориях текущей директории. На самом деле, каждая директория в Unix содержит запись .., которая указывает на родительскую директорию, поэтому относительные имена можно использовать для поиска не только в поддиректориях.

Параметр chdir(2):

path Путевое имя файла-директории, который должен стать новой текущей рабочей директорией.

Параметр fchdir(2):

fildes Файловый дескриптор открытого файла. Дескриптор возвращается системным вызовом open(2) и является одним из полей структуры DIR, используемой вызовом opendir(3C).

Чтобы назначить директорию текущей, процесс должен иметь право записи в эту директорию и все её родительские директории.

Оператор shell cd использует chdir(2). Поскольку текущая директория — это атрибут процесса, команда cd(1) не может быть внешней командой, запускаемой в подпроцессе, как ls(1), mv(1) или cp(1). Это должна быть именно встроенная команда shell, исполняющаяся в том же процессе.

Создание директории

Вызов mkdir(2) создает новую директорию. Параметры mkdir(2):

path Путевое имя файла-директории, которая должна быть создана.

mode Используется для установления прав доступа для директории. Изменяется значением cmask данного процесса

Кроме mkdir(2), каталоги также можно создавать системным вызовом mknod(2).

Команда mkdir(1) вызывает mkdir(2).

Удаление директории

Вызов rmdir(2) удаляет директорию. Параметр rmdir(2) path представляет собой путевое имя файла-директории, которая должна быть удалена.

Удаляемая директория не должна быть текущей ни у одного процесса, она не должна содержать записей, кроме . и .., и ни один процесс не должен иметь на эту директорию открытого дескриптора файла. Кроме того, вы должны иметь соответствующие права доступа (см. страницу руководства rmdir(2)).

Создание и удаление директории - Пример

Этот пример создает директорию и затем удаляет ее. Проверка создания и удаления выполняется вызовом system(3C) для исполнения команды ls(1).

Каталог /tmp присутствует во всех Unix-системах и, как ясно из названия, предназначен для хранения промежуточных файлов. При обычных настройках системы, этот каталог доступен для записи всем (чтобы пользователи не могли удалять чужие файлы, обычно на этом каталоге устанавливается «липкий» бит). Также, обычно, этот каталог очищается при каждой перезагрузке системы. Во многих системах, длительно работающих без перезагрузки, администратор планирует исполняющийся по расписанию скрипт, который удаляет из /tmp файлы, к которым долго не было обращений.

В Solaris, каталог /tmp обычно размещается на файловой системе tmpfs, которая размещает файлы не на диске, а в виртуальной памяти.

Эта программа работает так:

```
$ mkrmdir
drwxr-x---   2 jeg      unixc         32 Mar  4 10:50 /tmp/D
/tmp/D not found
```

Файл: mkrmdir.c

СОЗДАНИЕ И УДАЛЕНИЕ ДИРЕКТОРИИ - ПРИМЕР

```
 1 #include <stdlib.h>
 2 #include <unistd.h>
 3 #include <sys/types.h>
 4 #include <sys/stat.h>
 5
 6 main()
 7 {
 8     mkdir("/tmp/D", S_IREAD|S_IWRITE|
 9         S_IEXEC|S_IRGRP|S_IXGRP);
10
11     system("ls -ld /tmp/D");
12
13     rmdir("/tmp/D");
14
15     system("ls -ld /tmp/D");
16 }
```

Создание/удаление цепочки директорий

mkdirp(3G) создает все отсутствующие директории в заданном пути path (путевом имени) с правами доступа, заданными параметром mode.

mkdirp(3G) возвращает 0, если все требуемые директории успешно созданы или уже существовали.

rmdirp(3G) удаляет директории, содержащиеся в путевом имени path. Удаление начинается с конца пути и движется к корню, пока это возможно. Если возникает ошибка (например, очередная удаляемая директория содержит какие-то файлы или другие поддиректории, или у вас нет прав), остаток пути сохраняется в path.

rmdirp(3G) возвращает 0, если он смог удалить все директории в пути.

rmdirp(3G) может удалять только пустые директории.

rmdirp(2) возвращает -2, если путь содержит '.' или '..', и -3, если делается попытка удалить текущую директорию. Если возникла какая-то другая ошибка, возвращается -1.

Создание/удаление цепочки директорий - Пример

Эта программа использует mkdirp(3G) и rmdirp(3G) для создания и удаления цепочки директорий, соответственно. Она работает так:

8 Это путевое имя директории, компоненты которой должны быть созданы.

11 Если директории junk1 и dir1 не существуют, они создаются.

16 Сменить текущую директорию на /tmp/junk1/dir1

22-26 После использования этих директорий, они удаляются

Файл: mkrmdirp.c

СОЗДАНИЕ/УДАЛЕНИЕ ЦЕПОЧКИ ДИРЕКТОРИЙ - ПРИМЕР

```
 1 #include <unistd.h>
 2 #include <stdlib.h>
 3 #include <stdio.h>
 4 #include <libgen.h>
 5
 6 main(int argc, char *argv[])
 7 {
 8     char *path = "/tmp/junk1/dir1";
 9     char buf[50];
10
11     if (mkdirp(path, 0750) == -1)  {
12     perror(argv[0]);
13     exit(1);
14     }
15
16     chdir(path);
17     system("pwd");
18
19     /* use directory */
20
21     chdir("/tmp");
22     if (rmdirp("junk1/dir1", buf) != 0)  {
23     printf("cannot remove all directories\n");
24     printf("remaining directories %s\n",buf);
25     exit(1);
26     }
27 }
```

Чтение записей директории

Библиотечные функции, перечисленные на странице руководства directory(3C), используются для чтения записей директории.

Функция opendir(3C) открывает директорию. Она возвращает указатель на структуру DIR, которая используется в качестве параметра к readdir(3C), closedir(3C) и rewinddir(3C).

Функция readdir(3C) используется для чтения записей из директории. Она возвращает указатель на структуру dirent, которая содержит следующую непустую запись в директории, заданной с помощью dirp. Вам не нужно выделять или освобождать память из-под записи dirent; readdir(3C) возвращает указатель на внутренний буфер, который может быть переиспользован следующим вызовом readdir(3C). Это освобождает вас от необходимости определять максимальную длину имени файла в данном каталоге. Однако, если вам необходимо сохранять копии структур dirent, вам необходимо выделять память под копии с учетом реальной длины имени файла в каждой записи.

В файле <limits.h> определен препроцессорный символ NAME_MAX, но не следует полагаться на значение этого символа; в следующих версиях Solaris соответствующее ограничение может быть повышено и ваша программа потребует перекомпиляции. Чтобы избежать этого, рекомендуется выделять память с учетом реальной длины имени файла или значения, возвращаемого pathconf(2).

Функция closedir(3C) закрывает дескриптор директории, заданный параметром dirp, и освобождает связанную с ним структуру. Это может привести к освобождению памяти, используемой для размещения структур dirent, поэтому если вам необходимо сохранить какие-то из этих структур после закрытия дескриптора, их необходимо скопировать.

Функция rewinddir(3C) перемещает позицию считывания к началу директории.

telldir(2C) возвращает текущую позицию.

seekdir(3C) устанавливает эту позицию для последующей операции readdir(3C).

Нестандартный системный вызов getdents(2) в Solaris был сделан специально для реализации библиотечной функции readdir(3C). Справочное руководство программиста ОС UNIX System V предлагает использовать для чтения записей readdir(3C), поскольку эта функция входит совместима с другими системами.

Связь с файлом

Системный вызов link(2) создает жесткую связь для файла, создавая для него новую запись в директории. Связь — это другое имя для обращения к данным в этом же файле. Предоставляется также команда ln(1), которая вызывает link(2).

Обычные пользователи могут создавать связи только для обычных файлов. Супервизор может создавать связи для блочных и символьных специальных файлов. Создание жестких связей для каталогов запрещено на уровне драйвера файловой системы.

Параметры системного вызова link таковы:

path1 Путевое имя существующего файла, для которого нужно создать новую связь.

path2 Путевое имя новой связи.

path2 может указывать на ту же директорию, что и path1, или на другую.

Как path1, так и path2 могут быть абсолютными или относительными именами файлов.

Нельзя создать связь из файла в одной файловой системе в другой файловой системе. Это связано с тем, что жесткая связь представляет собой номер инода файла, но номера инодов уникальны только в пределах файловой системы.

Множественные связи

Иллюстрация на следующей странице показывает файл, для которого существуют записи в двух директориях. ОС UNIX не делает различия между этими двумя именами. Обе записи имеют один и тот же номер инода и ссылаются на одни и те же метаданные в иноде и блоки данных.

Поскольку права доступа и временные штампы хранятся в иноде, обе связи будут иметь одни и те же права доступа и одни и те же времена создания, доступа и модификации.

Если файл name1 уже существует, то команда для создания записи name2 в директорию directory2 выглядит так:

```
$ ln directory1/name1 directory2/name2
```

Создание связи с файлом - пример

Этот пример является упрощенной версией команды ln(1). Программа работает так:

7-10 Системный вызов link(2) создает связь для файла, заданного в качестве первого аргумента. Связь может находиться в той же или в другой директории.

Глядя на иллюстрацию на предыдущей странице, представьте, что ваша текущая директория - это общая родительская директория для directory1 и directory2, и файл с именем name1 в уже существует в directory1. Затем, в directory2 создается новая связь:

```
$ link directory1/name1 directory2/name2
```

Эта программа демонстрируется так:

```
$ ls -l -i
total 9
11621 -rw-r--r--   1 imr     ustg         96 Jan  3 17:45 data
25799 -rwxr-xr-x   2 imr     ustg       3350 Jan  3 17:40 link
$ link data data2
$ ls -l -i
total 10
11621 -rw-r--r--   2 imr     ustg         96 Jan  3 17:45 data
11621 -rw-r--r--   2 imr     ustg         96 Jan  3 17:45 data2
25799 -rwxr-xr-x   2 imr     ustg       3350 Jan  3 17:40 link
```

Первая команда ls(1) выдает файлы в текущей директории вместе с их номерами инодов. Затем программа-пример используется для создания новой связи для файла data в той же директории. Вторая команда ls(1) показывает, что data и data2 ссылаются на один и тот же файл, поскольку номера инодов одинаковы.

Файл: link.c

СОЗДАНИЕ СВЯЗИ С ФАЙЛОМ - ПРИМЕР

```
 1 #include <stdio.h>
 2 #include <unistd.h>
 3 #include <stdlib.h>
 4
 5 main(int argc, char *argv[])
 6 {
 7     if (link(argv[1], argv[2]) == -1) {
 8         perror(argv[0]);
 9         exit(1);
10     }
11     exit(0);
12 }
```

Создание символической связи с файлом

Использование жестких связей сопряжено со значительными ограничениями. Жесткие связи могут создаваться только в пределах одной файловой системы и не могут создаваться на каталоги. Символические связи лишены этих ограничений.

Системный вызов symlink(2) создает символическую связь с именем name2 для файла name1. Оба имени могут быть произвольными путевыми именами, файлы не обязаны принадлежать к одной файловой системе, а файл name1 может вообще не существовать.

Символическая связь

Эта иллюстрация показывает файл с символической связью. name2 — это символическая связь с name1. Заметьте, что символическая связь (name2) — это инод специального типа и запись в директории с номером инода,, отличающимся от инода файла, на который указывает связь. Инод символической связи, вместо ссылок на блоки данных, содержит строку: имя файла, на который она указывает. Это может быть как абсолютное, так и относительное путевое имя. При использовании относительного путевого имени, оно отсчитвается не от текущей директории процесса, а от родительской директории файла-связи.

Команда shell для создания такой связи выглядит так:

```
$ ln -s /dir_path/directory1/name1 name2
$ ls -lgo name2
lrwxrwxrwx  1   43 May 26 13:36 name2 -> /dir_path/directory1/name1
```

Команда ln -s создает файл символической связи с именем, равным второму аргументу. Первый аргумент - это содержимое блоков данных этой связи. При создании символической связи проверяется существование файла, на который она указывает, но затем целевой файл может быть удален или переименован, или может быть размонтирована файловая система, на которой он размещен. При этом получится «висячая» символическая связь; обращения к ней будут приводить к ошибке ENOENT.

Файл типа "l" - это файл символической связи. Права доступа для символической связи всегда 777, независимо от значения cmask. Эти права не могут быть изменены. chmod(2) прослеживает символические связи, поэтому изменение прав доступа для символической связи изменяет права доступа файла, на который она указывает.

Файл символической связи может указывать на другой файл символической связи и т.д. Может возникнуть цикл, если цепочка символических связей замыкается на саму себя. В этой ситуации системный вызов возвратит неуспех после того, как проследит заранее установленное число символических связей. По умолчанию такое число равно 20. Системный вызов, обнаруживший слишком большое число последовательных символических связей, возвращает -1 и устанавливает errno равным ELOOP.

Чтение значения символической связи

Системный вызов readlink(2) читает содержимое символической связи (строку с именем файла, на который указывает связь). Его параметры таковы:

path Путевое имя файла символической связи.

buf Адрес, куда нужно поместить данные, прочитанные из блоков данных файла символической связи.

bufsize Размер буфера buf, который ограничивает количество символов, которые будут считаны. Символы в буфере не завершаются нулем.

Для определения необходимого размера буфера рекомендуется использовать pathconf(2) с параметром _PC_PATH_MAX.

Следование символическим связям

Системные вызовы, работающие с файлами, по разному ведут себя по отношению к символическим связям. Большинство изученных ранее системных вызовов, например, open(2) следуют символическим связям, то есть, если передать им имя символической связи, они будут работать с тем файлом, на который указывает связь, а не с самой связью. При использовании таких вызовов, символическая связь «прозрачна» для прикладной программы, то есть ничем не отличается от целевого файла. Например, если xys — это файл символической связи, который указывает на файл abc, open(2) с параметром xyz откроет файл abc. Такой системный вызов вычисляет значения связей, пока не получит файл, не являющийся символической связью.

Некоторые вызовы работают не с целевым файлом, а с самой связью. Про такие вызовы говорят, что они не следуют символическим связям. Даже такие вызовы следуют символическим связям — компонентам путевого имени файла, кроме последней компоненты. Только от системного вызова зависит следует он последней компоненте имени или нет.

Первая группа системных вызовов в таблице следует символическим связям. Вторая группа системных вызовов не следует символическим связям. Эти системные вызовы производят свою операцию над самим файлом символической связи.

Обратите внимание, что системный вызов ulink(2) не следует символическим связям и не имеет аналога, следующего этим связям. Этот системный вызов удаляет файл. Он рассматривается далее в этом разделе. Поскольку ulink(2) не следует символическим связям, такие связи невозможно использовать для удаления файла; при попытке удаления файла через символическую связь будет удаляться связь, но не целевой файл.

следуют символическим связям	open(2)
	chmod(2)
	chown(2)
	chgrp(2)
	chdir(2)
	stat(2)
не следуют символическим связям	lchown(2)
	readlink(2)
	lstat(2)
	link(2)
	unlink(2)
	rename(2)
	rmdir(2)

Удаление записи из директории

Системный вызов unlink(2) удаляет запись из директории, то есть жесткую связь файла. Для удаления записи достаточно иметь право записи в директорию, в которой находится запись, если директория не имеет «липкого» бита. Если директория имеет «липкий» бит, для удаления записи необходимо быть владельцем файла, на который указывает эта запись.

Параметр unlink(2):

path Путевое имя удаляемого файла. Оно может быть абсолютным или относительным.

Инод файла поддерживает счетчик жестких связей, который уменьшается при удалении записей этого файла в директориях. Когда значение счетчика достигает нуля, то файл удаляется — его инод и его блоки данных освобождаются. Как известно, подсчет ссылок не может находить кольцевые списки, поэтому в Unix создание жестких связей на каталоги запрещено.

Если файл открыт каким-то процессом, то запись в директории удаляется нормально, но данные удаляются только тогда, когда исчезают все ссылки на этот файл, то есть не только когда будут удалены записи в директориях, но и когда все процессы закроют этот файл. Это называется задержанным удалением.

Библиотечная функция remove(3C) идентична unlink(2). Она удаляет из директории запись о файле.

Параметр remove(3C):

filename Путевое имя удаляемого файла. Оно может быть абсолютным или относительным.

Функция remove(3C) введена для облегчения переноса программного обеспечения с платформ, где не поддерживается отложенное удаление и нет вызова unlink(2).

При установке обновлений, система не может модифицировать файлы загрузочных модулей и библиотек, которые в данный момент исполняются. Но Unix может удалить такие файлы и подложить на их место новые версии. Процессы, которые были запущены до остановки обновления, будут использовать старые версии бинарников и библиотек, а вновь запускаемые — новые версии. Системы управления пакетами в Unix отслеживают зависимости между пакетами, поэтому, обычно, после установки обновления, система перезапускает процессы, зависимые от обновленного бинарника или библиотеки. Это позволяет устанавливать обновления без перезапуска всей системы. Перезапуск требуется только при установке обновлений ядра ОС.

Команда rm(1) использует unlink(2).

Удаление файла - Пример

Этот пример использует системный вызов unlink(2). Это упрощенная реализация команды rm(1). Программа работает так:

7-10 Удаляется файл, заданный первым аргументом в командной строке.

Программа демонстрируется так:

```
$ ls -l -i
total 10
11621 -rw-r--r--   2 tmm     ustg         96 Jan  3 17:45 data
11621 -rw-r--r--   2 tmm     ustg         96 Jan  3 17:45 data2
23937 -rwxr-xr-x   2 tmm     ustg       3346 Jan  6 08:16 unlink
$ unlink data2
$ ls -l -i
total 9
11621 -rw-r--r--   1 tmm     ustg         96 Jan  3 17:45 data
23937 -rwxr-xr-x   2 tmm     ustg       3346 Jan  6 08:16 unlink
```

Заметьте, что счетчик связей файла data уменьшился до единицы.

Файл: unlink.c

УДАЛЕНИЕ ФАЙЛА - ПРИМЕР

```
 1 #include <stdio.h>
 2 #include <unistd.h>
 3 #include <stdlib.h>
 4
 5 main(int argc, char *argv[])
 6 {
 7     if (unlink(argv[1]) == -1) {
 8         perror(argv[1]);
 9         exit(1);
10     }
11     exit(0);
12 }
```

Переименование файла

Системный вызов rename(2) переименовывает файл. Параметры rename(2):

old Старое путевое имя файла.

new Новое путевое имя.

Переименование возможно только в пределах одной файловой системы. Перенос файлов между файловыми системами должен включать в себя копирование. Для регулярных файлов, rename(2) можно было бы реализовать как последовательность link(2) и unlink(2), но для каталогов это было бы невозможно.

Выделение имени родительской директории из путевого имени

Библиотечная функция dirname(3G) возвращает указатель на строку с именем родительской директорию файла. Параметром dirname(3G) является путевое имя файла.

Библиотечная функция basename(3G) возвращает указатель на последний элемент путевого имени (имя файла относительно его родительской директории).

Чтение символической связи - пример

Программа на следующей странице демонстрирует системные вызовы symlink(2) и readlink(2). Она работает так:

17-20 Создается файл символической связи. Имя связи с файлом, указанным первым параметром в командной строке, - это второй параметр в командной строке.

22-25 Файл символической связи используется как аргумент системного вызова open(2). Поскольку open(2) следует символическим связям, файл переданный в качестве первого аргумента в командной строке будет открыт для чтения. Содержимое файла считывается и распечатывается.

31-34 Файл символической связи используется как аргумент системного вызова readlink(2). Поскольку readlink(2) не следует символическим связям, считывается содержимое блоков данных файла символической связи.

36-37 Содержимое файла символической связи распечатывается.

Эта программа демонстрируется так:

```
$ cat datafile
12345
abcde
$ symlink /usr1/jrs/datafile symdata
contents of datafile:
12345
abcde

contents of symdata: /usr1/jrs/datafile
```

Файл: symlinks.c

ЧТЕНИЕ СИМВОЛИЧЕСКОЙ СВЯЗИ - ПРИМЕР

```
 1 #include <stdio.h>
 2 #include <unistd.h>
 3 #include <stdlib.h>
 4 #include <fcntl.h>
 5 #define LEN 255
 6
 7 main(int argc, char *argv[])
 8 {
 9     char buf[LEN];
10     int n, fd;
...
17     if (symlink(argv[1],argv[2]) == -1) {
18     perror(argv[0]);
19     exit(1);
20     }
21
22     if ((fd = open(argv[2],O_RDONLY)) == -1) {
23     printf("can't open %s\n", argv[2]);
24     exit(2);
25     }
26
27     printf("contents of %s:\n", argv[1]);
28     while ((n = read(fd,buf,255)) != 0)
29     printf("%.*s",n,buf);
30
31     if ((n = readlink(argv[2], buf, LEN)) == -1)  {
32     printf("readlink fails on %s\n", argv[2]);
33     exit(3);
34     }
35
36     printf("\ncontents of %s: ",argv[2]);
37     printf("%.*s\n", n, buf);
38 }
```

8. Сигналы

Обзор

Сигналы — это сервис, предоставляемый ядром системы и используемый, главным образом, для обработки ошибок и исключительных ситуаций. Сигналы могут быть посланы ядром или процессом процессу или группе процессов. Например, сигнал посылается когда нажимается клавиша Ctrl-C, чтобы убить исполняющуюся программу. Также, сигнал посылается, когда программа пытается поделить на ноль. В обоих названных случаях, сигналы посылаются ядром.

Сигналы можно рассматривать как программный аналог аппаратных прерываний. Они могут быть обработаны получающим процессом.

Сигналы

Сигналы — это механизм передачи сообщений между процессами и от ядра к процессам. Они оповещают процесс о том, что произошло определённое событие. Говорят, что сигнал генерируется для процесса (или посылается процессу), когда в первый раз происходит событие, связанное с этим сигналом. Тип сигнала показывает, какое событие произошло.

Сигналы могут генерироваться программно, с использованием системных вызовов kill(2) и sigsend(2). Хотя говорят, что сигналы посылаются от процесса к процессу, на самом деле они посылаются через ядро. Ещё один программный способ генерации сигналов — это системный вызов alarm(2), который позволяет установить будильник (alarm clock). Будильник в Unix — это сигнал, генерируемый ядром в заданный момент времени.

Кроме того, различные аппаратные события могут генерировать сигналы. Например, попытка записать значение по недопустимому виртуальному адресу или деление на ноль вынуждают ядро генерировать сигнал. Внешние события, такие, как разрыв линии связи или нажатие клавиши INTR (Ctrl-C по умолчанию), также могут быть причиной возникновения сигналов.

Процесс может определить действия, которые должны быть выполнены при получении сигнала. Такие действия называются реакцией на сигнал. Каждый сигнал имеет реакцию по умолчанию, которая выполняется, если для этого сигнала в этом процессе не были явно определены какие-то другие действия. Для большинства сигналов реакция по умолчанию — завершение процесса. Процесс имеет возможности либо проигнорировать сигнал, либо исполнить функцию обработки при его получении. Когда для определённого сигнала установлена реакция, она не меняется, пока не будет явным образом установлена другая реакция.

Каждый процесс имеет сигнальную маску, определяющую множество сигналов, получение которых заблокировано. Маска сигналов процесса наследуется от родительского процесса.

В традиционном Unix, процесс может распознать только один необработанный сигнал данного типа. Если второй сигнал того же типа возникнет прежде, чем первый сигнал будет обработан, второй сигнал будет потерян. Современные Unix-системы поддерживают надёжные сигналы, которые доставляются процессу столько же раз, сколько раз возникло соответствующее событие. Такие сигналы отличаются как по процедуре установки обработчика, так и по процедуре генерации (необходимо использовать функцию sigqueue(3C)). В этом курсе такие сигналы не изучаются.

Предупреждение: Будьте осторожны, используя сигналы для синхронизации процессов. Сигнал проявляет своё присутствие для процесса в дискретные моменты времени, в действительности — в моменты передачи управления от ядра к процессу, то есть в момент выхода из системного вызова или при возобновлении процесса планировщиком. Может оказаться, что процесс получит сигнал до или после того момента, когда он был бы готов обработать его.

Типы сигналов

Некоторые сигналы, их действия по умолчанию и события, которые их вызывают, перечислены на следующей странице. Все сигналы перечислены в signal(5); для всех сигналов определены символьные константы в <signal.h>. В соответствии со стандартом POSIX, рекомендуется использовать символические имена, а не численные значения номеров сигналов.

Сигналы от SIGHUP до SIGTERM генерируются ядром, когда возникает соответствующее событие. Сигналы SIGUSR1 и SIGUSR2 могут использоваться программистами для своих целей.

Пример посылки сигнала от ядра к вашей программе — использование клавиши INTR (Ctrl-C по умолчанию), при нажатии которой на терминале генерируется сигнал SIGINT. Ядро, обнаружив незаконное обращение к памяти в вашей программе, генерирует сигнал "нарушение сегментации" (segmentation violation) — SIGSEGV. Аналогично, деление на ноль с плавающей точкой генерирует сигнал "особая ситуации при работе с плавающей точкой (floating point exeption) SIGFPE. Этот же сигнал генерируется и при целочисленном делении на ноль.

Некоторые сигналы зависят от аппаратуры, такие, как SIGEMT и SIGBUS. Поэтому причины и значение этих сигналов могут меняться.

Не путайте аппаратные прерывания и исключения с сигналами. Некоторые аппаратные исключения обрабатываются ядром и иногда переводятся в сигналы, которые посылаются вашей программе.

сигнал	реакция	событие
SIGHUP	**exit**	обрыв линии (см. termio(7))
SIGINT	**exit**	прерывание (см. termio(7))
SIGQUIT	**core**	завершение (см. termio(7))
SIGILL	**core**	неправильная инструкция
SIGTRAP	**core**	прерывание трассировки
SIGABORT	**core**	аборт
SIGEMT	**core**	команда EMT (программное прерывание)
SIGFPE	**core**	арифметическая особая ситуация
SIGKILL	**exit**	принудительное завершение ("убийство")
SIGBUS	**core**	ошибка шины
SIGSEGV	**core**	нарушение сегментации
SIGPIPE	**exit**	разрыв конвейера
SIGALRM	**exit**	будильник

SIGTERM	**exit**	программный сигнал прерывания от kill
SIGCLD	**ignore**	изменение состояния подпроцесса
SIGPWR	**ignore**	сбой питания
SIGSTOP	**stop**	остановка (сигналом)
SIGTSTP	**stop**	остановка (пользователем) (см. termio(7))
SIGCONT	**ignore**	продолжение
SIGTTIN	**stop**	ожидание ввода с терминала(см. termio(7))
SIGTTOU	**stop**	ожидание вывода на терминал (см.termio(7))
SIGVTALRM	**exit**	сигнал виртуального таймера
SIGPROF	**exit**	сигнал таймера профилирования
SIGXCPU	**core**	исчерпался лимит времени (см. getrlimit(2))
SIGXFSZ	**core**	выход за пределы длины файла (см. getrlimit(2))

Получение сигнала

Сигнал обрабатывается получающим процессом. Даже при выполнении действий по умолчанию, когда пользовательский код не вызывается, всё равно обработчик исполняется в контексте процесса, получающего сигнал.

Доставка сигналов прерывает некоторые системные вызовы, такие как wait(2) или read(2) с медленного символьного устройства. Если сигнал не привёл к завершению процесса, прерванные вызовы возвращают индикацию неуспеха (например, -1) и устанавливают код ошибки равным EINTR. Если вызов был прерван, возможно, вам следует вызвать его снова. SVR4.0 предоставляет возможность автоматического перезапуска системных вызовов, прерванных из-за перехвата сигнала. Это будет обсуждаться, когда пойдёт речь о sigaction(2).

Если не указано иное, процесс выполняет при получении сигнала реакцию по умолчанию. Реакция по умолчанию (SIG_DFL) для каждого сигнала указана на предыдущей странице. В Unix-системах бывает четыре типа реакции по умолчанию:

exit получающий процесс завершается. Слово состояния процесса при этом содержит признак, что процесс был завершён по сигналу.

core получающий процесс завершается, как и при реакции exit. Кроме того, в текущей рабочей директории создаётся дамп памяти (core-файл).

stop получающий процесс останавливается.

ignore получающий процесс игнорирует этот сигнал. Это идентично установке реакции в SIG_IGN.

Если реакция на сигнал установлена в SIG_IGN, полученный сигнал игнорируется, как если бы он никогда не возникал. Если реакция на сигнал является адресом функции, при получении сигнала процесс исполнит функцию обработки по этому адресу.

Когда функция обработки сигнала завершается, получающий процесс возобновляет исполнение с того места, где он был прерван, если функция обработки не сделала других распоряжений.

Реакции на сигналы SIGKILL и SIGSTOP не могут быть изменены; кроме того, доставка этих сигналов не может быть заблокирована маской. Когда процесс получает сигналы SIGSTOP, SIGTSTP, SIGTTIN или SIGTTOU, независимо от установленной реакции на них, ожидающий сигнал SIGCONT будет отменен. Когда процесс получает сигнал SIGCONT, независимо от установленной реакции на этот сигнал, все ожидающие сигналы SIGSTOP, SIGTSTP, SIGTTIN или SIGTTOU будут отменены. Кроме того, если процесс был остановлен, он продолжит исполнение.

Установка реакции на сигнал

Системные вызовы signal(2) и sigset(2) устанавливают реакцию на сигнал, т.е. действия, которые процесс предпринимает, когда получает заданный сигнал. Аргументы:

sig задаёт номер сигнала (кроме SIGKILL или SIGSTOP).

disp задаёт реакцию и может принимать одно из значений:

. SIG_DFL - при получении сигнала sig выполняется реакция по умолчанию.

. SIG_IGN - сигнал sig игнорируется.

. адрес функции - При получении сигнала процесс вызывает заданную функцию. Значение sig передается этой функции как аргумент. Это называется перехватом (catching) сигнала.

. SIG_HOLD (только у sigset(2)) — сигнал блокируется, как если бы он был запрещён маской. Поступающие сигналы соответствующего типа не игнорируются, а запоминаются и при смене реакции на сигнал будут получены процессом.

При использовании signal(2), после перехвата сигнала, реакция на этот сигнала сбрасывается к реакции по умолчанию. Если вы хотите снова перехватывать этот сигнал, вы должны опять переустановить реакцию на вашу функцию обработки. Удобное место для этого — сама функция-обработчик. Но при этом всё равно существует окно, в течении которого действует реакция по умолчанию.

При использовании sigset(2), ядро задерживает новые сигналы до момента возврата из функции обработки. После завершения обработчика, реакция на сигнал снова устанавливается ядром в то же значение, которое было установлено вызовом sigset(2). Если в это время был получен ещё один сигнал, опять вызовется функция обработки.

Вы можете задать различные реакции для каждого сигнала. Также можно использовать одну функцию для обработки различных сигналов.

Значение, возвращаемое signal(2) и sigset(2) — предыдущая установка реакции. Вы можете сохранить это значение и затем восстановить реакцию такой, какой она была изначально.

При выполнении exec(2), сигналы, реакция на которые была установлена в указатель на функцию, сбрасываются в SIG_DFL, в то время, как реакции SIG_IGN и SIG_HOLD действуют и далее.

Перехват сигнала - пример

Программа в этом примере вычисляет простые числа от 1 до MAXNUM. Эта программа требует много времени для выполнения. Вы можете убить эту программу в любой момент нажатием клавиши Ctrl-C. Вместо того, чтобы просто завершиться, программа перехватывает SIGINT и исполняет функцию, которая показывает количество вычисленных к данному моменту простых чисел, закрывает выходной файл и завершается.

12 Приведено объявление функции обработки сигнала, чтобы передать вызову signal(2) параметр нужного типа (указательн на функцию).

15 Вызывается signal(2). Теперь при получении сигнала SIGINT будет вызвана функция sigcatch(). Заметьте, что если сигнал никогда не будет получен, sigcatch() никогда не будет вызвана.

... Полный листинг программы приведен в конце раздела. Код, который не показан, выполняет собственно вычисление простых чисел.

29-34 Эта функция обработки сигнала будет вызвана, когда будет перехвачен сигнал SIGINT. Функция выводит сообщение, закрывает выходной файл и завершает программу.

Если бы exit(2) не был вызван в функции обработки сигнала, исполнение возобновилось бы с того места, где оно остановилось внутри main(2). Однако, второй полученный SIGINT прервал бы эту программу, потому что реакция на сигнал была бы установлена ядром в SIG_DFL.

Файл: primes1.c

ПЕРЕХВАТ СИГНАЛА - ПРИМЕР

```
1   #include <stdio.h>
2   #include <signal.h>
3   #define   OUTPUT   "Primes"
4   #define   MAXNUM   10000
5
6   int count;
7   FILE *fptr;
8
9   main()
10  {
11     int number, divisor;
12     void sigcatch();
13
14     fptr = fopen(OUTPUT, "w");
15     signal(SIGINT, sigcatch);
   ...
25     fclose(fptr);
26  }
27
28
29  void sigcatch()
30  {
31     printf("%d primes computed\n", count);
32     fclose(fptr);
33     exit(1);
34  }
```

Переустановка реакции на сигнал - пример

Эта программа похожа на предыдущий пример, за исключением того, что SIGINT может быть перехвачен много раз без завершения программы. Каждый раз, когда SIGINT перехватывается, вызывается функция обработки, которая печатает сообщение и возобновляет исполнение с того места, где был перехвачен сигнал.

14-15 Устанавливается реакция на сигналы SIGINT и SIGQUIT. Сигнал SIGQUIT генерируется символом FS (клавиши CTRL и \, нажатые одновременно). Заметьте, что реакция на оба сигнала — одна и та же функция.

29-38 Значение сигнала передаётся как аргумент функции sigcatch(). При перехвате ядро сбрасывает реакцию на сигнал к реакции по умолчанию. Поэтому функция обработки начинает с того что требует ядро игнорировать сигналы того типа, который возник. Это предохраняет программу от завершения, если второй сигнал возникнет во время вычисления функции обработки. Затем, в конце функции, реакция на сигнал снова переустанавливается на вызов sigcatch(). Это нужно сделать для того, чтобы перехватить сигнал снова.

Между перехватом сигнала и вызовом signal(2) с SIG_IGN в функции обработки сигнала, существует небольшой интервал времени, в котором вновь прибывший сигнал может завершить программу. Такая аномалия более вероятна на сильно загруженной системе. Вы можете видеть из вывода программы, что каждый раз, когда нажимается Ctrl-C, вызывается функция обработки.

```
$ primes2
<Ctrl-C>
111 primes computed
<Ctrl-C>
132 primes computed
<Ctrl-C>
147 primes computed
<Ctrl-C>
177 primes computed
<Ctrl-C>
203 primes computed
<CTRL \>
224 primes computed
```

Файл: primes2.c

ПЕРЕУСТАНОВКА РЕАКЦИИ НА СИГНАЛ - ПРИМЕР

```
1   #include <stdio.h>
2   #include <signal.h>
3   #define   OUTPUT   "Primes"
4   #define   MAXNUM   10000
5   int count;
6   FILE *fptr;
7
8   main()
9   {
10      int number, divisor;
11      void sigcatch(int);
12
13      fptr = fopen(OUTPUT, "w");
14      signal(SIGINT, sigcatch);
15      signal(SIGQUIT, sigcatch);
   ...
25      fclose(fptr);
26   }
27
28
29   void sigcatch(int sig)
30   {
31      signal(sig, SIG_IGN);
32      printf("%d primes computed\n",count);
33      if (sig == SIGQUIT) {
34         fclose(fptr);
35         exit(1);
36      }
37      signal(sig, sigcatch);
38   }
```

Игнорирование сигнала - пример - перенаправление вывода.

Эта полезная программа позволяет вам оставлять исполняющуюся в фоновом режиме (background) программу после того, как вы выйдете из системы. Это достигается путем игнорирования сигнала SIGHUP, и исполнения после этого новой программы, командная строка которой переда`тся как аргументы. Этот пример является упрощенной версией команды nohup(1).

В командных процессорах с управлением заданиями, фоновые задания не получают SIGHUP (его получают только процессы первого плана, см. раздел «Терминальный ввод-вывод»), поэтому в таких командных процессорах эта команда не нужна. Для проверки того, что в командном процессоре без управления заданиями ваша программа работает как заявлено, вы можете заменить командный процессор на sh командой exec sh или запустить терминальный эмулятор, указав, что запускать в терминальной сессии, например, командой gnome-term -x /bin/sh -i. Чтобы убедиться, что ваша программа продолжает исполняться после закрытия сессии, можно вывести полный список ваших процессов командой ps -u $USER. Обычно, графическая сессия содержит много процессов; для поиска вашего процесса можно воспользоваться утилитой grep, например ps -u $USER | grep command-name.

15-20 Закрывается дескриптор файла 1. Затем, открывается выходной файл для перенаправления стандартного вывода. open возвращает дескриптор файла 1. Это не является рекомендованным способом переназначать ввод-вывод, но такой код короче.

22 Реакция на сигнал SIGHUP установлена так, чтобы игнорировать этот сигнал.

23 Запускается новая программа с именем argv[1]. Заметьте, что реакции на сигналы SIG_IGN и SIG_DFL не изменяются системным вызовом exec(2). Однако, если бы реакция была установлена на какую-либо функцию обработки, то она была бы сброшена ядром к SIG_DFL, потому что значение указателя на функцию становится неправильным в новой программе.

Файл: hangup.c

ИГНОРИРОВАНИЕ СИГНАЛА - ПРИМЕР
ПЕРЕНАПРАВЛЕНИЕ ВЫВОДА

```
1  #include <sys/types.h>
2  #include <fcntl.h>
3  #include <sys/stat.h>
4  #include <signal.h>
5  #define  FILE   "Hangup.out"
6
7  main(int argc, char *argv[])
8  {
9     int fd;
10
11     if (argc < 2) {
12        printf("usage: %s command [args]\n",argv
13        exit(1);
14     }
15     close(1);    /* redirect standard output */
16     if ((fd = open(FILE, O_WRONLY | O_CREAT |
17                     O_APPEND, 0644)) == -1) {
18        perror(FILE);
19        exit(2);
20     }
21
22     signal(SIGHUP, SIG_IGN);
23     execvp(argv[1], &argv[1]);
24     perror(argv[1]);
25     exit(127);
26  }
```

Генерация сигналов

Сигналы могут генерироваться различными способами. Для того, чтобы программно послать сигнал, вы можете использовать команду kill(1) или системные вызовы kill(2) или sigsend(2). Вы можете посылать сигналы только тем процессам, которые имеют ваш идентификатор пользователя. Однако, суперпользователь (root) и/или обладатель привилегии PRIV_PROC_OWNER может послать сигнал любому процессу. Также, можно посылать сигнал SIGCONT любому процессу своей терминальной сессии.

Кроме того, вы можете установить будильник (alarm clock), используя системный вызов alarm(2). После того, как интервал времени в секундах, заданный в качестве аргумента alarm, закончится, ядро пошлёт сигнал процессу, вызвавшему alarm(2).

Кроме того, сигналы генерируются при ошибках, таких, как плохой указатель или ошибка деления с плавающей точкой на ноль. При нажатии некоторых клавиш на клавиатуре терминала, также генерируются сигналы. Например, нажатие клавиши Ctrl-C распознаётся кодом драйвера терминального устройства в ядре и превращается в сигнал SIGINT.

Посылка сигнала

Вы можете послать сигнал одному или нескольким процессам, используя системный вызов kill(2). Аргументы:

sig - сигнал, который нужно послать. Если sig равен нулю, сигнал не посылается, но выполняется проверка ошибок. Это можно использовать для проверки допустимости значения pid.

pid определяет процесс(ы), которые должны получить сигнал. Следующая таблица представляет собой список всех возможных значений pid и их смысл.

pid	получающий процесс (процессы)
> 0	идентификатор процесса
0	все процессы группы, к которой принадлежит данный процесс
-1	все процессы, чей действительный ID равен эффективному ID посылающего процесса
< -1	все процессы в группе процессов -pid

Из Справочного руководства программиста по kill(2):

sigsend(2) является более гибким способом посылки сигналов процессам. Пользователю рекомендуется использовать sigsend(2) вместо kill(2). sigsend(2) рассматривается далее в этом разделе.

Аналогичным образом интерпретирует свой параметр одноимённая команда kill(1). Так, передав -1 в качестве номера процесса, вы можете послать сигнал всем вашим процессам. Например, это может быть полезно, чтобы убить форк-бомбу. Если вы случайно запустили процесс, который вызывает fork(2) в бесконечном цикле, потомки такого процесса быстро переполнят вашу квоту процессов и вы не сможете запустить ни одной команды, даже ps(1), чтобы определить номер родительского процесса или группы процессов форк-бомбы. Для убийства такого дерева процессов можно зайти в систему по ssh(1) от имени другого пользователя, сделать su your-username (при смене uid при помощи setuid(2), не проверяется переполнение квоты процессов) и из полученной командной оболочки убить все ваши процессы командой kill -SIGKILL -1. Если написать просто kill -1, команда kill(1) воспримет параметр, начинающийся с -, как номер сигнала (в данном случае это будет SIGHUP), поэтому обязательно нужно указать номер сигнала.

Посылка сигнала

sigsend(2) предоставляет более гибкий способ посылки сигнала, чем системный вызов kill(2). Аргументы:

sig - сигнал, который нужно послать. Если sig равен нулю, сигнал не посылается, но выполняется проверка ошибок.

idtype определяет, как будет интерпретироваться id

id определяет процесс(ы), которые должны получить сигнал.

Следующая таблица представляет собой список возможных значений idtype и соответствующий смысл аргумента id.

idtype	Получающий процесс (процессы)
P_PID	процесс, чей PID равен id
P_PGID	все процессы, чей ID группы процессов равен id
P_SID	все процессы, чей ID сессии равен id
P_UID	все процессы, чей EUID равен id
P_GID	все процессы, чей EGID равен id
P_CID	все процессы, чей ID класса планировщика равен id [см. priocntl(2)]
P_ALL	все процессы; id игнорируется
P_MYID	вызывающий процесс; id игнорируется

Принудительное завершение подпроцессов - пример

Этот пример показывает, каким образом вы можете убить несколько порожденных вами процессов, если один из них возвращает плохой код.

13-17 Запускаются пять подпроцессов, и их идентификаторы записываются в chpid[].

19-26 wait(2) в цикле ожидает завершения всех подпроцессов.

22-25 Если какой-то из подпроцессов завершается с плохим статусным кодом, то все остальные принудительно завершаются посылкой сигнала SIGTERM. Заметьте, что сигнал посылается также тому процессу, который только что завершился, но это не причиняет вреда. Внутри цикла каждый порождённый процесс ожидается индивидуально, повторяющимися вызовами wait(2).

Эта программа исполняется каждым из подпроцессов. Каждому из них требуется различное время для того, чтобы завершиться. Различное время выполнения изображается вызовом библиотечной функции sleep(3C). sleep(3C) будет описано ниже. Каждый процесс спит случайное количество времени.

Замечание: сначала генератор случайных чисел должен быть инициализирован с использованием srand(3C), после чего rand(3C) выдаёт случайные числа. См. rand(3C) для более детального знакомства.

Рассмотрите выполнение этого примера. Получилось так, что порождённый процесс номер четыре завершился первым, с кодом выхода равным 1 (status равен 256). Поэтому остальные подпроцессы будут завершены сигналом SIGTERM. Заметьте, что здесь печатается неразобранное слово состояния процесса, а не код завершения exit(2).

Это программа, которую исполняют порожденные процессы:

```
1       #include <stdlib.h>
2       #include <sys/types.h>
3       main()
4       {
5           srand( (unsigned)getpid() );
6
7           /* processing done here */
8           sleep((rand() % 5) + 1);
9           exit(1);
10       }
```

```
$ parent
parent: waiting for children
child 9725: terminated, status: 256
child 9722: terminated, status: 15
child 9723: terminated, status: 15
child 9724: terminated, status: 15
child 9726: terminated, status: 15
parent: all children terminated
```

Файлы: parent.c и child.c

ПРИНУДИТЕЛЬНОЕ ЗАВЕРШЕНИЕ ПОДПРОЦЕССОВ - ПРИМЕР

```
1   #include <sys/types.h>
2   #include <sys/signal.h>
3   #include <sys/procset.h>
4   #include <unistd.h>
5   #include <wait.h>
6   #define  NUMCHILD 5
7
8   main(int argc, char **argv)
9   {
10      int i, status;
11      pid_t pid, chpid[NUMCHILD];
12
13      for (i = 0; i < NUMCHILD; i++) {
14          if ((pid = fork()) == 0)
15              execlp("child","child",(char*) 0);
16          chpid[i] = pid;
17      }
18      printf("parent: waiting for children\n");
19      while ((pid = wait(&status)) != -1) {
20          printf("child %d: terminated, status:
21                          %d\n", pid, status);
22          if (WEXITSTATUS(status) != 0) {
23              for (i = 0; i < NUMCHILD; i++)
24                  sigsend(P_PID, chpid[i], SI
25          }
26      }
27      printf("parent: all children terinated\n")
28
29      exit(0);
30  }
```

Воздействие сигнала на ввод

Этот пример показывает результат получения сигнала во время чтения с терминала.

12-13 Если сигнал игнорировался, то действие по прежнему будет состоять в игнорировании сигнала.

14-24 Этот цикл читает строки со стандартного ввода и пишет их на стандартный вывод.

14-19 Если read(2) завершается неудачно из-за получения сигнала, исполнение продолжается с начала цикла. Иначе цикл завершается, потому что достигнут настоящий конец файла.

Ниже показано, как работает этот пример. Необходимо выполнять демонстрацию из под командной оболочки /bin/sh, чтобы гарантировать, что все процессы принадлежат к одной группе. Сначала в фоновом режиме запускается команда, которая спит, а потом посылает сигнала SIGINT (сигнал номер 2) всем процессам в группе, управляемой TTY.

Замечание: Интерпретатор shell получает сигнал прерывания, но игнорирует его.

Затем, сигнал прибывает во время печатания букв X. Эти буквы должны быть прочитаны строкой 18 программы. Заметьте, что ввод из-за сигнала не теряется.

```
$ sh
$ (sleep 5; kill -2 0) &
19952
$ input
XXXXXsignal 2 received
XX
n: 8
XXXXXXX
<CTRL d>
```

Файл: input.c

ВОЗДЕЙСТВИЕ СИГНАЛА НА ВВОД

```
1   #include <stdio.h>
2   #include <signal.h>
3   #include <errno.h>
4   #include <stdlib.h>
5
6   main()
7   {
8   void sigcatch(int);
9   char buf[BUFSIZ];
10   int n;
11
12   signal(SIGINT, sigcatch);
13
14   for (;;) {
15      if ((n = read(0, buf, BUFSIZ)) <= 0) {
16         if (errno == EINTR) {
17            errno = 0;
18            continue;
19         }
20         break;
21      }
22      printf("n: %d\n", n);
23      write(1, buf, n);
24   }
25   exit(0);
26   }
27
28   void sigcatch(int sig)
29   {
30   signal(SIGINT, SIG_IGN);
31   printf("signal %d received\n", sig);
32   signal(SIGINT, sigcatch);
33   }
```

Будильник (alarm clock)

Системный вызов alarm(2) используется для того, чтобы установить будильник для процесса, который исполнил этот вызов. Через sec секунд после вызова alarm, ядро генерирует сигнал SIGALRM и посылает его этому процессу.

Одно из использований alarm(2) — установка предела времени для системных вызовов, которые могут слишком долго исполняться. Например, сигнал SIGALRM может прервать read(2), который не получает никакого ввода с терминала.

В каждый момент в одном процессе может существовать только одна установка будильника. Если следующий системный вызов alarm(2) будет исполнен до того, как истечет время, установленное предыдущим вызовом, новый вызов возвратит количество времени, оставшееся от первой установки (0, если ничего не было установлено), и будильник приобретет новое значение. Если задано 0 секунд, будильник будет выключен.

Ограничение процесса по времени - Пример

Этот полезный пример начинает исполнять какую-либо команду и прерывает ее, если она исполняется дольше заданного времени. Команда, у которой ограничено время исполнения, запускается как порождённый процесс. Будильник устанавливается на заданное время, и затем родительский процесс ожидает, когда порождённый завершится. Если до этого момента возникнет сигнал SIGALRM, то порождённый процесс принудительно завершается сигналом SIGKILL из родительской функции обработки сигнала.

14-18 Команда, за исполнением которой мы наблюдаем, запускается как порождённый процесс.

19-20 Определяется функция обработки, для того чтобы отреагировать на истечение времени будильника. После этого будильник устанавливается на заданный интервал времени.

21 Здесь процесс ждёт завершения порождённого процесса. Если wait(2) будет прерван сигналом, он будет запущен снова, так как он находится внутри цикла while. Когда истекает время, на которое был установлен будильник, вызывается sigalarm(), и wait(2) возвращает -1. Следующий вызов wait(2) в цикле подтверждает смерть порождённого процесса.

22-29 Если wait(2) возвращает неудачу, проверяется, произошло ли это из-за получения сигнала. Иначе считается, что возникла какая-то ошибка, и цикл прерывается.

35-38 Это функция обработки сигнала SIGALRM. Когда возникает сигнал SIGALRM, она убивает порождённый процесс.

Аналогичного эффекта можно было бы достичь проще, установив будильник в порождённом процессе, после fork(2), но перед вызовом exec(2). Но некоторые процессы могут переустанавливать будильник, например, при вызове sleep(3C), и устанавливать обработчики SIGALARM.

Файл: timeout.c
ОГРАНИЧЕНИЕ ПРОЦЕССА ПО ВРЕМЕНИ - ПРИМЕР

```
 1  #include <stdio.h>
 2  #include <sys/types.h>
 3  #include <signal.h>
 4  #include <sys/procset.h>
 5  #include <errno.h>
 6  #include <unistd.h>
 7  #define TIMEOUT 10
 8  static int pid;
 9
10   main(int argc, char **argv)
11   {
12        void sigalarm(int);
13        int status;
14        if ((pid = fork()) == 0) {
15           execvp(argv[1], &argv[1]);
16           perror(argv[1]);
17           exit(127);
18        }
19        signal(SIGALRM, sigalarm);
20        alarm(TIMEOUT);
21        while (wait(&status) == -1) {
22           if (errno == EINTR) {
23              errno = 0;
24              printf("%s: timed out\n", argv[1]);
25           }
26           else {
```

```
27              perror(argv[0]);
28              break;
29          }
30      }
31      printf("time remaining: %d\n", alarm(0));
32      exit(WEXITSTATUS(status));
33  }
34
35  void sigalarm(int sig)
36  {
37      sigsend(P_PID, pid, SIGKILL);
38  }
```

Нелокальный goto

Библиотечная функция longjmp(3C) часто описывается как аналог оператора goto для перехода между функциями. Управление может быть передано только в точку, которая была помечена вызовом setjmp(3C). На самом деле, более точным аналогом пары функций setjmp(3C)/longjmp(3C) являются операторы try/throw языков C++ или Java/C#.

setjmp(3), который метит точку для передачи управления, должен получить в качестве параметра буфер для сохранения текущего контекста функции в данной точке. longjmp(3) вызывается с тем же самым буфером в качестве параметра. Исполнение возобновится с положения последнего setjmp(3) для этого буфера. Значение val будет возвращено setjmp(3) после выполнения longjmp(3), и должно быть ненулевым.

Тип jmp_buf представляет собой массив целых чисел, описанный с помощью конструкции typedef. При использовании этого типа в качестве параметра он передаётся как указатель, поэтому, на самом деле, для функции setjmp(3C) он представляет собой выходной параметр, хотя и выглядит как передаваемый по значению. Число элементов в этом массиве зависит от аппаратной архитектуры и от компилятора, точнее от используемого компилятором соглашения о формате стекового кадра.

Функция longjmp(3) возвращает управление к последнему setjmp(3), который заполнил буфер env, используемый как аргумент longjmp. Различие между прямым вызовом setjmp(3) и возвратом к нему через longjmp(3) состоит в том, что возвращаемое значение будет, соответственно, нулевым и ненулевым. Если longjmp(3) получит ноль в качестве val, значение, выданное setjmp(3), будет 1.

Буфер, созданный вызовом setjmp(3C), действителен только «вниз по стеку» от той функции, где он был создан. Иными словами, если вы создали буфер buf в функции foo(), то вызывать longjmp(3C) с этим буфером можно только из тех функций, которые были прямо или косвенно вызваны из foo(). После возврата из foo() буфер становится недействителен и при повторном вызове foo() его необходимо создать заново. Ни при каких обстоятельствах нельзя вызывать longjmp(3C) с буфером, созданным в другой нити.

Нарушение этих требований не контролируется компилятором, но практически неизбежно приводит к разрушению стека вызовов. Обычно такое разрушение приводит к завершению программы по SIGSEGV, но на практике SIGSEGV может быть получен не сразу после разрушения стека; до получения этого сигнала, программа может выполнить практически произвольный участок кода с произвольными значениями параметров и локальных переменных.

При использовании функций setjmp/longjmp в программах на C++, необходимо следить за соответствием используемой версии libc и версии компилятора C++. Так, на Solaris, при использовании longjmp в программах, скомпилированных SunStudio, при longjmp выполняются деструкторы локальных переменных в уничтожаемых стековых кадрах функций, а при компиляции компилятором GNU CC — не выполняются.

Справки: /usr/include/setjmp.h

setjmp(3C) и longjmp(3C) - Пример

Эта программа демонстрирует, каким образом setjmp(3C) и longjmp(3C) можно использовать при обработке сигналов для того, чтобы воздействовать на исполнение основного потока управления программы. Эта программа представляет собой циклическое меню внутри функции main. Вызов setjmp(3C) используется для того, чтобы пометить точку в программе непосредственно перед циклом.

Меню в примере содержит следующие команды:

s печатать квадраты целых чисел. Исполнение этой команды само по себе не заканчивается, и её необходимо прерывать.

t печатать дату и время

q выйти из программы

Для каждой буквы-команды вызывается определенная функция, выполняющая требуемую операцию. Если пользователь нажимает <Ctrl-C>, программа продолжается с точки, помеченной setjmp, так как функция обработки сигнала вызывает longjmp(3), вместо возврата к месту, где находилось управление до перехвата сигнала.

2-3 Включить <setjmp.h> и объявить переменную env типа jmp_buf.

8 Сигналы прерывания будут перехватываться и обрабатываться заданной пользователем функцией ljmain.

9 Текущая точка программы помечена setjmp(3). Возвращаемое значение будет нулевым. Оно станет ненулевым при последующих переходах на эту точку при вызове longjmp.

12 Вход в цикл меню.

15-26 Меню вызывает функции, выполняющие требуемые команды.

28-33 Функция обработки сигнала SIGINT переустанавливает сигнал на ту же функцию (т.е. на себя), печатает "INTERRUPTED" и выполняет longjmp обратно в функцию main.

... Не показаны sfn(), которая печатает последовательность квадратов, и tfn(), печатающая дату и время.

Возможно несколько вызовов setjmp(3), позволяющие делать longjmp(3) в различные точки. Однако, каждый вызов требует своей собственной переменной jmp_buf.

Обратите внимание, что в этой программе обработчик устанавливается вызовом signal(2). Дело в том, что при установке обработчика через sigset(2), сигнал блокируется до возврата из обработчика. Но при вызове longjmp(3C) из обработчика, возврата из обработчика не происходит, и сигнал так и останется заблокированным. Для решения этой проблемы, совместно с sigset(2) необходимо использовать функции sigsetjmp(3C)/siglongjmp(3C), которые в нашем курсе подробно не изучаются.

Этот пример демонстрируется следующим образом:

```
$ setjmp
cmd: s
0    14916253649
64   81100121144169225
<DELETE>
INTERRUPTED
Before loop
cmd: q
$
```

Файл: setjmp.c

setjmp(3C) И longjmp(3C) - ПРИМЕР

```
1   #include <signal.h>
2   #include <setjmp.h>
3   jmp_buf env1;
4   void ljmain(int);
5
6   main() {
7         int retcode;
8         signal(SIGINT, ljmain);
9         retcode = setjmp(env1);
10         if (retcode != 0) printf("\nBefore loop\n");
11
12         while( 1 ) menu();
13  }
14
15  menu() {
16          char resp[2];
17  prompt: printf("cmd: ");
18          scanf("%s",resp);
19          switch(resp[0]){
20                  case 's': sfn(); break;
21                  case 't': tfn(); break;
22                  case 'q': exit(0);
23                  default:  printf("?\n");
24                          goto prompt;
25          }
26  }
27
28  void ljmain(int sig)
29  {
30          signal(SIGINT, ljmain);
31          printf("\nINTERRUPTED\n");
32          longjmp(env1, 1);
33  }
34
...
```

Задержка процесса до сигнала

Системный вызов pause(2) задерживает исполнение процесса на неопределённое время до момента, пока не возникнет неигнорируемый сигнал.

В традиционных Unix-системах, pause(2) используется вместе с alarm(2) для реализации библиотечной функции sleep(3C). Если прекращение pause(2) было вызвано сигналом, отличным от SIGALRM, необходимо сбросить будильник вызовом alarm(2) с нулевым аргументом. Иначе, ваша программа будет вести себя непредсказуемым образом из-за получения "дополнительного" сигнала. Заметьте, что alarm(2) может быть использован вместе с другими блокирующимися системными вызовами, такими как read(2) или wait(2).

Задержка исполнения на заданный промежуток времени - Пример

Этот пример — упрощенная версия функции sleep(3C) в традиционной версии библиотеки libc. Он демонстрирует совместное использование системных вызовов alarm(2) и pause(2). longjmp(3) используется для возврата из функции обработки сигнала.

В многопоточной версии libc такая реализация sleep(3C) недопустима, ведь процесс имеет только один будильник, и его переустановка в разных потоках приводила бы к конфликтам. Поэтому, в Solaris 9 и последующих версиях, функция sleep(3C) реализована через специальный системный вызов. Тем не менее, данный пример интересен тем, что показывает способ обхода так называемой «ошибки потерянного пробуждения» (lost wakeup error).

12 Устанавливается функция обработки для сигнала SIGALRM. Предыдущая установка реакции запоминается.

13 Если это был прямой вызов setjmp(3), то выполняются операторы внутри if. Иначе, если мы вернулись сюда через longjmp(3), возвращаемое значение будет ненулевым, и тело оператора if выполняться не будет.

14-15 Эти строки кода устанавливают интервал времени. setjmp(3) и longjmp(3) используются, потому что после установки будильника существует вероятность, что сигнал возникнет до вызова pause(2). Это может произойти при большой загрузке системы или, например, если ноутбук с системой будет переведён в спящий режим. В этом случае, процесс будет задержан навсегда. При использовании longjmp(3) в функции обработки сигнала, передача управления назад к setjmp(3) происходит всегда, независимо от того, была вызвана pause(2) или нет.

17 Если переменная unslept положительна, значит возврат из pause(2) был вызван каким-то другим сигналом, а не SIGALRM. Кроме того, будильник выключается, чтобы предотвратить нежелательный сигнал SIGALRM. Таким образом, эта строка служит не только для получения количества "недоспанного" времени.

23-27 Это функция обработки сигнала SIGALRM. Заметьте, что это статическая функция, так что она будет локальной в этом файле исходного кода. Передача управления назад к setjmp(3) обсуждалась выше.

Эта программа тестируется следующим драйвером:

```
30      #ifdef DEBUG
31      main(int argc, char **argv)
32      {
33      int  unslept, mysleep(int);
34      void sigint();
35
36      signal(SIGINT, sigint);
37      unslept = mysleep(atoi(argv[1]));
38      printf("remaining time: %d\n", unslept);
39      }
40
41     void sigint() {}
42     #endif
```

Файл: mysleep.c

ЗАДЕРЖКА ИСПОЛНЕНИЯ НА ЗАДАННЫЙ ПРОМЕЖУТОК ВРЕМЕНИ - ПРИМЕР

```
1       #include <signal.h>
2       #include <setjmp.h>
3       #include <stdlib.h>
4
5       static jmp_buf env;
6
7       mysleep(int seconds)
8       {
9       void sigcatch(int), (*astat)(int);
10       int unslept = seconds;
11
12       astat = signal(SIGALRM, sigcatch);
13       if (setjmp(env) == 0)  {
14       alarm(seconds);
15       pause();
16       }
17       unslept = alarm(0);
18       signal(SIGALRM, astat);
19       return(unslept);
20       }
21
22
23       static void sigcatch(int sig)
24       {
25       longjmp(env, 1);
26       }
27
```

Задержка исполнения на заданный промежуток времени

Библиотечная функция sleep(3C) используется, если вы хотите задержать исполнение вашей программы на несколько секунд. Упрощенная реализация этой функции для однопоточной среды была только что обсуждена.

Перехваченный сигнал вызывает преждевременное завершение sleep(3C). Возвращаемое значение — количество "недоспанных" секунд.

Управление сигналами

В ОС UNIX System V Версия 3 были введены новые системные вызовы для управления сигналами, которые лучше поддерживают сигналы и исправляют некоторые из недостатков системного вызова signal(2). В частности, при перехвате сигнала, реакция на который установлена с помощью signal(2), ядро сбрасывает реакцию на него в реакцию по умолчанию. Прежде чем реакция на сигнал будет установлена в требуемое значение, вновь возникший сигнал того же типа может убить процесс.

При перехвате сигналов с использованием sigset(2) эта проблема не возникает. При получении сигнала ядро автоматически устанавливает задержку таких сигналов до возврата из функции обработки. После этого, реакция устанавливается ядром на то значение, которое было изначально задано вызовом sigset(2). Если в промежутке был задержан сигнал, он освобождается, и функция обработки вызывается снова тем же образом.

Руководство по signal(2) содержит набор вызовов, связанных с сигналами, которые обсуждались выше. Некоторые из них предоставлены далее.

sigset(2)

Системный вызов sigset(2) имеет такие же параметры, как signal(2). Реакции, которые могут быть заданы для сигнала при помощи этой функции, таковы:

SIG_DFL - Установить реакцию на сигнал в реакцию по умолчанию. Как правило, эта реакция состоит в завершении программы.

SIG_IGN - Игнорировать сигнал.

SIG_HOLD - Задерживать сигналы при их прибытии. Ранее ожидавшие сигналы остаются задержанными.

func - При возникновении сигнала вызывается функция, на которую указывает func. Когда sigset вызывается с таким аргументом, задержанные сигналы освобождаются.

При установке обработчика через sigset(2), сигнал блокируется до возврата из обработчика. Но при вызове longjmp(3C) из обработчика, возврата из обработчика не происходит, и сигнал так и останется заблокированным. Для решения этой проблемы, совместно с sigset(2) необходимо использовать функции sigsetjmp(3C)/siglongjmp(3C), которые в нашем курсе подробно не изучаются.

sighold(2) После вызова sighold(2) вновь прибывающие сигналы задерживаются. Этот вызов аналогичен вызову sigset(2) с реакцией SIG_HOLD с тем отличием, что он не меняет старую реакцию на сигнал. Этот системный вызов предоставлен для удобства.

sigrelse(2) освобождает задержанные сигналы.

sigignore(2) Этот системный вызов эквивалентен вызову sigset(2) с реакцией SIG_IGN. Этот системный вызов предоставлен для удобства.

sigpause(2) Системный вызов sigpause(2) представляет собой атомарно исполняющуюся пару вызовов sigrelse(sig) и pause(2). Слово «атомарно», в данном случае, означает, что сигнал не может быть получен в интервале между sigrelse() и pause(). Если sigrelse() приведёт к доставке сигнала sig, процесс не войдёт в pause(). Системный вызов sigpause(2) представляет более элегантное решение проблемы потерянного пробуждения, чем рассматривавшаяся выше программа с применением setjmp/longjmp.

Задержка и освобождение сигнала - Образец

Этот образец кода показывает, как задерживать сигнал во время исполнения критической секции кода. Бывают ситуации, когда прибывший сигнал может влиять на правильность исполнения программы, если возникнет в неправильном месте кода. Одно решение состоит в игнорировании сигналов в этой критической секции, но при использовании такого метода сигналы могут быть потеряны.

Более удачный метод состоит в том, чтобы задержать сигналы до момента, когда программа будет готова прореагировать на сигнал. Вызов sighold(2) должен стоять перед входом в критический участок, а вызов sigrelse(2) должен быть сделан после выхода из него.

Файл: sighold1.c
ЗАДЕРЖКА И ОСВОБОЖДЕНИЕ СИГНАЛА - ОБРАЗЕЦ

```
 1  #include <signal.h>
 2
 3  main()
 4  {
 5     void (*istat)(int), sigcatch(int);
 6
 7    istat = sigset(SIGINT, sigcatch);
 8
 9    while(1) {
10     /*
11      * processing loop ...
12      */
13
14       sighold(SIGINT);
15       /*
16        * critical section of code ...
17        */
18       sigrelse(SIGINT);
19     }
20      sigset(SIGINT, istat);
21  }
22
23
24  void sigcatch(int sig)
25  {
26   /*
27    * signal catching routine here
28    */
29  }
```

Остановка до сигнала - Пример

Этот пример показывает, как останавливаться, ожидая прибытия сигнала, с использованием sigpause(2). В функции обработки сигнала внешняя переменная flag увеличивается на единицу. Перед проверкой flag вызывается sighold(2). Если flag равен нулю, не было перехвачено ни одного сигнала SIGINT, и программа засыпает в ожидании сигналов,

используя sigpause(2). Когда сигнал прибывает, программа просыпается и исполняет функцию обработки. Если сигнал прибудет до проверки flag, то sigpause() не будет вызвана. Наконец, переменная flag обнуляется, и задержанные сигналы освобождаются. Заметьте, что sigpause(2) автоматически освобождает заданный сигнал, если он прибыл, но был задержан.

Файл: sigpause1.c

ОСТАНОВКА ДО СИГНАЛА - ПРИМЕР

```
 1  #include <signal.h>
 2
 3  int flag;
 4
 5  main()
 6  {
 7     void sigcatch();
 8
 9     sigset(SIGINT, sigcatch);
10      sigset(SIGQUIT, sigcatch);
11     while (1) {
12      /*
13       * processing here
14       */
15        sleep(3);
16
17        sighold(SIGINT);
18        if (flag == 0) {
19           printf("pausing for SIGINT\n");
20           sigpause(SIGINT);
21        }
22        flag = 0;
23        sigrelse(SIGINT);
24     }
25  }
26
27
28  void sigcatch(int sig)
29  {
30     printf("entering sigcatch()\n");
31     if (sig == SIGQUIT)
32        exit(1);
33     flag++;
34  }
```

Маска сигналов процесса

Сигнальная маска - объект типа sigset_t, содержимое которого представляет собой множество (битовую маску) сигналов, поддерживаемых данной реализацией.

Если бит, соответствующий номеру сигнала, установлен в маске, считается, что сигнал принадлежит множеству. Напротив, если бит сброшен, то считается, что сигнал не принадлежит множеству.

Первоначально, маска представляла собой целое число, но в стандарте POSIX было предложено описать её как непрозрачный тип, манипуляции над которым осуществляются только при помощи предоставленных библиотекой функций. Это облегчает перенос программ, работающих с масками сигналов, на платформы, поддерживающие различное количество типов сигналов. В SVR4 это массив из 4 unsigned long.

Библиотечные функции (sigsetops(3C)) используются для манипуляции с отдельными битами в копии маски sigset_t, размещённой в пользовательском адресном пространстве. Использование новых системных вызовов обычно включает в себя создание копии маски, установку маски так, чтобы она отражала сигналы, которые должны быть блокированы или разблокированы, и вызов соответствующей системной функции изменения сигнальной маски ядра.

Руководство по sigsetops(2) содержит функции для манипуляций с объектами типа sigset_t.

sigemptyset(3C) Инициализирует маску сигналов, на которое указывает параметр set так, чтобы исключить все сигналы.

sigfillset(3C) Инициализирует маску сигналов, на которое указывает параметр set так, чтобы включить все сигналы.

sigaddset(3C) Добавляет индивидуальный сигнал, заданный значением signo, к маске, на которую указывает set.

sigdelset(3C) Удаляет индивидуальный сигнал, заданный значением signo, из маски, на которую указывает set.

sigismember(3C) Проверяет, установлен ли сигнал, заданный значением signo, в маске, на которую указывает set.

Замечание: Объект типа sigset_t должен быть инициализирован вызовом sigfillset, sigemptyset или системным вызовом до того, как применять к нему любые другие операции.

Манипуляции с сигнальной маской sigset_t - Пример

Эта программа использует функции sigsetops(3C) для манипуляций с битами в маске объекта sigset_t. Имейте в виду, эта программа не влияет на обработку сигналов

процессом. Она только работает с битами в пользовательской памяти.

6 Создать экземпляр объекта типа sigset_t.

9 Инициализировать переменную mask.

10-13 Установить в переменной mask биты, представляющие сигналы 1-3 и 15-17.

15-16 Распечатать четыре элемента массива unsigned long, которые образуют mask.

17-28 Для всех возможных сигналов выдать, установлен он или нет в mask, а также можно ли вообще маскировать его.

Вывод этой программы выглядит так:

```
$ sigmsk
00000340007
00000000000
00000000000
00000000000
Signal  0 ILLEGAL  Signal  1 Set      Signal  2 Set
Signal  3 Set      Signal  4 Not Set  Signal  5 Not Set
Signal  6 Not Set  Signal  7 Not Set  Signal  8 Not Set
Signal  9 Not Set  Signal 10 Not Set  Signal 11 Not Set
Signal 12 Not Set  Signal 13 Not Set  Signal 14 Not Set
Signal 15 Set      Signal 16 Set      Signal 17 Set
Signal 18 Not Set  Signal 19 Not Set  Signal 20 Not Set
Signal 21 Not Set  Signal 22 Not Set  Signal 23 Not Set
Signal 24 Not Set  Signal 25 Not Set  Signal 26 Not Set
Signal 27 Not Set  Signal 28 Not Set  Signal 29 Not Set
Signal 30 Not Set  Signal 31 Not Set  Signal 32 ILLEGAL
```

Файл: sigmsk.c

МАНИПУЛЯЦИИ С СИГНАЛЬНОЙ МАСКОЙ sigset_t - ПРИМЕР

```
1   #include <stdio.h>
2   #include <signal.h>
3
4   main()
5   {
6       sigset_t mask;
7       int i;
8
9       sigemptyset(&mask);
10       for( i = SIGHUP; i <= SIGQUIT; i++)
11          sigaddset(&mask, i);
12       for( i = SIGTERM; i <= SIGUSR2; i++)
13          sigaddset(&mask, i);
14
15       for( i = 0; i < 4; i++)
16          printf("%011o\n", mask.sigbits[i]);
17       for(i = 0; i <= NSIG; i++) {
18          switch( sigismember( &mask, i) ) {
19             case -1 :
20                printf("Signal %2d ILLEGAL  ", i);
21                break;
22             case 1 :
23                printf("Signal %2d Set      ", i);
24               break;
25             case 0 :
26                printf("Signal %2d Not Set  ", i);
27                break;
28          }
29          if( !((i + 1) % 3) ) putchar('\n');
30       }
31       putchar('\n');
32   }
```

Изменение или исследование маски сигналов процесса

Маска сигналов — это атрибут, хранящийся в пользовательской области процесса и представляющий собой множество заблокированных сигналов. Сигналы, упомянутые в маске, называются также замаскированными (masked). Такие сигналы не игнорируются, но и не доставляются процессу, до того момента, пока не будут разблокированы. В действительности, вызов sigset(2) с параметром SIG_HOLD и вызов sighold(2) реализованы как включение сигнала в маску, а sigrelse(2) — как исключение сигнала из маски.

Маска сигналов наследуется при fork(2) и exec(2).

Системный вызов sigprocmask(2) используется, чтобы изменить текущую маску сигналов процесса, определяющую сигналы, доставка которых в данный момент заблокирована. Кроме того, он может быть использован для исследования этой маски без произведения изменений в ней. Первый аргумент how принимает такие значения:

SIG_BLOCK - Множество сигналов, на которое указывает set, будет добавлено к текущей маске сигнала.

SIG_UNBLOCK - Множество set будет удалено из текущей маски.

SIG_SETMASK - Текущая маска будет заменена на set.

Если oset ненулевой, предыдущая сигнальная маска будет помещена в область памяти, на которую он указывает. sigprocmask(2) может быть использован для исследования текущей маски, если второй аргумент равен 0. В этом случае, how игнорируется, и текущая сигнальная маска вызывающего процесса остается без изменений.

sigpending(2)

Системный вызов sigpending(2) позволяет процессу посмотреть, какие сигналы были посланы, но в данный момент заблокированы сигнальной маской процесса.

Изменение сигнальной маски - Пример

Эта программа - вариант программы блокировки клавиатуры терминала, представленной в разделе «Терминальный ввод-вывод». В той версии, программа вызывала функции termios(2), чтобы выключить посылку сигналов, генерируемых терминалом, в терминальном драйвере. В этой альтернативной программе, вызовом sigprocmask(2) маска сигналов процесса устанавливается таким образом, чтобы единственным незаблокированным сигналом был сигнал разрыва линии SIGHUP.

11 Объявляется область памяти для сохранения копии сигнальной маски.

13-15 Все сигналы, кроме SIGHUP, блокируются

Восстанавливать маску сигналов не нужно, так как маска сигналов родительского процесса (например, shell) не менялась.

Файл: termlock.c

ИЗМЕНЕНИЕ СИГНАЛЬНОЙ МАСКИ - ПРИМЕР

```
 1  #include <stdio.h>
 2  #include <string.h>
 3  #include <signal.h>
 4  #include <sys/termios.h>
 5  #include <sys/types.h>
 6  main()          /* lock the terminal */
 7  {
 8     struct termios tty;
 9     tcflag_t savflags;
10      char key[BUFSIZ], *getkey(void);
11      sigset_t mask;
12
13      sigfillset(&mask);
14      sigdelset(&mask, SIGHUP);
15      sigprocmask(SIG_SETMASK, &mask, NULL);
16
17      tcgetattr(fileno(stdin),  &tty);
18      savflags = tty.c_lflag;
19      tty.c_lflag &= ~ECHO;
20      tcsetattr(fileno(stdin), TCSANOW, &tty);
21      strcpy(key, getkey());
22      for (;;)
23      if (strcmp(key, getkey()) == 0){
24         tty.c_lflag = savflags;
25         tcsetattr(fileno(stdin), TCSANOW, &tty);
26         break;
27      }
28  }
29
30  char *getkey(void)          /* prompt user for key */
31  {
32     static char line[BUFSIZ];
33
34     fputs("Key: ", stderr);
35     line[0] = '\377';        /* impossible char */
36     fgets(line, BUFSIZ, stdin);
37     fputs("\n", stderr);
38     return(line);
39   }
```

Новые методы управления сигналами

Системный вызов sigaction(2) дает ранее недоступный уровень управления сигналами. Кроме того, он совместим с требованиями стандарта POSIX. sigaction(2) предоставляет возможности, ранее реализованные в sigset(2): закрывает окно уязвимости при использовании signal(2) и автоматически блокирует рекурсивный вызов функции обработки сигнала.

Параметры act и oact, если они не равны нулю, указывают на экземпляры struct sigaction со следующими полями:

void (*sa_handler)(); - Адрес функции обработки сигнала, SIG_IGN или SIG_DFL

sigset_t sa_mask - Маска сигналов, которые должны быть заблокированы, когда вызывается функция обработки сигнала.

int sa_flags - Флаги, управляющие доставкой сигнала.

Если аргумент act ненулевой, он указывает на структуру, определяющую новые действия, которые должны быть предприняты при получении сигнала sig. Если аргумент oact ненулевой, он указывает на структуру, где сохраняются ранее установленные действия для этого сигнала.

Поле sa_flags в struct sigaction формируется побитовым ИЛИ следующих значений:

SA_ONSTACK - Используется для обработки сигналов на альтернативном сигнальном стеке.

SA_RESETHAND - Во время исполнения функции обработки сбрасывает реакцию на сигнал к SIG_DFL; обрабатываемый сигнал при этом не блокируется. Этот флаг воспроизводит поведение signal(2).

SA_NODEFER - Во время обработки сигнала сигнал не блокируется.

SA_RESTART - Системные вызовы, которые будут прерваны исполнением функции обработки, автоматически перезапускаются.

SA_SIGINFO - Используется для обработки надёжных сигналов и передачи обработчику дополнительных параметров. В этом курсе подробно не рассматривается.

Сигналы для управления заданиями

В ОС UNIX System V Версия 4 доступны сигналы, которые полезны в программах, реализующих интерактивное управление процессами первого плана (foreground) и фоновыми (background) процессами. Эти механизмы используются, главным образом, при реализации управления заданиями в командных процессорах. При помощи этих сигналов интерпретатор shell SVR4 позволяет пользователю делать такие операции, как приостановка процесса первого плана, перевод его в фоновый режим, или фонового процесса на первый план (foreground). Два сигнала, SIGSTOP и SIGTSTP вынуждают получивший процесс остановиться. SIGTSTP генерируется терминальным драйвером, при нажатии клавиши, заданной в c_cc[VSUSP] (<CTRL Z> по умолчанию) (см. termios(2), ioctl(2), <termios.h>). Посланный с терминала SIGTSTP заставляет процессы, входящие в основную (foreground) группу остановиться. SIGCONT возобновляет выполнение приостановленного процесса.

Если фоновый процесс не задерживает и не игнорирует SIGTTIN, система будет посылать SIGTTIN этому процессу при попытках выполнить read(2) с управляющего терминала. Если процесс игнорирует или задерживает SIGTTIN, read(2) возвращает неуспех, и устанавливает errno в EIO, (в предыдущих версий ОС UNIX System V, read(2) с терминала из фонового процесса немедленно возвращал управление с кодом 0, не прочитав ни одного байта).

В ранних версиях стандартный вывод фонового процесса просто шел на терминал, если не был перенаправлен. А в SVR4 фоновый процесс при попытке выдачи на терминал, если установлен бит TOSTOP в поле c_lflag (см. termios(2), ioctl(2), <termios.h>), получает сигнал SIGTTOU. Обычно, это вынуждает фоновый процесс приостановиться. Однако, если процесс игнорирует или задерживает SIGTTOU, выдача на терминал все-таки происходит.

Командный процессор bash(1) под SVR4 традиционно собирают с игнорированием SIGTTOU. Чтобы наблюдать остановку фоновых процессов при попытке вывода на экран, необходимо использовать стандартные командные оболочки SVR4, поддерживающие управление заданиями — ksh(1) или jsh(1).

Управление заданиями - Пример

Эта программа создаёт подпроцесс, который пишет числа на экран терминала. Родительский процесс позволяет порождённому писать в течение 5 секунд, останавливает его на 10 секунд, а затем перезапускает, позволяя ему исполняться в течение еще 10 секунд, пока не пошлёт ему SIGTERM.

Программа, исполняемая порожденным процессом:

```
main()
{
        int i = 0;
        while(1) {
        printf("%d\n", i++);
        sleep(1);
        }
}
```

Файл: job_cont.c

УПРАВЛЕНИЕ ЗАДАНИЯМИ - ПРИМЕР

```
 1   #include <sys/types.h>
 2   #include <signal.h>
 3   #include <sys/procset.h>
 4
 5   main()
 6   {
 7       pid_t pid;
 8       if ((pid = fork()) == 0) {
 9           execl("./forever", "forever", 0);
10           fprintf("execl failed\n");
11           exit(1);
12       }
13       else {
14           sleep(5);
15           sigsend(P_PID, pid, SIGSTOP);
16           sleep(10);
17           sigsend(P_PID, pid, SIGCONT);
18           sleep(10);
19           sigsend(P_PID, pid, SIGTERM);
20           wait(0);
21       }
22   }
23
```

9. УПРАВЛЕНИЕ ТЕРМИНАЛЬНЫМ ВВОДОМ/ВЫВОДОМ

Обзор

Этот раздел обсуждает основы интерфейса для управления асинхронными коммуникационными портами (терминальными портами). Функции, перечисленные на странице руководства TERMIOS(2) используются для доступа и конфигурации аппаратного интерфейса с терминалом. Эти функции и их аргументы будут обсуждаться в этом разделе. Первая секция этого раздела предоставляет информацию, необходимую для понимания характеристик терминала и принципов работы аппаратного и программного терминального интерфейса. Затем будут обсуждаться некоторые аспекты программного интерфейса с терминалом. Приводятся примеры использования функций termios(2) для изменения этих установок.

Характеристики терминального интерфейса

Приблизительно до конца 80х-начала 90х, терминалы были основным средством организации взаимодействия человека с компьютером. Терминал (дословно — оконечное устройство) представляет собой электронную пишущую машинку (телетайп) или устройство, состоящее из клавиатуры и дисплея (видеотерминал). Оба типа терминалов соединены с компьютером последовательным портом (обычно, RS232 или токовая петля); при этом символы, вводимые с клавиатуры, передаются компьютеру, а данные, передаваемые компьютером, показываются на дисплее (в случае видеотерминала) или печатаются на бумаге (в случае телетайпа). Как телетайпы, так и видеотерминалы предназначены для ввода и отображения текстовой информации. С точки зрения компьютера, терминальный порт представляет собой двунаправленный (полнодуплексный) последовательный порт, по которому производится обмен символами кодировки ASCII или национальной кодировки, такой, как КОИ8.

Кроме ASCII, большинство видеотерминалов могут передавать и принимать коды расширения (escape sequence). Обычно это многобайтовые коды, начинающиеся с символа '\0x1B' (ASCII ESC), обозначающие нажатия клавиш, для которых нет соответствующих кодов в ASCII (стрелки, «функциональные» клавиши и т. д.), а также команды терминалу: передвижение курсора, изменения цвета текста и т.д..

Так, на многих видеотерминалах, последовательность символов "\0x1B[A" обозначает нажатие клавиши «стрелка вверх» на клавиатуре, а также команду на перемещение курсора на одну строку вверх.

Поскольку терминалы были основным средством взаимодействия человека с компьютером, в системах семейства Unix в драйвер терминала был встроен ряд функций, не сводящихся к простой передаче данных через порт. Для управления всеми этими функциями, терминальные устройства поддерживали набор специальных команд ioctl(2). Среди этих функций следует упомянуть:

Редактирование ввода: стирание последнего введенного символа, последнего слова и всей строки.

Преобразование ввода: преобразование символов конца строки, замена табуляций на пробелы, и др.

Генерация сигналов: при вводе определенных символов, ядро посылает группе процессов первого плана сигналы.

Поддержка терминальных сессий и управления заданиями.

Было разработано множество программ, рассчитанных на работу с терминалами: экранные текстовые редакторы, интегрированные среды разработки, почтовые клиенты, клиенты gopher, веб-браузеры (lynx и links), файловые менеджеры, игры и др. Командные процессоры с управлением заданиями (ksh(1), jsh(1), bash(1)) использовали поддержку со стороны терминала (фоновые группы и группы первого плана, а также сигналы управления заданиями). Кроме того, многие программы, такие как su, sudo, login, использовали некоторые простые терминальные функции, такие, как включение и выключение «эхо» (отображения вводимых пользователем символов). Действительно, при наборе команд, пользователю необходимо видеть на экране набираемые им символы, а при вводе пароля это может быть нежелательно. Поэтому утилиты, требующие ввода пароля, выключают отображение ввода, а потом включают его обратно.

Интерфейс ввода/вывода

Диаграмма показывает отношения между пользовательской программой, ядром системы и портом ввода/вывода при работе с физическим терминалом.

Физические терминалы присоединяются к компьютеру через аппаратный интерфейс ввода/вывода, называемый портом. Порты, сделанные различными изготовителями, устроены по разному, но поддерживают стандартные протоколы обмена, обычно стандарт RS232. В IBM PC-совместимых компьютерах терминальные порты RS232 называются COM-порты. Эти порты обычно конфигурируются или программируются, так что они могут работать с различными терминалами и на различных скоростях.

Пользовательские программы не имеют прямого доступа к портам ввода/вывода. Программный интерфейс с портами предоставляется драйвером устройства. Драйвер устройства — это модуль ядра Unix, который предоставляет набор аппаратно-зависимых функций, которые, в свою очередь, управляют портами и осуществляют доступ к ним. Доступ к драйверу происходит через специальный байт-ориентированный файл терминала. Имена этих файлов находятся в директории, обычно /dev или /dev/term, и доступ к ним может осуществляться так же, как к обычным файлам. Ввод и вывод на терминал осуществляется чтением и записью в соответствующий специальный файл. Многие программы не видят разницы между терминалом и обычным файлом, что позволяет перенаправлять ввод-вывод таких программ на терминал или в файл в зависимости от потребностей пользователя.

В Unix SVR4, терминальные драйверы сами имеют модульную структуру и состоят из нескольких модулей STREAMS, STREAMS — это появившийся в Unix SVR3 асинхронный интерфейс для драйверов последовательных (байт-ориентированных) устройств. API для разработки драйверов в Solaris описано в секции руководства 9 и не будет подробно обсуждаться в этом курсе.

Терминальный драйвер STREAMS состоит из, как минимум, двух модулей: собственно драйвера порта, который обеспечивает прием и передачу данных в физический порт, и модуля терминальной дисциплины ldterm(7m), который и отвечает за специфически терминальные функции и обработку терминальных команд ioctl(2).

Псевдотерминалы

При переходе к графическим дисплеям, возник вопрос, как обеспечить совместимость с программами, ориентированными на работу с терминалом. Для этого в ядре Unix предусмотрен интерфейс для создания специальных псевдоустройств, которые так и называются псевдотерминалами. Псевдоустройство или виртуальное устройство — это виртуальный объект, обслуживаемый специальным драйвером. Такой драйвер поддерживает такие же программные интерфейсы и структуры данных (минорную запись, специальный файл в каталоге /dev), как и обычный драйвер, но, в отличие от обычного драйвера, драйвер псевдоустройства не связан ни с каким физическим устройством, а имитирует все функции устройства программно. Примерами псевдоустройств являются файлы /dev/null, /dev/zero, /dev/random (в Linux). Псевдотерминалы создаются для поддержки сессий удаленного доступа через telnet(1), rlogin/rsh(1) и ssh(1), а также для терминальных сессий xterm(1) и gnome-terminal(1).

Псевдотерминал состоит из двух псевдоустройств, ведущего (/dev/pts/XX) и ведомого (/dev/pty/XX, где X — десятичная цифра). Процедура создания и открытия этих устройств различается в разных Unix-системах и подробно не рассматривается в этом курсе. Соответствующая процедура для Solaris описана на странице руководства pts(7D). Оба устройства, в действительности, обслуживаются одним и тем же модулем ядра и представляют собой нечто вроде трубы: данные, записываемые в дескриптор ведущего устройства, читаются из ведомого, и наоборот. Кроме того, ведомое устройство поддерживает все команды ioctl(2), обязательные для терминала, и все терминальные функции, перечисленные на предыдущей странице.

Рассмотрим использование псевдотерминала терминальным эмулятором xterm(1). xterm(1) представляет собой графическое приложение, использующее протокол X Window. При запуске, xterm(1) создает и открывает окно на локальном или удаленном дисплее, имя которого определяется переменной среды DISPLAY. Это окно представляет собой текстовое окно, по умолчанию имеющее размер 80x25 символов, что соответствует стандартному размеру экрана большинства видеотерминалов. Кроме того, xterm(1) создает пару ведущего и ведомого устройств псевдотерминала, запускает подпроцесс, в этом подпроцессе создает сессию вызовом setsid(2), открывает ведомое устройство на дескрипторы 0, 1 и 2 (при этом, соответствующий псевдотерминал становится управляющим терминалом этой сессии), устанавливает переменную среды TERM=xterm и запускает в этом подпроцессе программу. По умолчанию, имя программы берется из переменной среды SHELL, но, если указать соответствующие параметры xterm(1), можно запустить произвольную программу.

Затем, xterm(1) преобразует нажимаемые пользователем клавиши в коды ASCII или текущей национальной кодировки (в наше время, обычно, UTF-8) или, если это необходимо, в коды расширения, и передает эти символы в ведущее устройство, так что они поступают на стандартный ввод запущенной программы или какого-то из её подпроцессов.. Кроме того, данные, выводимые запущенной программой в дескрипторы 1 и 2, считываются процессом xterm(1) из ведущего устройства и отображаются в текстовом окне так же, как они отображались бы на дисплее видеотерминала. При этом, передаваемые программой коды расширения интерпретируются программой xterm(1) как команды перемещения курсора, изменения цвета текста и т. д., поэтому программы, рассчитанные на работу с видеотерминалом, работают в привычной для них среде.

Программный интерфейс ввода/вывода

Многие из системных вызовов для работы с обычными файлами также используются и для работы с терминальными специальными файлами. Для доступа к терминалам можно использовать следующие системные вызовы:

open(2) Как и регулярные файлы, специальные байт-ориентированные файлы открываются этим системным вызовом. По соглашению, имена всех терминальный файлов находятся в директории /dev или одной из поддиректорий /dev. В Solaris они размещены в /dev/term/XX (физические терминалы) и /dev/pty/XX (псевдотерминалы), где XX — двузначное десятичное число. Кроме того, управляющий терминал вашей сессии доступен вашей программе как /dev/tty.

ioctl(2) Этот системный вызов используется передачи устройствам команд, которые не могут быть сведены к чтению или записи. У терминалов, ioctl(2) используется как для конфигурации физического порта ввода/вывода, так и для управления функциями терминальной дисциплины. Соответствующие команды ioctl(2) не стандартизованы, различаются в разных Unix-системах и не будут обсуждаться в этом курсе. Параметры ioctl(2) для работы с терминалами в Solaris, описаны на странице руководства termio(7I).

termios(3C) Эта страница руководства содержит набор функций, предоставляющих стандартизованный интерфейс для управления терминальными устройствами. Это более предпочтительный интерфейс, чем ioctl(2), потому что он соответствует стандарту POSIX и обеспечивает разработку переносимых программ. В этом разделе будут обсуждаться, главным образом, функции termios(3C).

isatty(3F) Этот системный вызов определяет, связан ли файловый дескриптор с терминальным устройством или с файлом какого-то другого типа. Если isatty(3F) возвращает ненулевое значение, файловый дескриптор поддерживает терминальные ioctl(2) и функции termios(3C).

read(2) Используется для чтения данных из специального терминального файла. read(2) возвращает количество прочитанных байтов, которое может быть меньше запрошенного. По умолчанию, терминал ожидает ввода полной строки, оканчивающейся символом '\n' (ASCII NL) и считывает данные по строкам. Однако не обязательно читать всю строку за один раз. Если буфер read(2) меньше длины текущей строки, будет считано только начало строки.

Чтение с терминала разрушает данные, то есть прочитанные данные не могут быть прочитаны опять. Поэтому если два процесса одновременно читают с терминала, это может приводить к потере данных. Для управления доступом к чтению с терминала используются сигналы управления заданиями и функция tcsetpgrp(3C), которые рассматриваются далее в этом разделе

write(2) Системный вызов write(2) используется для записи символов в специальный байт-ориентированный файл.

poll(2) и select(3C). Эти вызовы часто используются для мультиплексирования ввода-вывода, если процессу необходимо одновременно работать с терминалом и другими устройствами или псевдоустройствами, работа с которыми может привести к блокировке.

libcurses(3LIB) библиотека для генерации кодов расширения терминала в зависимости от его типа.

close(2) Системный вызов close(2) закрывает дескриптор файла, связанный со специальным файлом.

lseek(2), mmap(2) Терминальные устройства эти вызовы не поддерживают.

Библиотека libcurses(3LIB) и другие библиотеки

Видеотерминалы и терминальные эмуляторы поддерживают довольно богатые функции работы с текстом. Одной из главных функций является изменение положения курсора — точки, в которой будет выводиться текст на экран. Перемещая курсор, пользовательская программа может рисовать на экране почти всё, что можно изобразить при помощи символов ASCII: экранные формы, выпадающие меню, диалоговые окна, даже изображения (так называемый ASCII art). Кроме того, многие терминалы поддерживают расширенные символы, например, так называемую «псевдографику». Эти символы позволяю рисовать на экране прямоугольные рамки и таблицы. Многие видеотерминалы и практически все терминальные эмуляторы также поддерживают управление цветом символов и фона.

Доступ ко всем этим функциям осуществляется при помощи кодов расширения (escape sequences) — многобайтовых последовательностей, начинающихся с символа ASCII ESC. Разные модели терминалов поддерживают разные наборы кодов расширения. Существует несколько де-юре и де-факто стандартов этих кодов. Основной стандарт де-юре — это ECMA-48, известный также как ANSI X3.64 и ISO/IEC 6429, соответствующий набор кодов расширения также называется ANSI escape sequences. Наиболее известные стандарты де-факто — это управляющие коды терминалов DEC VT-50, DEC VT-100 и xterm. Набор кодов расширения VT-100 сильно перекрывается со стандартом ANSI и считается одним из первых аппаратных терминалов, реализовавших этот стандарт. Набор кодов xterm эмулирует как VT-100, так и ANSI, а также добавляет ряд функций, характерных для терминального эмулятора — так, код расширения ESC]2;stringBEL (где ESC — это ASCII ESC, а BEL — ASCII BEL) заменяет название окна xterm на string. В системном руководстве bash(1) содержится информация о том, как настроить строку приглашения bash, чтобы она содержала эту последовательность и формировала string так, чтобы эта строка, в свою очередь, содержала интересную для пользователя информацию, например, текущий каталог данной терминальной сессии, имя сетевого узла и т. д.

Так или иначе, многие терминалы, реализующие стандартные наборы команд, поддерживают их не полностью или добавляют какие-то свои расширения. Ядро Unix не предоставляет сервисов для работы с кодами расширения терминалов и передает эти коды терминалу «как есть». Для работы с кодами расширения необходимо использовать библиотеки, например, libcurses(3LIB), которая входит в поставку Solaris.

Библиотека libcurses обеспечивает формирование кодов расширения для выполнения заданных функций, например, для перемещения курсора в заданную точку экрана, а также интерпретацию кодов расширения, посылаемых терминалом, и их перевод в независимые от терминала псевдосимволы. Библиотека определяет тип терминала на основе переменной среды TERM, и использует базу данных, хранящуюся в каталоге /usr/share/lib/terminfo/. В этой базе для каждого известного типа терминала хранится таблица, описывающая поддерживаемые им типы команд и код расширения, соответствующий каждой команда. Libcurses обеспечивает некоторую оптимизацию: так, если требуется переместить курсор в заданную точку и терминал поддерживает команды для установки позиции курсора, библиотека сгенерирует соответствующую команду. Если же терминал такой команды не поддерживает, библиотека сгенерирует последовательность перемещений курсора на одну позицию вверх, вниз или вбок (такая функция есть почти у всех видеотерминалов).

Почти все «полноэкранные» программы, такие, как текстовые редакторы, веб-браузеры или файловые менеджеры, используют libcurses или какой-то из ее аналогов, чаще всего ncurses. Однако в нашем курсе мы изучать эту библиотеку не будем.

Канонический ввод

Терминал имеет две очереди ввода и одна - вывода. Символы, вводимые с клавиатуры терминала, помещаются в "сырую" очередь ввода. Кроме того, если требуется эхо (вывод печатаемых символов на экран), копии этих символов добавляются в очередь вывода.

Если разрешен канонический режим обработки ввода (обычно это делается по умолчанию), символы из "сырой" очереди подвергаются предобработке при копировании в каноническую очередь ввода. Копирование происходит по строкам, когда поступает символ перевода строки (NL). Пользовательская программа читает строки ввода из канонической очереди.

Каноническая предобработка ввода включает обработку символов забоя и стирания строки. Символ забоя ERASE (на видеотерминалах это обычно '\0x8' (Ctrl-H, ASCII BS) или '\0x7F' (ASCII DEL); на телетайпах часто использовался символ #) удаляется вместе с символом, введенным перед ним.

Клавиша «Backspace» на терминале может быть настроена посылать либо ASCII BS, либо ASCII DEL. Клавиша «Del» в xterm или gnome-terminal посылает последовательность ASCII ESC [3~. У gnome-terminal поведение обоих клавиш настраивается на закладке настроек Edit->Profile Preferences->Compatibility.

Символ стирания строки KILL (Ctrl-W по умолчанию) приводит к стиранию всей текущей строки.

Например, предполагая, что символ забоя равен ASCII BS, вы набираете на клавиатуре:

datx*BS*e
Символы, набранные на клавиатуре, записываются в "сырую" очередь по мере ввода. Затем при копировании из "сырой" очереди в каноническую, символы просматриваются. При этом символы BS и стоящий перед ним выбрасываются. Поэтому, программа, читающая с терминала, получит строку:

date
При выводе, символы, генерируемые вашей прикладной программой, накапливаются в выходной очереди. При этом может происходить требуемая настройками постобработка.

Использование termios(3C)

Существует более пятидесяти различных параметров и флагов, управляющих терминальным интерфейсом. Эти параметры можно изменять через ioctl termio(7I), функции termios(3C) и команду stty(1). Ниже приводятся некоторые из основных характеристик терминала.

Параметры RS232

Используя флаг c_cflag, вы можете управлять скоростью передачи, количеством бит в символе, количеством стоповых битов, обработкой бита четности и т.д. Это необходимо потому, что протокол RS232 не имеет средств автосогласования указанных параметров, и их необходимо настраивать вручную так, чтобы настройки терминала и терминального порта компьютера совпадали. У псевдотерминалов эти параметры сохраняются только для совместимости; изменение большинства из них (кроме количества бит в символе) ни на что не влияют.

Отображение символов.

Вы можете управлять обработкой символов возврата каретки (CR) и перевода строки (NL). Вы можете преобразовывать NL в CR (INLCR) и CR в NL (ICRNL). Вы можете также игнорировать поступающие символы CR (IGNCR). Это полезно для терминалов, которые генерируют последовательность символов CR-NL при нажатии клавиши <RETURN>. На других терминалах, которые генерируют одиночный символ CR, лучше

принимать этот символ и преобразовывать его в NL. Флаги для этих режимов помещаются в c_iflag. Соответственно, флаги ONLCR и OCRNL в c_oflag используются для преобразования NL в CR-NL или CR в NL, соответственно, при выводе.

Для терминалов, которые отображают только буквы верхнего регистра, могут быть установлены флаги XCASE в c_iflag, IUCLC в c_iflag и OLCUC в c_oflag. Буквы верхнего регистра отображаются в нижний регистр при вводе и наоборот - при выводе. Буквы верхнего регистра предваряются символом обратной косой черты (backslash, \).

По умолчанию, некоторые терминалы используют семибитный код ASCII для представления символов. Однако может быть необходима работа с восьмибитными данными. Например, это может потребоваться программам, работающим с национальными алфавитами или UTF-8. Для того, чтобы выключить срезание восьмого бита при вводе, вы должны очистить флаг ISTRIP в

c_iflag, установить флаг CS8 для передачи восьмибитных символов и, возможно, выключить контроль четности очисткой PARENB в c_cflag.

Задержки и табуляции.

Для механических терминалов можно установить различные флаги в c_oflag, управляющие задержками при обработке перевода строки, возврата каретки, горизонтальной табуляции, сдвига каретки назад (backspace), вертикальной табуляции и перевода страницы. Горизонтальная табуляция может преобразовываться в соответствующее число пробелов установкой флага TAB3.

(Продолжение на следующей странице)

Управление потоком.

Вывод на терминал может быть приостановлен нажатием СТОП-символа, по умолчанию CTRL-S, и возобновлен нажатием СТАРТ-символа, по умолчанию CTRL-Q. Эта возможность разрешается установкой флага IXON в c_flag. Иначе, эти символы не имеют специального смысла. Кроме того, если установить флаг IXANY, то любой символ будет возобновлять вывод. Для управления потоком вывода используется функция tcflow(2). Кроме управления потоком вывода, установкой флага IXOFF в c_iflag можно управлять потоком ввода. При этом, если входная очередь приближается к заполнению, драйвер терминального устройства пошлет

СТОП-символ для приостановки ввода, и СТАРТ-символ - для его возобновления. Это может быть полезно, если прикладная программа получает данные из удаленной системы.

Более удобный способ управлять потоком данных при выводе на терминал — это утилита more(1) или её аналоги. Однако надо подчеркнуть, что при использовании more(1), программа осуществляет вывод не на терминал, а в программный канал, направленный на вход more(1); это может повлиять на поведение некоторых программ.

Управляющие символы

При вводе некоторые символы имеют специальное значение. Например, # или ASCII BS

является символом забоя, CTRL-W — символом стирания строки, CTRL-D — концом ввода и т.д. Эти символы хранятся в массиве c_cc[] и могут быть изменены. Например, при использовании дисплея, гораздо удобнее использовать в качестве символа забоя BS (сдвиг каретки назад), чем #

Эхо

Терминалы обычно работают с UNIX-системами в полнодуплексном режиме. Это означает, что данные передаются в обоих направлениях одновременно и что компьютер обеспечивает эхо (отображение на экране или на печати) получаемых символов. Эхо выключается очисткой флага ECHO в c_lflag. Кроме того, стертые нажатием клавиши забоя символы могут затираться установкой флага ECHOE. Если установить флаг ECHOK, то стирание строки символом KILL будет приводить к переводу строки на терминале.

Немедленный ввод

Обычно символы при вводе накапливаются, пока не соберется полная строка, завершенная NL. Только после этого удовлетворяется запрос read(2), даже если он требовал только один символ. Значение, возвращаемое вызовом read(2), равно количеству прочитанных в действительности символов. Во многих прикладных программах, таких как редакторы форм или полноэкранные текстовые редакторы, строки ввода не имеют смысла. В таких программах необходимо читать символы по мере их ввода. При очистке флага ICANON в c_lflag вводимые символы не группируются в строки, и read(2) читает их по мере поступления. Этот режим известен также как режим неканонического ввода. Вместо этого удовлетворение запроса read(2) определяется параметрами MIN (минимальное количество нажатых клавиш) и TIME (промежуток времени

между введенными символами). Если ICANON установлен, то режим ввода называется каноническим.

(Продолжение на следующей странице)

"Сырой" терминальный ввод/вывод

Терминальные порты могут использоваться для подключения не только терминалов, но и других устройств, например, модемов, мышей или различной контрольно-измерительной аппаратуры. При работе с такими устройствами, прикладные программы и драйверы терминальных устройств могут быть вынуждены передавать и принимать произвольные восьмибитные данные. Это могут быть сообщения мыши о передвижении и нажатии кнопок, данные протокола PPP или поступающие с измерительного устройства данные. Для разрешения этого необходимо установить восьмибитные данные, неканонический ввод, запретить все отображения символов и управление потоком, устранить специальные значения всех управляющих символов и выключить эхо. В конце раздела приводится пример использования "сырого" терминального ввода/вывода.

Получение и установка атрибутов терминала

Первые две функции, описанные на странице Руководства termios(3C), используются для получения и установки атрибутов терминала. tcgetattr(3C) получает текущие установки терминального устройства. tcsetattr(3C) изменяет эти установки. Эти функции получают и передают требуемые параметры в виде управляющей структуры termios.

Параметры функций tcgetattr(3C) и tcsetattr(3C) таковы:

fildes дескриптор файла, соответствующий терминальному устройству. Для этого дескриптора вызов isatty(3F) должен возвращать ненулевое значение.

optional_actions Комбинация флагов, определенных в <termios.h>. optional_actions определяет, когда должны быть выполнены изменения и что делать с находящимися в буферах устройства данными при изменении параметров.

termios_p указатель на структуру termios. Эта структура содержит флаги и битовые поля, используемые для управления терминальным интерфейсом ввода/вывода. Флаги и поля этой структуры обсуждаются позже.

После исполнения tcgetattr(3C) рекомендуется сохранить копию этой структуры, чтобы программа могла вернуть начальное состояние терминального интерфейса. Дело в том, что функции tcsetattr(3C) изменяют настройки не файлового дескриптора вашего процесса, а настройки драйвера в ядре Unix. Внесенные вами изменения не откатываются автоматически при завершении программы, поэтому ненормально завершившаяся программа может оставить терминал в непригодном для работы состоянии.

В некоторых случаях, для восстановления параметров терминала можно воспользоваться командой stty(1). В Solaris, команда stty sane пытается привести терминал в режим, пригодный для интерактивной работы.

При изменении настроек терминала, вместо того, чтобы формировать значения полей структуры termios самостоятельно, рекомендуется получить текущие настройки терминала вызовом tcgetattr(3C), затем поменять нужные вам параметры в структуре и установить новые параметры вызовом tcsetattr(3C). При этом вы, по возможности, сохраните те настройки, которые пользователь мог сделать до запуска вашей программы. Кроме того, в последующих версиях Solaris и в других Unix-системах, в структуре termios могут появиться дополнительные поля или флаги. Если ваша программа не будет без нужды изменять незнакомые ей настройки, это значительно облегчит её адаптацию к новым версиям Solaris и ее перенос на другие платформы.

Параметр optinal_actions функции tcsetattr(2)

Параметр optional_actions функции tcsetattr(2) может принимать следующие значения:

TCSANOW Атрибуты изменяются немедленно.

TCSADRAIN Изменения атрибутов происходят только после того, как был передан («осушен») весь вывод в fildes. Этот запрос может быть использован при изменении атрибутов, которые влияют на обработку вывода.

TCSAFLUSH Это похоже на TCADRAIN. Изменение происходит после того, как весь вывод в fildes был передан, а непрочитанный ввод сброшен. Например, этот запрос полезен, если нужно проигнорировать все символы в буфере ввода.

Структура termios

Структура termios используется для представления характеристик терминального устройства. Структура одна и та же для всех терминальных устройств, независимо от изготовителя аппаратуры. Это предоставляет единообразный способ изменения характеристик и поведения терминального устройства. В частности, эта структура используется модулем STREAMS для изменения поведения аппаратного и программного интерфейса ввода/вывода.

Ниже перечислены поля структуры termios:

c_iflag флаги, управляющие предобработкой ввода с терминала.

c_oflag флаги, управляющие системной постобработкой вывода на терминал.

c_cflag флаги, описывающие аппаратные характеристики терминального интерфейса.

c_lflag флаги, управляющие разбиением потока на строки.

c_cc[] массив специальных управляющих символов.

Эти поля будут подробнее обсуждаться далее в этом разделе.

Ссылка: /usr/include/sys/termios.h

Управляющие символы

Управляющие символы, определенные в массиве c_cc[] имеют специальное значение и могут быть изменены вызовом tcsetattr(). Далее в этом разделе, приведены символьные константы, определенные в <termios.h>. Эти символьные константы могут быть использованы как индексы в массиве c_cc[]. Значения по умолчанию для соответствующих элементов перечислены в скобках. Некоторые из этих управляющих символов описаны ниже.

Следует иметь в виду, что в кодировке ASCII, первые 32 символа зарезервированы для выполнения различных управляющих функций (см. ascii(5)). Видеотерминалы и терминальные эмуляторы генерируют такие символы при одновременном нажании комбинации алфавитных (или некоторых неалфавитных) клавиш и клавиши Ctrl. Нажатие клавиши Ctrl приводит к срезанию старших бит кода соответствующего символа. Исключение составляет комбинация Ctrl-?, которая выдает код '\0x1F' (ASCII DEL). Так, комбинация клавиш Ctrl-D — это символ '\0x3' (ASCII EOT).

VINTR (Ctrl-C или ASCII DEL) генерирует сигнал SIGINT, который посылается всем процессам в группе первого плана, связанной с этим терминалом. По умолчанию процесс при получении этого сигнала будет завершен, но он может проигнорировать этот сигнал или перехватить его при помощи функции обработки.

VQUIT (CTRL-\) генерирует сигнал SIGQUIT. Этот сигнал обрабатывается так же, как и SIGINT.

VERASE (Ctrl-H, Ctrl-? или #) стирает предыдущий символ. Он не может стереть символ перед началом строки, ограниченной символами NL, EOF, EOL или EOL2.

VWERASE (CTRL-W) очищает предыдущее "слово". Он не может стереть слово из предыдущей строки, ограниченной символами NL, EOF, EOL или EOL2. Это функция расширения, поэтому для ее использования необходимо установить флаг IEXTEN.

VKILL (Ctrl-U) стирает всю строку, ограниченную символами NL, EOF, EOL или EOL2.

VEOF (CTRL-D) может использоваться для обозначения конца файла при вводе с терминала. Когда получен этот символ, все символы, ожидающие считывания, будут немедленно переданы программе, без ожидания символа новой строки, и остаток строки игнорируется. Таким образом, если в очереди не было символов, то есть EOF был послан в начале строки, read получит ноль символов, что является стандартным обозначением конца файла.

(Продолжение на следующей странице)

VSTOP (CTRL-S) может использоваться для временной приостановки вывода. Это полезно на экранных терминалах, чтобы вывод не исчезал с экрана, пока пользователь его не прочитал. Если вывод уже приостановлен, вводимые СТОП-символы игнорируются и не будут прочитаны.

VSTART (CTRL-Q) используется для возобновления вывода, остановленного СТОП-символом. Если ввод не был приостановлен, символы VSTART игнорируются и не читаются.

VSUSP (CTRL-Z) генерирует сигнал SIGTSTP, который приостанавливает все процессы в группе процессов первого плана этого терминала. Например, этот символ используется для функций управления заданиями в shell.

VDISCARD (CTRL-O) приводит к тому, что весь вывод будет игнорироваться, пока не будет послан еще один символ DISCARD, программа не выведет новые символы или не сбросит соответствующее условие. Это функция расширения, и исполняется, только если установлен флаг IEXTEN.

VLNEXT (CTRL-V) игнорирует специальное значение следующего символа. Это работает для всех специальных символов из массива c_cc[]. Это позволяет вводить символы, которые в ином случае были бы проинтерпретированы системой (такие, как KILL, QUIT). Символы VERASE, VKILL и VEOF могут также быть введены после символа обратной косой черты (backslash, \). В этом случае они также не вызовут исполнения специальной функции.

VREPRINT (CTRL-R или ASCII DC2) печатает символ новой строки и все символы, которые ожидали в очереди ввода (как если бы это была новая строка). Это считается функцией расширения, поэтому работает, только если установлен IEXTEN.

Для изменения управляющего символа, необходимо получить текущие терминальные атрибуты вызовом tcgetattr(3C), присвоить требуемому элементу массива c_cc[] новое значение и изменить атрибуты терминала вызовом tcsetattr(3C). Если значение управляющего символа будет _POSIX_VDISABLE, то функция, ассоциированная с этим символом, будет выключена.

Некоторые флаги режимов

Следующая страница перечисляет некоторые атрибуты терминала, которые могут быть изменены. Флаги, перечисленные во второй колонке таблицы, являются символьными константами, определенными в <sys/termios.h>, и представляют собой значения отдельных битов. Значения флагов хранятся в следующих четырех полях структуры termios:

c_iflag Поле c_iflag описывает режим обработки ввода. Если установлен флаг IGNBRK, то последовательность нулевых бит (break condition, некоторые терминалы или модемы таким образом кодируют разрыв линии) игнорируется, то есть не помещается в очередь ввода и не может быть считано ни одним процессом. Иначе, если установлен флаг BRKINT, условие разрыва генерирует сигнал прерывания и сбрасывает входную и выходную очереди.

Если установлен ISTRIP, то вводимые символы обрезаются до 7 бит, иначе они передаются как 8-битные значения. Если установлен ICRNL, то символ CR переводится в символ NL.

Если установлен IXON, разрешается старт/стоповое управление выводом. Получение СТОП-символа будет задерживать вывод, а СТАРТ-символ - возобновляет его. Все СТАРТ/СТОП-символы игнорируются и не читаются. Если установлен IXANY, любой введенный символ будет возобновлять приостановленный вывод.

c_oflag Поле c_oflag содержит флаги, управляющие обработкой вывода. Если установлен флаг OPOST, выводимые символы подвергаются постобработке в соответствии с остальными флагами, иначе они передаются без изменений.

Если установлен ONLCR, символ NL передается как пара CR-NL. TAB3 и XTABS задают замену символов табуляции пробелами.

c_cflag Поле c_cflag управляет аппаратными атрибутами терминального интерфейса. Биты CBAUD задают скорость передачи. Биты CSIZE задают размер символа в битах как для приема, так и для передачи.

Если CSTOPB установлен, передаются два стоповых бита. Флаги PARENB и PARODD управляют контролем четности.

c_lflag Если установлен ICANON, разрешена каноническая обработка ввода. Допускаются функции редактирования (забой и стирание строки) и объединение вводимых символов в строки, ограниченные символами NL, EOF, EOL, EOL2. Если ICANON не установлен, данные для удовлетворения запросов чтения берутся прямо из "сырой" очереди. Неканоническая обработка будет обсуждаться далее.

Если установлен ECHO, на каждый полученный символ выдается эхо. Если установлен режим ICANON, доступен ряд функций управления эхо. Если установлены флаги ECHO и ECHOE, а ECHOPRT не установлен, эхо для символа забоя выдается как ASCII BS SP BS (сдвиг каретки назад - пробел - сдвиг каретки назад), что очищает последний символ на экране терминала. Если ECHOK установлен, а ECHOKE нет, то после символа стирания строки передается NL, чтобы подчеркнуть, что строка была стерта.

Символ переключения режима (escape), идущий перед символами очистки или стирания строки, лишает эти символы их функции. Если установлен флаг ISIG, вводимые символы проверяются на совпадение с символами INTR, QUIT, SUSP и DSUSP. Если вводимый символ соответствует одному из них, посылается соответствующий сигнал. Если ISIG не установлен, не выполняется никакой проверки.

Если установлен флаг IEXTEN, то над входными данными будут выполняться функции из расширенного набора, зависящие от реализации. Этот флаг должен быть установлен для распознавания символов WERASE, REPEINT, DISCARD и LNEXT.

Запирание терминала - пример

Эта программа запирает терминал пользователя. Она запрашивает пользователя ввести «ключ». Конечно, пользователю было бы удобнее, чтобы ключ совпадал с его паролем, но в современных Unix-системах хэши паролей доступны только пользователю root, поэтому ключ необходимо вводить при запуске программы. Эхо выключено, поэтому ключ не будет показан на терминале. Этот же ключ должен быть введен, чтобы получить доступ к терминалу. Программа работает так:

9 Объявляет структуру termios для сохранения установок терминала.

10 Объявляет переменную tcflag_t для сохранения текущих значений c_lflag.

11 Объявляет массив символов для сохранения значения первого ключа. Ключ может быть очень длинным, так как BUFSIZ имеет большое значение.

13 Программа получает текущие установки терминала. Первый параметр - дескриптор файла стандартного ввода, полученный макросом fileno, определенным в <stdio.h>. Если stdin переназначен, то программа работать не будет. Второй аргумент - адрес структуры termios.

14 Сохраняется значение поля c_lflag. Это значение затем будет использовано для восстановления состояния терминального интерфейса.

15-16 Запрещается сравнение ввода с управляющими символами INTR, QUIT, SUSP и DSUSP. Эхо выключается. tcsetattr(3C) вызывается с флагом TCSAFLUSH для изменения состояния терминального интерфейса и сброса всех введенных символов.

17 Запоминается ключ, который позднее будет использован для отпирания терминала.

18-25 Для того, чтобы выйти из этого цикла, программа должна получить значение, совпадающее со значением исходного ключа. Если ключи совпадают, терминальный интерфейс возвращается в исходное состояние. Это не делается автоматически при завершении программы. Если программа завершится ненормально, терминал может остаться в странном состоянии.

28 Эта функция возвращает указатель на строку символов, которая содержит ключ, введенный пользователем.

30 Объявляется массив символов для сохранения значения ключа. Он объявлен как static, и указатель на этот массив будет возвращен функцией.

32 Выдается приглашение

33 Первый символ в массиве line[] устанавливается в невозможное значение. Для чего это нужно? Ответ: Если getkey() был вызван во второй раз, и был введен только EOF, не будет считано ни одного символа; содержимое line[] будет тем же, что и раньше, и возвращенная строка совпадет со значением, полученным от первого вызова этой функции, что совершенно неправильно.

34 Строка ввода считывается без эхо. Не делается проверки на совпадение с управляющими символами, такими как INTR, QUIT и т.д. Эти символы обрабатываются так же, как обычные.

Файл: termlock.c

ЗАПИРАНИЕ ТЕРМИНАЛА - ПРИМЕР

```
 1 #include <string.h>
 2 #include <unistd.h>
 3 #include <stdio.h>
 4 #include <termios.h>
 5 static char *getkey(void);
 6
 7 main()         /* lock the terminal */
 8 {
 9      struct termios tty;
10      tcflag_t savflags;
11      char key[BUFSIZ];
12
13      tcgetattr(fileno(stdin), &tty);
14      savflags = tty.c_lflag;
15      tty.c_lflag &= ~(ISIG | ECHO);
16      tcsetattr(fileno(stdin), TCSAFLUSH, &tty);
17      strcpy(key, getkey());
18      for (;;) {
19       if (strcmp(key, getkey()) == 0){
20            tty.c_lflag = savflags;
21            tcsetattr(fileno(stdin), TCSAFLUSH, &tty);
22            break;
23       }
24       fprintf(stderr,"incorrect key try again.\n");
25      }
26 }
27
28 static char *getkey(void)     /* prompt user for key */
29 {
30      static char line[BUFSIZ];
31
32      fputs("Key: ", stderr);
33      line[0] = '\377';  /*change first char for EOF to fgets*/
34      fgets(line, BUFSIZ, stdin);
35      fputs("\n", stderr);
36      return(line);
37 }
```

Неканонический ввод

В обычном (каноническом) режиме символы собираются вместе, пока не будет введена полная строка, завершенная символом NL. Только после этого вызов read(2) возвращает управление, даже если он запрашивал только один символ. Возвращенное вызовом read(2) значение показывает количество символов, которые были прочитаны на самом деле до ввода NL.

Однако в некоторых прикладных программах, таких, как обработка экранных форм или полноэкранных редакторах, "строки" ввода не имеют смысла. Например, эти программы могут требовать символы по мере их ввода с клавиатуры. Если очистить флаг ICANON в c_iflag, вводимые символы не будут собираться в строки и read(2) будет читать их по мере ввода.

Параметры MIN и TIME определяют условия, при которых будет удовлетворен запрос read(2). MIN определяет минимальное количество символов, которые должны быть получены. TIME представляет собой таймер с квантом времени 0.1 секунды, который сбрасывается при вводе каждого символа. Таким образом, TIME кодирует не общее время ввода строки, а межсимвольный интервал. Это сделано для упрощения считывания терминальных кодов расширения, ведь терминалы передают последовательные символы кода быстрее, чем обычный человек может нажимать клавиши.. Символы EOF и EOL в неканоническом режиме не используются, поэтому эти позиции в массиве c_cc[] используются для MIN и TIME соответственно. Ниже описаны четыре возможных сочетания значений MIN и TIME:

MIN > 0, TIME > 0. В этом случае, TIME служит для измерения времени между вводом одиночных символов и стартует после получения первого символа. Счетчик времени сбрасывается после каждого очередного символа. Если до истечения интервала времени будет получено MIN символов, запрос read(2) удовлетворяется. Если, наоборот, время истекает раньше, чем было считано MIN символов, то все введенные до этого момента символы возвращаются пользователю. Замечание: если TIME истекло, то будет возвращен по крайней мере один символ. Если MIN равен 1, значение TIME не играет роли.

MIN > 0, TIME = 0 Если значение TIME равно нулю, таймер не используется. Имеет значение только MIN. В этом случае запрос read(2) удовлетворяется только тогда, когда получены MIN символов.

MIN = 0, TIME > 0 Если MIN равен нулю, TIME больше не является счетчиком межсимвольного

времени. Теперь таймер активизируется при обработке системного вызова read(2). Запрос read(2) удовлетворяется когда поступил хотя бы один символ или истекло время. Если в течении TIME*0.1 секунд после начала чтения не поступило ни одного символа, запрос возвращает управление с нулевым количеством прочитанных символов.

MIN = 0, TIME = 0 В этом случае, read(2) возвращает управление немедленно. Возвращается минимум из запрошенного и имеющегося на данный момент в буфере количества символов, без ожидания ввода дополнительных символов.

Клавиатурный тренажер - Пример

Эта программа может служить в качестве клавиатурного тренажера и является примером неканонического ввода. Она выводит на экран строку текста, и пользователь должен напечатать ее. Символы по мере ввода проверяются. Если введен неправильный символ, программа издает звуковой сигнал и печатает звездочку. Если введен правильный символ, он выводится на экран. Программа работает так:

15-20 Терминальный специальный файл открывается для чтения, и его текущий режим записывается в структуру termios. Библиотечная функция isatty(3F) проверяет, связан ли файловый дескриптор 1 с терминалом. Иными словами, эта программа не должна исполняться с использованием перенаправления стандартного ввода/вывода shell. Функция isatty описана на странице руководства ttyname(3F).

22-24 Терминальный интерфейс переводится в режим неканонического ввода без эхо и без обработки специальных символов.

25 Вызовом стандартной библиотечной функции setbuf(3) запрещается локальная буферизация в библиотечных функциях вывода. Это приводит к тому, что putchar(3) в следующих строках будет немедленно вызывать write(2) в стандартный вывод.

28-37 Этот цикл читает по одному символу и сравнивает его с соответствующим символом строки text. Если символ введен правильно, он немедленно выводится на терминал. Иначе издается звуковой сигнал и на терминал выводится звездочка.

Ниже приведен пример работы программы:

```
$ typtut
Type in beneath the following line

The quick brown fox jumped over the lazy dog's back
The *uick b*own f** jumped over t*e lazy dog's back

number of errors: 5
```

Файл: typtut.c

КЛАВИАТУРНЫЙ ТРЕНАЖЕР - ПРИМЕР НЕКАНОНИЧЕСКОГО ВВОДА

```
 1 #include <unistd.h>
 2 #include <stdio.h>
 3 #include <fcntl.h>
 4 #include <termios.h>
 5 #include <string.h>
 6 #include <stdlib.h>
 7
 8 main()
 9 {
10      char ch, *text =
11      "The quick brown fox jumped over the lazy dog\'s back";
12      int fd, i, errors = 0, len;
13      struct termios tty, savtty;
14
15      fd = open("/dev/tty", O_RDONLY);
16      tcgetattr(fd, &tty);
17      if (isatty(fileno(stdout)) == 0) {
18       fprintf(stderr,"stdout not terminal\n");
19       exit(1);
20      }
21      savtty = tty;
22      tty.c_lflag &= ~(ISIG | ICANON | ECHO);
23      tty.c_cc[VMIN] = 1;     /* MIN */
24      tcsetattr(fd, TCSAFLUSH, &tty);
25      setbuf(stdout, (char *) NULL);
26      printf("Type beneath the following line\n\n%s\n", text);
27      len = strlen(text);
28      for (i = 0; i < len; i++) {
29       read(fd, &ch, 1);
30       if (ch == text[i])
31            putchar(ch);
32       else {
33            putchar('\07');
34            putchar('*');
35            errors++;
36       }
37      }
38      tcsetattr(fd, TCSAFLUSH, &savtty);
39      printf("\n\nnumber of errors: %d\n", errors);
40 }
```

Программа просмотра файла - Пример

Эта программа может использоваться для просмотра файла на терминале и служит другим примером неканонического ввода. Она служит альтернативой CTRL-S и CTRL-Q, которые, соответственно, задерживают и возобновляют вывод. В этой программе любая клавиша приостанавливает или возобновляет вывод на терминал. Например, пробел может использоваться как переключатель. Этот эффект достигается изменением значения MIN

между нулем и единицей. Эта программа работает так:

13-16 Стандартная библиотечная функция fopen(3) открывает файл для просмотра.

17 Считывается текущий режим терминального интерфейса.

18 Этот режим сохраняется. Позднее он будет использоваться для восстановления состояния терминального интерфейса.

19-22 Терминальный интерфейс переключается в режим неканонического ввода. Кроме того, INTR, QUIT и остальные управляющие символы не анализируются и эхо выключено. Чтение с терминала будет ожидать в течении 0.1 секунды, потому что MIN равен нулю, а TIME равен 1.

24-33 Этот цикл считывает строки из файла и выводит их на терминал.

25 read(2) пытается считать с терминала один символ. Так как чтение возвращает управление немедленно (без ожидания), символ будет прочитан, только если он был введен до вызова read(2). Непрочитанные символы накапливаются в буфере. Если считан символ, read(2) возвращает 1 и исполняются операторы 27-31. Если не прочитано ни одного символа,read(2) возвращает 0. Это называется опросом ввода с терминала.

26-27 MIN установлен в единицу, так что запросы чтения с терминала будут ждать ввода.

28 Как только символ введен, запрос read(2) удовлетворяется и возвращает управление.

29-30 Чтение с терминала снова переводится в режим опроса.

34 Восстанавливается исходный режим работы терминала. После нажатия клавиши для приостановки вывода возникает небольшая задержка, во время которой выводится несколько лишних строк. Это связано с буферизацией вывода и низкой скоростью работы терминала.

Как можно прекратить просмотр длинного файла? Например, после чтения в строке 25, мы можем проверять символ на равенство букве q (quit). Если был введен этот символ, программа выходит из цикла. Можно даже использовать символ DEL, так как он читается наравне с остальнымисимволами.

Файл: lister.c

ПРОГРАММА ПРОСМОТРА ФАЙЛОВ - ПРИМЕР НЕКАНОНИЧЕСКОГО ВВОДА

```
 1 #include <unistd.h>
 2 #include <stdio.h>
 3 #include <stdlib.h>
 4 #include <fcntl.h>
 5 #include <termios.h>
 6
 7 main(int argc, char *argv[])
 8 {
 9     struct termios tty, savtty;
10     char ch, line[BUFSIZ];
11     FILE *fp;
12
13     if ((fp = fopen(argv[1], "r")) == NULL) {
14         printf("Cannot open %s\n", argv[1]);
15         exit(1);
16     }
17     tcgetattr(fileno(stdin), &tty);
18     savtty = tty;
19     tty.c_lflag &= ~(ISIG | ICANON | ECHO);
20     tty.c_cc[VMIN] = 0;     /* no characters */
21     tty.c_cc[VTIME] = 1;    /* wait for 100 msec */
22     tcsetattr(fileno(stdin), TCSANOW, &tty);
23
24     while (fgets(line, BUFSIZ, fp) != NULL) {
25         if (read(fileno(stdin), &ch, 1) == 1) {
26             tty.c_cc[VMIN] = 1;     /* one char */
27             tcsetattr(fileno(stdin), TCSANOW, &tty);
28             read(fileno(stdin), &ch, 1);
29             tty.c_cc[VMIN] = 0;     /* no chars */
30             tcsetattr(fileno(stdin), TCSANOW, &tty);
31         }
32         fputs(line, stdout);
33     }
34     tcsetattr(fileno(stdin), TCSANOW, &savtty);
35     fclose(fp);
36 }
```

Передача двоичного файла - Пример

На следующей странице приведены подпрограммы для установки терминального интерфейса в режим ввода/вывода необработанных данных ("сырой") и восстановления исходного режима. Эти подпрограммы работаюттак:

8 fd присваивается 0 или 1, если setrawio вызывается получателем (recv) или передатчиком (xmit), соответственно.

10-11 Если дескриптор файла 0 или 1 не ассоциирован с терминальным специальным файлом, эта функция возвращает управление немедленно. Это позволяет программе, использующей setrawio, перенаправить свой стандартный ввод/вывод в файл или программный канал (в этом случае, setrawio вообще не нужен). Если же дескриптор ассоциирован с терминальным специальным файлом, то режим интерфейса переключается на "сырой" ввод/вывод.

12-15 Для заданного дескриптора файла считывается значение структуры termios.

16 Копия структуры termios сохраняется для восстановления режима терминального интерфейса, который был до вызова setrawio.

17-22 Флаги в структуре termios устанавливаются для приема и передачи произвольных восьмибитных данных. Срезание старшего бита, отображение вводимых символов и управление потоком ввода выключены. Размер символа установлен равным восьми битам, и выключен контроль четности. Поиск специальных управляющих символов, канонический ввод и эхо в поле флагов локального режима также выключены. Заметьте, что эта функция одна и та же как для ввода, так и для вывода, так как установки флагов для ввода не влияют на вывод, и наоборот.

23-24 Для неканонического ввода MIN устанавливается равным размеру буфера ввода, используемого в вызове read(2). Таймер не используется, поэтому TIME устанавливается в ноль.

25 Режим терминального интерфейса будет изменен после того, как весь вывод будет передан, а ввод - сброшен.

32 Терминальный интерфейс возвращается в то состояние, в котором он находился до вызова функции setrawio.

Файл: setrawio.c

ПЕРЕДАЧА ДВОИЧНОГО ФАЙЛА - ПРИМЕР setrawio.c

```
 1 #include <stdlib.h>
 2 #include <stdio.h>
 3 #include <termios.h>
 4 #include "xmit.h"
 5
 6 static struct termios tty, savtty;
 7
 8 void setrawio(int fd)     /* set "raw" input/output modes */
 9 {
10      if (!isatty(fd))
11       return;
12      if (tcgetattr(fd, &tty) == -1) {
13       perror("tcgetattr");
14       exit(2);
15      }
16      savtty= tty;
17      tty.c_iflag &= ~(BRKINT | ISTRIP | INLCR | ICRNL
18       | IUCLC | IXON);
19      tty.c_oflag &= ~OPOST;
20      tty.c_cflag |= CS8;
21      tty.c_cflag &= ~PARENB;
22      tty.c_lflag &= ~(ISIG | ICANON | ECHO);
23      tty.c_cc[VMIN] = BLOCKSIZE;     /* MIN */
24      tty.c_cc[VTIME] = 0;          /* TIME */
25      tcsetattr(fd, TCSAFLUSH, &tty);
26 }
27
28 void restorio(int fd)     /* restore terminal modes */
29 {
30      if (!isatty(fd))
31       return;
32      tcsetattr(fd, TCSAFLUSH, &savtty);
33 }
```

Сессии и группы процессов

Все процессы объединены в сессии. Процессы, принадлежащие к одной сессии, определяются общим идентификатором сессии (sid). Лидер сессии - это процесс, который создал сессию вызовом setsid(2). Идентификатор процесса лидера сессии совпадает с его sid. Сессия может выделить себе управляющий терминал для того, чтобы дать пользователю возможность управлять исполнением заданий (групп процессов) внутри сессии. При входе в систему создается сессия, которая имеет идентификатор сессии, равный идентификатору процесса вашего входного shell'а. Также, при открытии каждого окна xterm(1) или закладки gnome-terminal(1), создается сессия, идентификатор которой совпадает с идентификатором дочернего процесса, запущенного терминальным эмулятором.

Если ваш командный процессор не предоставляет управления заданиями, все процессы в вашей сессии являются также членами единственной в этой сессии группы процессов, которая была создана вызовом setsid(2). В этом случае, функциональность сессии совпадает с функциональностью группы процессов.

В командном процессоре, предоставляющем управление заданиями (ksh(1), jsh(1), bash(1)), управляющий терминал совместно используется несколькими группами процессов, так как для каждой команды, запущенной с управляющего терминала, создается своя группа процессов. Командный процессор называет такие команды «заданиями» (jobs). Одновременно работающие задания идентифицируются номерами, обычно совпадающими с порядком их запуска.

Каждая группа процессов имеет лидера - процесс, идентификатор которого совпадает с идентификатором группы процессов. Управляющий терминал выделяет одну из групп процессов в сессии, как группу основных процессов (процессов первого плана). Все остальные процессы в сессии принадлежат к группам фоновых процессов. Группа процессов первого плана получает сигналы, посланные с терминала. По умолчанию, группа процессов, связанная с процессом, который выделил себе управляющий терминал, изначально становится группой основных процессов.

Кроме того, если процесс из фоновой группы пытается читать с терминала или выводить на него данные, он получает сигнал, соответственно, SIGTTIN или SIGTTOUT. Оба эти сигнала приводят к остановке соответствующего процесса. Shell при этом выводит сообщение [имя программы] stopped: tty input. Для продолжения исполнения такой программы необходимо перевести соответствующую группу процессов на первый план командой fg.

Обычные команды запускаются как задания первого плана. Shell ожидает завершения лидера группы этого задания и выдает приглашение только после его завершения. Если в конце команды стоит символ &, shell запускает такую команду как фоновое задание и не дожидается ее завершения. Если пользователь во время работы команды первого плана введет символ VSUSP (Ctrl-Z), группа получает сигнал SIGTSTP и останавливается, а shell выдает приглашение. Пользователь может вернуться к взаимодействию с этой группой процессов, введя команду fg, или продолжить её исполнение в фоне командой bg. Если пользователь имеет несколько приостановленных или фоновых заданий, он может выбирать между ними, идентифицируя их по номерам. Так, переключение на задание 3 делается командой fg %3.

Вывести список заданий, их номера и состояния можно командой jobs.

Также, встроенная команда kill у shell'ов с управлением заданиями, может принимать номер задания вместо номера процесса; при этом сигнал будет послан всем процессам соответствующей группы.

Получение/установка идентификатора сессии

Каждый процесс принадлежит к сессии и группе процессов. Сессия создается для вас, когда вы входите в систему. Первый терминал, открытый лидером сессии, который не был уже ассоциирован с другой сессией, становится управляющим терминалом для этой сессии. Если при открытии терминала лидер сессии укажет флаг NOCTTY, терминал не станет управляющим. Это позволяет процессам-демонам выводить сообщения на системную консоль.

Управляющий терминал генерирует сигналы завершения и прерывания (quit и interrupt), а также сигналы управления заданиями. Управляющим терминалом для вашего shell'a является тот терминал, с которого вы вошли в систему. Управляющий терминал наследуется процессом, порожденным при помощи fork(2). Процесс может разорвать связь со своим управляющим терминалом, создав новую сессию с использованием setsid(2).

Если сессия так и не откроет управляющий терминал, соответствующий процесс будет называться «демоном» (daemon). Большинство системных сервисных процессов, таких, как init(1M), svc.startd(1M) crond(1M) или сетевых сервисов, таких, как sshd(1M), запускаются как демоны. Иногда демонами называют также системные процессы ttymon(1M), обслуживающие терминальные порты, хотя эти процессы имеют управляющие терминалы.

Если вызывающий процесс не является уже лидером группы процессов, setsid(2) устанавливает идентификаторы группы процессов и сессии вызывающего процесса равными его идентификатору процесса и отсоединяет его от управляющего терминала. setsid(2) создает новую сессию, превращая вызвавший процесс в лидера этой сессии. Новые сессии создаются чтобы:

1. отсоединить вызвавший процесс от терминала, так что этот процесс не будет получать от этого терминала сигналы SIGHUP, SIGINT и сигналы управления заданиями.

2. позволить процессу назначить новый управляющий терминал. Только лидер сессии может назначить управляющий терминал. Например, ttymon создает новую сессию и, таким образом, назначает управляющий терминал, когда пользователь входит в систему.

getsid(2) возвращает идентификатор сессии процесса с идентификатором, равным pid. Если pid равен нулю, getsid(2) возвращает идентификатор сессии вызвавшего процесса.

Получение/установка идентификатора группы процессов

Группа процессов - это совокупность процессов с одним и тем же идентификатором группы процессов. Управляющий терминал считает одну из групп процессов в сессии группой основных процессов. Все процессы в основной группе будут получать сигналы, относящиеся к терминалу, такие как SIGINT и SIGQUIT.

Новый идентификатор группы процессов может быть создан вызовом setpgid(2). Группы процессов, отличные от основной группы той же сессии, считаются группами фоновых процессов. ksh использует группы процессов для управления заданиями. Фоновые процессы не получают сигналов, генерируемых терминалом. Системный вызов setpgid(2) устанавливает идентификатор группы процессов следующим образом:

pid == pgid создается группа процессов с идентификатором, равным pid; вызвавший процесс становится лидером этой группы

pid != pgid процесс pid становится членом группы процессов pgid, если она существует и принадлежит к этой сессии.

Если pid равен 0, будет использован идентификатор вызывающего процесса. Если pgid равен нулю, процесс с идентификатором pid станет лидером группы процессов. pid должен задавать процесс, принадлежащий к той же сессии, что и вызывающий.

Идентификатор группы процессов - атрибут, наследуемый порожденными процессами. Процесс может определить свой идентификатор группы, вызывая getpgrp(2) или getpgid(2).

Системный вызов waitid(2) и библиотечная функция waitpid(3C) могут использоваться для ожидания подпроцессов, принадлежащих определенной группе. Кроме того, можно послать сигнал всем процессам в заданной группе.

В ksh и bash для каждой исполняемой команды создается новая группа процессов.

В sh все процессы принадлежат к одной группе, если только сам процесс не исполнит setsid(2) или setpgid(2).

Установить идентификатор группы процессов - Пример

Этот пример показывает, как создать группу процессов, используя setpgid(2). Создаются три подпроцесса, и каждый распечатывает значение своего идентификатора группы процессов. Пример демонстрируется так:

```
$ setpgid
[6426] Original process group id: 179
[6426] New process group id: 6426

    [6427] Process group id: 6426

    [6428] Process group id: 6426

    [6429] Process group id: 6426
```

Эта выдача предполагает, что программа запущена из sh. Любой процесс, запущенный с управляющего терминала, принадлежит основной группе. Таким образом, процесс изначально принадлежит к группе основных процессов. Затем, в строке 14, он становится лидером группы процессов. Его подпроцессы наследуют новый идентификатор группы процессов, и принадлежат той же группе, что и их родитель. Эта новая группа процессов будет фоновой, и поэтому не будет получать сигналы, связанные с терминалом. Если программа исполняется из ksh или bash, вывод будет выглядеть так:

```
$ setpgid
[6426] Original process group id: 6426
[6426] New process group id: 6426

    [6427] Process group id: 6426

    [6428] Process group id: 6426

    [6429] Process group id: 6426
```

ksh создает новую группу процессов для каждой исполняемой команды. Поэтому setpgid(2) в строке 14 не делает ничего наблюдаемого, ведь процесс уже является лидером группы. Чтобы добиться более интересного поведения, можно сначала запустить sh, а только потом запускать программу, тогда при запуске программы лидером ее группы будет процесс sh.

Файл: setpgid.c

УСТАНОВИТЬ ИДЕНТИФИКАТОР ГРУППЫ ПРОЦЕССОВ - ПРИМЕР

```
 1 #include <sys/types.h>
 2 #include <unistd.h>
 3 #include <stdlib.h>
 4 #include <stdio.h>
 5 #define   NUMCHILD   3
 6
 7 main()
 8 {
 9     int i;
10
11     printf("[%ld] Original process group id: %ld\n",
12     getpid(), getpgid(0));
13
14     if (setpgid(0, 0) == -1) {
15     perror("");
16     exit(1);
17     }
18
19     printf("[%ld] New process group id: %ld\n",
20     getpid(), getpgid(0));
21
22     for (i = 0; i < NUMCHILD; i++ ) {
23     if (fork() == 0) {    /* child */
24         printf("\n\t[%ld] Process group id: %ld\n",
25         getpid(), getpgid(0));
26         exit(0);
27     }
28     }
29 }
```

Получение и установка группы процессов первого плана.

Функция tcsetpgrp(3C) устанавливает для терминала с дескриптором файла fildes идентификатор группы первого плана равным pgid. Помните, что процессы из основной группы получают сигналы, связанные с терминалом, такие как SIGINT и SIGQUIT. Если фоновый процесс попытается сделать tcsetpgrp(3C), он получит сигнал SIGTTOU. Например, если фоновый процесс попытается стать основным процессом, он получит этот сигнал. Фоновый процесс, однако, может проигнорировать или обработать этот сигнал.

Кроме того, фоновые процессы получают сигналы SIGTTIN и SIGTTOU при попытке читать с управляющего терминала или писать на него.

Командный процессор bash под SVR4 традиционно собирают с игнорированием SIGTTOU, поэтому фоновые процессы, запущенные из-под bash, не могут читать с терминала, но могут писать на него, так что их вывод может смешиваться с выводом текущей программы первого плана. Это сделано для совместимости с традиционными Unix-системами, где не было способа заблокировать вывод фоновых процессов.

tcgetpgrp(2) возвращает идентификатор группы основных процессов для терминала с дескриптором файла fildes.

tcgetsid(2) возвращает идентификатор сессии, для которой управляющим терминалом является терминал с дескриптором файла fildes.

Пример - Группа первого плана, связанная с терминалом

Эта программа демонстрирует изменение группы процессов первого плана. Это одна из основных задач при управлении заданиями. Программа работает так:

11 Распечатывается начальный идентификатор группы для этого процесса.

13 Создается подпроцесс.

14-17 При помощи setpgid(2) создается новая группа.

19 Распечатывается новый идентификатор группы процессов.

21 Подпроцесс засыпает на 10 секунд.

26 Распечатывается идентификатор группы основных процессов. Родительский процесс принадлежит группе первого плана.

27 Так как родительский процесс принадлежит основной группе, во время исполнения вызова sleep(2), этот процесс может получать сигналы, связанные с терминалом. Например, пользователь может послать SIGINT и родительский процесс завершится.

28 Группа основных процессов изменяется вызовом tcsetpgrp(2). Теперь порожденный процесс принадлежит к основной группе.

30 Все связанные с терминалом сигналы будут получены подпроцессом и вызовут его завершение.

Из-под ksh или bash эта программа исполняется следующим образом:

```
$ tcsetpgrp
Original PGID: 8260
New PGID: 8376
Foreground PGID: 8260  (terminal signals received by parent)
Foreground PGID: 8376  (terminal signals received by child)
child done
parent done
```

Из-под sh, shell необходимо снова сделать основным процессом прежде, чем наш процесс завершится. Иначе пользователь будет выброшен из системы при завершении программы.

Файл: tcsetpgrp.c

ПРИМЕР - ГРУППА ОСНОВНЫХ ПРОЦЕССОВ

```
 1 #include <sys/types.h>
 2 #include <unistd.h>
 3 #include <stdlib.h>
 4 #include <termios.h>
 5 #include <stdio.h>
 6
 7 main()
 8 {
 9    pid_t pid;
10
11    printf("Original PGID: %ld\n", getpgid(0));
12
13    if ((pid = fork()) == 0) {
14       if (setpgid(0, 0) == -1) {
15       perror("");
16       exit(1);
17       }
18
19       printf("New PGID: %ld\n", getpgid(0));
20
21       sleep(10);
22       printf("child done\n");
23       exit(0);
24    }
25
26    printf("Foreground PGID: %ld\n", tcgetpgrp(0));
27    sleep(5);  /* parent receives terminal signals */
28    tcsetpgrp(0, pid);
29    printf("Foreground PGID: %ld\n", tcgetpgrp(0));
30    wait(0);  /* child receives terminal signals */
31    printf("done parent\n");
32 }
```

10. ПРОГРАММНЫЕ КАНАЛЫ

Обзор

В предыдущих разделах вы научились создавать несколько параллельно исполняющихся процессов. Вы научились делать так, чтобы родительский процесс ожидал завершения одного или нескольких порождённых. Часто параллельные процессы должны каким-то образом взаимодействовать для решения общей задачи, и поэтому нужны средства для обмена информацией между ними.

Операционная система UNIX предоставляет богатый набор средств межпроцессного взаимодействия (IPC - InterProcess Communication). Аббревиатура IPC будет использоваться далее в этом курсе, потому что она короче фразы «межпроцессное взаимодействие». В этом разделе вы будете изучать одну из таких возможностей — программные каналы или трубы (pipes). Программные каналы — это механизм передачи информации от одного процесса к другому. Вы изучите системные вызовы и стандартные библиотечные функции, которые создают программные каналы и выполняют ввод и вывод через них.

IPC с использованием файлов

Рассмотрим использование обычных файлов для межпроцессного взаимодействия:

Доступность файлов определяется тремя наборами битов прав доступа. Процессы, обменивающиеся информацией через файл, не обязаны быть «родственниками». Под «родством» здесь понимаются отношения родитель-порождённый или наличие общего родителя.

Обычные файлы имеют ограничения по размеру. Размеры файлов могут быть ограничены административно, вызовом ulimit(2) или дисковой квотой, логически, максимальным размером длины файла в файловой системе данного типа, или физически, объёмом доступного пространства.

Программные каналы, в отличие от регулярных файлов, представляют собой непрерывный поток байтов, по которому может быть передано произвольно большое количество информации. При этом собственная ёмкость канала очень невелика. В качестве аналогии можно предложить тоннель Линкольна, соединяющий Нью-Йорк и Нью-Джерси, пропускающий через себя миллионы автомобилей, в то время как в любой заданный момент тоннель вмещает не более чем, скажем, семь сотен машин.

Время жизни обычного файла не зависит от времени жизни использующих его процессов. Файлы могут создаваться и уничтожаться вовсе не теми программами, которые используют их для взаимодействия. Файлы, как правило, переживают перезагрузку ОС. Кроме того, данные в файле сохраняются и тогда, когда ни одна программа их не использует.

Основная проблема, возникающая при обмене информацией через обычный файл – это отсутствие синхронизации. Если предположить, что ёмкость файла не является проблемой, как читающий процесс узнает, что пишущий процесс окончил запись? Использование сигналов для этой цели — неудовлетворительное решение. В разделе «Захват файлов и записей» изучалось более подходящее средство, блокировка записей. Но даже с использованием блокировок, сложность задачи возрастает, так как, кроме правильности самих процессов, вы должны заботиться и о правильности синхронизации между ними.

Программные каналы

Программные каналы - это линии связи между двумя или более процессами. По традиции, прикладные программы используют каналы следующим образом: один процесс пишет данные в канал, а другой читает их оттуда. В SVR4 каналы стали двунаправленным механизмом, так что два процесса могут передавать информацию в обоих направлениях через один программный канал.

Существует два типа программных каналов: неименованные, часто называемые просто трубами, и именованные каналы. Существуют стандартные библиотечные функции, упрощающие использование каналов.

В отличие от обычных файлов, каналы могут пропускать через себя неограниченно большой объем информации, хотя сами имеют небольшую ёмкость (несколько десятков логических блоков). Размер внутреннего буфера канала можно посмотреть вызовом pathconf(2) с параметром _PC_PIPE_BUF. На самом деле, вызов pathconf(2) с этим параметром возвращает не полный размер внутреннего буфера, а размер блока данных, который гарантированно может быть записан при помощи вызова write(2) с одной попытки. В Solaris 10 pathconf("/",_PC_PIPE_BUF) возвращает 5120 байт (10 блоков), но эксперименты показывают, что реальный объем буфера трубы составляет более 100 килобайт.

Процессы не обязаны заботиться о переполнении канала избытком данных или о невозможности читать из пустого канала. В канальный механизм встроена синхронизация между читающим и пишущим процессами: пишущий процесс блокируется, т.е. приостанавливает исполнение, при попытке записи в переполненный канал, и, соответственно, читающий процесс останавливается при попытке чтения из пустого канала.

Также, процесс останавливается, если он открывает именованный канал для чтения, а его партнёр не открыл этот же канал для записи, и наоборот. Далее в этом разделе вы изучите флаги O_NDELAY и O_NONBLOCK, выключающие этот автоматический механизм синхронизации при открытии именованного канала.

Оба типа каналов работают, в основном, одинаково, но используются несколько различным образом. Они создаются различными системными вызовами. Время жизни неименованного канала не больше времени жизни процессов, использующих его. Напротив, именованный канал имеет соответствующую запись в директории и существует, пока эту запись явным образом не удалят, поэтому такой канал, обычно, переживает не только процессы, работающие с ним, но и перезагрузку системы. Запись в директории используется для управления доступом к именованному каналу. Главное преимущество именованных каналов над обычными состоит в том, что их могут использовать неродственные процессы.

Доступ к данным в программном канале

Канал идентифицируется таким же файловым дескриптором, как и открытые обычные и специальные файлы. Большинство системных вызовов для работы с файловыми дескрипторами применимо к каналам.

Каналы используются командными оболочками для реализации конвейеров (pipeline). При создании конвейера, стандартный вывод одного процесса перенаправляется в дескриптор трубы, открытый на запись, а стандартный ввод следующего процесса — в дескриптор, открытый на чтение. Многие стандартные утилиты Unix, такие, как sort(1), grep(1), sed(1), gzip(1) представляют собой «фильтры», то есть программы, последовательно обрабатывающие поток данных на вводе и преобразующие его в поток данных на выводе. Такие программы могут работать с терминалом, регулярными файлами, многими типами устройств и программными каналами, не замечая различий. В виде фильтров реализованы также модули компилятора GNU Compiler Collection: препроцессор, собственно компилятор и ассемблер запускаются в отдельных процессах и обмениваются промежуточными представлениями программы через канал. К сожалению, работа редактора связей требует произвольного доступа к объектному файлу, поэтому ассемблер не может передавать объектный модуль линкеру через трубу, то есть последний этап компиляции всё-таки требует создания промежуточных файлов.

Данные пишутся в канал так же, как и в обычный файл, при помощи системного вызова write(2). Как упоминалось выше, если канал не имеет места для записи всех данных, write(2) останавливается. Система не допускает частичной записи: write(2) блокируется до момента, пока все данные не будут записаны, либо пока не будет обнаружена ошибка.

Данные читаются из канала при помощи системного вызова read(2). В отличие от обычных файлов, чтение разрушает данные в канале. Это означает, что вы не можете использовать lseek(2) для попыток прочитать данные заново.

С одним каналом может работать несколько читающих и пишущих процессов. На практике, нежелательно иметь несколько читающих процессов — без дополнительной синхронизации доступа к каналу это, скорее всего, приведёт к потере данных. Но канал с несколькими пишущими в него процессами может иметь смысл.

Если процесс пытается писать в канал, из которого никто не читает, он получит сигнал SIGPIPE.

Вопрос: если вы хотите прекратить команду, использующую конвейер, какой из процессов вы должны убить?

Ответ: последний процесс в конвейере.

Диаграмма на иллюстрации показывает данные, записанные в программный канал, но еще не прочитанные из него. Система использует два указателя для того, чтобы следить, куда пишутся данные, и откуда они читаются. Для этого повторно используются те же самые десять блоков.

Системный вызов pipe(2)

Программные каналы создаются системным вызовом pipe(2). Возвращаемое значение показывает, успешно завершился вызов или нет.

Системный вызов pipe(2) заполняет массив целых чисел двумя дескрипторами файлов. В ранних версиях системы первый элемент массива содержал дескриптор, связанный с концом канала, предназначенным для чтения; второй - для записи. В SVR4 оба дескриптора открыты для чтения и записи, позволяя двусторонний обмен данными.

Как правило, программные каналы используются следующим образом: после системного вызова pipe(2), создавшего программный канал, вызовом fork(2) создаётся подпроцесс. Затем родительский и порождённый процессы закрывают тот из концов канала, который не собираются использовать. Родительский процесс может также создать два подпроцесса, каждый из которых закроет ненужный ему конец программного канала. Если родитель не хочет взаимодействовать с порождёнными им процессами, он должен закрыть оба конца канала.

Неиспользуемые файловые дескрипторы необходимо закрывать потому, что программный канал выдаёт условие конца файла только когда его пишущий конец закрыт. При fork(2) происходит дублирование файлового дескриптора, а программный канал считается закрытым только когда будут закрыты все копии связанного с этим каналом дескриптора. Если вы забудете закрыть один из дескрипторов, процесс, ожидающий конец файла в канале (возможно, ваш же собственный процесс!), никогда его не дождётся.

Особенности системных вызовов для неименованных каналов

Есть некоторые особенности при работе следующих системных вызовов с программными каналами:

open(2) Этот системный вызов не нужен при работе с каналом, так как pipe(2) сам открывает оба конца канала.

close(2) Этот системный вызов используется обычным образом и закрывает канал, так же как любой другой файл, когда работа с ним окончена. Ниже описано, как он действует на read(2) и write(2).

read(2) Этот системный вызов читает столько данных, сколько на момент вызова есть в канале. Если количество байтов в канале меньше, чем требуется, read(2) возвращает значение меньшее, чем его последний аргумент. read(2) возвращает 0, если обнаруживает, что другой конец канала закрыт, т.е. все потенциальные пишущие процессы закрыли свои файловые дескрипторы, связанные с входным концом канала. Если пишущий процесс опередил читающий, может потребоваться несколько операций чтения, прежде чем read(2) возвратит 0, показывая конец файла. Если буфер канала пуст, но файловый дескриптор другого конца ещё открыт, read(2) будет заблокирован. Это поведение может быть изменено флагами O_NONBLOCK и O_NDELAY.

write(2) В отличие от read(2), который штатно возвращает меньше данных, чем было запрошено, write(2) стремится записать все данные, запись которых была запрошена. Если количество байт, которые должны быть записаны, больше свободного пространства в канале, пишущий процесс остановится, пока читающий процесс не освободит достаточно места для записи. Ядро обеспечивает атомарность записи: если два или более процессов пишут в один канал, то система поставит их в очередь, так что запись следующего процесса в очереди начнётся только после того, как закончится запись предыдущего. Если читающий процесс закроет свой конец канала, все пишущие процессы получат сигнал SIGPIPE при попытке записи в этот канал. Это приведёт к прерыванию вызова write(2) и, если сигнал не был обработан, к завершению пишущего процесса.

lseek(2), mmap(2) Эти системные вызовы не допустимы, так как нет способа перечитать данные из канала; чтение разрушает их.

dup(2) Системный вызов dup(2) часто используется для перенаправления стандартного вывода и стандартного ввода на соответствующие концы программного канала.

poll(2) и select(3C). Эти вызовы часто используются для мультиплексирования ввода-вывода в каналы, если процессу необходимо работать с несколькими каналами или другими устройствами или псевдоустройствами, работа с которыми может привести к блокировке.

fcntl(2) Команды F_SETLK, F_GETLK и F_FREESP к каналам неприменимы, однако команды F_GETFD и F_SETFD часто используются для изменения флагов O_NDELAY и O_NONBLOCK. Обычно, чтение из пустого канала и запись в переполненный канал блокируются, как обсуждалось выше. Однако, вы имеете возможность выключить это свойство каналов установкой флагов O_NDELAY и O_NONBLOCK. При установленном O_NDELAY, read(2) при попытке читать из пустого канала немедленно возвратит 0. Если установлен O_NONBLOCK, чтение из пустого канала заставит read(2) возвратить -1 и установить errno в EAGAIN. Аналогично будет себя вести write(2) при записи в переполненный канал.

Почти все сказанное приложимо и к именованным каналам, обсуждаемым далее в этом разделе, за исключением того, что именованные каналы создаются и открываются другим способом.

Программные каналы - Пример

Этот пример показывает использование канала. Программа посылает текст порождённому процессу, который, в свою очередь, выводит текст, полученный от родителя. Эта программа будет работать на любой системе, совместимой со стандартом POSIX.

11-13 Здесь создаётся программный канал. Заметьте, что канал создаётся до запуска подпроцесса, иначе подпроцесс не смог бы унаследовать файловые дескрипторы.

16-18 После создания подпроцесса, родительский процесс пишет в канал текстовое сообщение.

19-22 Порождённый процесс читает сообщение из канала, и выводит его на терминал.

Обратите внимание, что в этом примере ненужные концы канала не закрываются. В данном случае это не представляет опасности, так как читающий процесс не ожидает конца файла. Поэтому, для упрощения программы, вызовы close(2) в этой программе пропущены.

Эта программа работает так:

```
$ pipe1
Hello, world
```

Файл: pipe1.c

ПРОГРАММНЫЕ КАНАЛЫ - ПРИМЕР

```
 1 #include <sys/types.h>
 2 #include <unistd.h>
 3 #define   MSGSIZE   20
 4
 5 main(int argc, char **argv)
 6 {
 7     int fd[2]; pid_t pid;
 8     static char msgout[MSGSIZE]="Hello,world";
 9     static char msgin[MSGSIZE];
10
11     if (pipe(fd) == -1) {
12          perror(argv[0]);
13          exit(1);
14     }
15
16     if ((pid=fork()) > 0) {  /* parent */
17          write(fd[1], msgout, MSGSIZE);
18     }
19     else if (pid == 0) {      /* child */
20          read(fd[0], msgin, MSGSIZE);
21          puts(msgin);
22     }
23     else {          /* cannot fork */
24          perror(argv[0]);
25          exit(2);
26     }
27
28     exit(0);
29 }
```

Программные каналы - Пример

Этот пример иллюстрирует двунаправленные каналы, использующие реализацию pipe(2) в SVR4. Эта программа посылает текст порождённому процессу, который распечатывает текст, полученный от родительского процесса. Порождённый процесс также посылает сообщение родителю через тот же самый канал, из которого он прочитал сообщение от родителя.

12-14 Здесь создаётся программный канал. Заметьте, что канал создаётся до запуска подпроцесса.

16-22 После запуска подпроцесса, родитель пишет в канал сообщение, ожидает сообщения из того же самого конца канала, и пишет полученное сообщение на терминал.

23-29 Аналогичным образом, порождённый процесс пишет сообщение в свой конец канала и ждёт сообщения из него.

Эта программа работает так:

```
$ pipe1.SVR4
Parent hears: I want ice cream

$ Child hears: Eat your spinach

$
```

Файл: pipe1.SVR4.c

ПРОГРАММНЫЕ КАНАЛЫ - ПРИМЕР

```
1   #include <sys/types.h>
2   #include <unistd.h>
3   #define   MSGSIZE   20
4
5   main(int argc, char **argv)
6   {
7       int fd[2]; pid_t pid;
8       static char msgout1[MSGSIZE] = "I want ice cream\n";
9       static char msgout2[MSGSIZE] = "Eat your spinach\n";
10      static char msgin[MSGSIZE];
11
12      if (pipe(fd) == -1) {
13          perror(argv[0]); exit(1);
14      }
15
16      if ((pid = fork()) > 0) {       /* parent */
17          if(write(fd[1], msgout2, MSGSIZE) == -1)
18              perror("Parent write");
19          if(read(fd[1], msgin, MSGSIZE) == -1)
20              perror("Parent read");
21          printf("Parent hears: %s\n", msgin);
22      }
23      else if(pid == 0)  {            /* child */
24          if(write(fd[0], msgout1, MSGSIZE) == -1)
25              perror("Child write");
26          if(read(fd[0], msgin, MSGSIZE) == -1)
27              perror("Child read");
28          printf("Child hears: %s\n", msgin);
29      }
30      else {      /* cannot fork */
31          perror(argv[0]);
32          exit(2);
33      }
34      exit(0);
35  }
```

Программные каналы - Пример - who | sort

В этом примере создаются два подпроцесса, взаимодействующие через канал. Каждый из этих процессов исполняет свою программу, а родительский процесс ждет их завершения.

9-11 До запуска подпроцессов создаётся канал.

12-20 Здесь создаётся первый подпроцесс. Его стандартный вывод перенаправляется на входной конец канала. Весь вывод в дескриптор файла 1 будет направлен в канал. Это относится к printf(3C) так же, как и к любому выводу программы, запущенной системным вызовом exec(2). Затем закрываются все неиспользуемые дескрипторы канала. Затем исполняется команда who(1). fflush(stdout) делается, чтобы быть уверенным, что буферизованный текст будет выведен перед вызовом exec(2), а не потерян.

21-30 Создаётся второй подпроцесс, и его стандартный ввод перенаправляется на выходной конец канала. Стандартный ввод, дескриптор файла 0, этого процесса, теперь будет идти из выходного конца канала. Здесь также закрываются все неиспользуемые дескрипторы канала. Это необходимо, так как sort(1) не может завершить сортировку и не может выдать данные, пока не увидит конец файла во входном потоке. Функция read_to_nl(), не показанная здесь, читает из файлового дескриптора 0, пока не получит символ конца строки. Эта функция, как и любая программа, запущенная exec(2), будет получать ввод из канала.

28 fflush(stdout) нужен только если вывод who(1) перенаправлен. Вывод на терминал буферизуется по строкам.

31 Родительский процесс должен закрыть оба конца канала, если не использует его. Если входной конец закрыт, то читающий процесс, sort(1) в этом примере, увидит конец файла, когда единственный пишущий процесс who(1) закроет свой стандартный вывод. Иначе читающий процесс будет висеть, ожидая ввода, который никогда не поступит. Выходной конец закрывается для удобства пишущего процесса who(1), который получит SIGPIPE при попытке записи, если все читающие процессы закрыли выходной конец, например, в результате аварийного завершения.

32-33 Родительский процесс ожидает завершения обоих порождённых, но выходной статус завершившегося процесса игнорируется.

Эта программа работает так:

```
$ whos3
Heading: who display sorted
bern      tty26     May  8    18:14
console   console   May  5    22:47
ipd       tty31     May  8    07:46
```

Файл: whos3.c

ПРОГРАММНЫЕ КАНАЛЫ - ПРИМЕР -
who | sort

```
 1  #include <sys/types.h>
 2  #include <stdio.h>
 3  #include <unistd.h>
 4  #include <wait.h>
 5  char text[80];
 6  main(int argc, char **argv)
 7  {
 8      int fd[2]; void read_to_nl(char *);
 9      if (pipe(fd) == -1) {
10          perror(argv[0]); exit(1);
11      }
12      if (fork() == 0) { /* first child */
13          close(1);
14          dup(fd[1]); /* redirect std output */
15          close(fd[0]); close(fd[1]);
16          printf("who display sorted\n");
17          fflush(stdout);
18          execl("/bin/who", "who", (char *) 0);
19          exit(127);
20      }
21      if (fork() == 0) { /* second child */
22          close(0);
23          dup(fd[0]); /* redirect std input */
24          close(fd[0]); close(fd[1]);
25          read_to_nl(text);
26          printf("\tHeading: %s\n", text);
27          fflush(stdout);
28          execl("/bin/sort","sort", (char *)0);
29          exit(127);
30      }
31      close(fd[0]);close(fd[1]);
32      while (wait((int *) 0) != -1)
33          ;       /* null */
34      exit(0);
35  }
```

Функции стандартной библиотеки для работы с каналами

Библиотечные функция popen(3S) для работы с каналами аналогична fopen(3) для работы с обычными файлами. Вместо имени файла вы передаете командную строку shell как один аргумент. Так же, как fopen(3), popen(3S) возвращает указатель на тип FILE. Затем стандартные библиотечные функции, такие, как printf, используются для ввода и вывода. Наконец, когда вы закончите, вы закрываете указатель на файл с использованием pclose(3) вместо fclose(3).

В большинстве случаев проще использовать popen(3) и pclose(3), чем системные вызовы.

Аргументы popen(3):

command - указатель на строку символов, содержащую любую правильную команду интерпретатора shell. Эта командная строка может использовать все механизмы shell, такие как поиск команд в PATH, расширение метасимволов, перенаправление ввода/вывода и т.д.. Это работает потому, что popen(3) исполняет shell и передает ему эту командную строку как аргумент. В качестве shell используется значение переменной среды SHELL, и /bin/sh, если эта переменная не определена.

type - "r" для чтения и "w" для записи. Выбор "r" или "w" определяется тем, как программа будет использовать полученный указатель на файл.

. type должен быть "r", если программа хочет читать данные, выдаваемые command в стандартный вывод.

. type должен быть "w", если вывод в полученный указатель на файл должен быть стандартным вводом command.

Возвращаемое значение popen(3) - указатель на стандартный библиотечный тип FILE.

Аргументы pclose(3) таковы:

stream - указатель на FILE, полученный при вызове popen(3).

Главное отличие pclose(3) от fclose(3) состоит в том, что pclose(3) ожидает завершения созданного процесса. Запоминание pid созданного процесса в разных системах осуществляется по разному; в Linux, в структуре FILE для этого предусмотрено специальное поле. В Solaris, в структуре FILE подходящего поля нет, библиотека создаёт отдельный список pid, ассоциированных со структурами FILE.

pclose(3) возвращает выходной статус команды, когда она завершается.

Функции стандартной библиотеки для работы с каналами - Иллюстрация

Традиционные библиотечные функции для работы с каналами - popen(3) и pclose(3). SVR4 добавляет функции p2open(3G) и p2close(3G). Их действия аналогичны, но popen(3S) создает однонаправленную линию связи, а p2open(3G) - двунаправленную линию для связи между родительским и порожденным процессами.

Сначала popen(3) создает канал, затем порождает процесс, исполняющий команду, заданную первым аргументом, и перенаправляет ее стандартный ввод или вывод. Соответственно, pclose(3) закрывает канал и ждет завершения порожденного процесса.

Используя popen(3) и pclose(3), программа может исполнить любую допустимую команду, которая может быть набрана в командной строке вашего интерпретатора shell. Затем программа может читать со стандартного вывода этой команды или писать в ее стандартный ввод.

Функция popen(3) реализована с использованием pipe(2) для создания канала, fork(2) для запуска подпроцесса и dup(2) для перенаправления стандартного ввода или вывода в канал. Функция pclose(3), в свою очередь, использует close(2), чтобы закрыть канал, и waitid(2) или waitpid(2) для того, чтобы дождаться завершения порождённого процесса. На диаграмме пунктирная линия показывает, что родительский процесс, ожидает завершения своего подпроцесса.

Библиотечные функции - Пример - who | sort

Этот пример демонстрирует использование popen(3) и pclose(3) для реализации who | sort.

5 Эта строка описывает указатели на входной и выходной файлы.

6 Определяется буфер ввода и вывода в виде массива символов. Макрос BUFSIZ определен в файле stdio.h.

8-9 Здесь создаются указатели файлов ввода и вывода.

11-12 Здесь символьные данные копируются из вывода команды who(1) на ввод команды sort(1). Цикл ввода/вывода прекращается, когда больше нет входных данных. Это происходит, когда завершается who(1).

14-15 Наконец, закрываются указатели на файлы. Порядок этих закрытий важен, потому что sort(1) начинает сортировку только после того, как увидит конец стандартного ввода.

Заметьте, насколько эта программа проще предыдущего примера, использующего системные вызовы. Также заметьте, что при помощи библиотечных функций мы не можем создать конвейер, напрямую соединяющий запущенные процессы, и вынуждены сами копировать данные из вывода одной команды на ввод другой. Впрочем, при использовании popen(3C) можно было бы возложить создание такого конвейера на shell, вызвав popen("who|sort", "r");

Эта программа работает так:

```
$ whos1
anil      tty41     May  8    08:42
bern      tty26     May  8    18:14
bern      xt082     May  8    14:39
console   console   May  5    22:47
ipd       tty31     May  8    07:46
```

Файл: whos1.c
БИБЛИОТЕЧНЫЕ ФУНКЦИИ - ПРИМЕР - who | sort

```
 1  #include <stdio.h>
 2
 3  main()
 4  {
 5  FILE *fpin, *fpout;
 6  char line[BUFSIZ];
 7
 8  fpin = popen("who", "r");
 9  fpout = popen("sort", "w");
10
11   while(fgets(line,BUFSIZ,fpin) != (char *)NULL)
12   fputs(line, fpout);
13
14   pclose(fpin);
15   pclose(fpout);
16   }
```

Стандартные библиотечные функции для работы с каналами

SVR4 предоставляет p2open(3G), который похож на popen(3S) тем, что запускает в порожденном процессе shell-команду и устанавливает связь с этим процессом через канал. Он отличается от popen(3S) тем, что предоставляет двунаправленную связь. Перенаправляются как стандартный ввод, так и стандартный вывод запущенного процесса.

p2open(3G) вызывает fork(2) и исполняет командную строку, на которую указывает cmd. При возврате, ptr[0] содержит указатель на FILE, запись в который будет стандартным вводом команды. ptr[1] содержит указатель на файл, который может быть использован для чтения стандартного вывода команды.

p2close(3G) закрывает оба указателя на FILE. Если программист хочет закрыть только один из потоков, возможно, следует использовать fclose(3C). Это необходимо, если команда должна увидеть конец файла перед тем, как начать обработку ввода, как sort(1).

Если команда при закрытии входного потока только начинает обработку ввода, мы не можем читать её вывод до закрытия её ввода. Если для закрытия ввода использовать pclose(3C), эта функция будет ждать завершения команды. Но, поскольку мы ещё не начинали читать вывод, команда может быть заблокирована из-за переполнения трубы на выходе. Таким образом, и pclose(3C), и запущенная команда в такой ситуации могут никогда не завершиться. В этой ситуации необходимо закрывать поток при помощи fclose(3C).

Как правило, нужно сначала закрывать поток ввода команды вызовом fclose(3C), потом считывать данные из потока вывода команды и только потом закрывать этот поток вызовом pclose(3C). Но точные требования к порядку закрытия потоков следует определять в зависимости от особенностей поведения запускаемой команды.

Библиотечные функции для работы с каналами - Пример - p2open(3G)

Эта программа демонстрирует использование p2open(3G) для двунаправленной связи между родительским и порождённым процессами.

Родительский процесс пишет в стандартный ввод порождённого через fptr[0] и читает его стандартный вывод из fptr[1]. В этой программе важно, что родитель делает fclose(3S) для того файла, в который он писал, так что команда sort(1) исполняющаяся в подпроцессе, увидит конец файла.

Эта программа работает так:

```
$ p2exam
Sorted line 0: zfk
Sorted line 1: sdcjjden
Sorted line 2: njkdnk
Sorted line 3: ldef
Sorted line 4: bfifim
```

Файл: p2exam.c
БИБЛИОТЕЧНЫЕ ФУНКЦИИ ДЛЯ РАБОТЫ С КАНАЛАМИ - ПРИМЕР - p2open(3G)

```
 1      #include <stdio.h>
 2      #include <libgen.h>
 3
 4      main()
 5      {
 6      FILE *fptrs[2];
 7      int i, pid;
 8      char buf[79];
 9      char *lines[5] = {"njkndk\n",
10       "sdcjjden\n",
11       "ldef\n",
12       "bfifim\n",
13       "zfk\n" };
14
15
16       p2open("/bin/sort -r", fptrs);
17
18       for( i=0; i < 5; i++)
19       fputs(lines[i], fptrs[0]);
20       fclose(fptrs[0]);
21
22       i = 0;
23       while(fgets(buf, 80, fptrs[1]) != NULL) {
24       printf("Sorted line %d: %s", i++, buf);
25       }
26       }
```

Именованные каналы - Введение

В этом разделе вы изучите именованные каналы. Именованные каналы также известны как FIFO-файлы. Процессы, взаимодействующие через неименованный канал, должны быть родственными, иначе они не смогут получить друг от друга файловые дескрипторы канала.

Именованные каналы, в отличие от неименованных, могут использоваться неродственными процессами. Они дают вам, по сути, те же возможности, что и неименованные каналы, но с некоторыми преимуществами, присущими обычным файлам. Именованные каналы используют специальный файл для управления правами доступа. Имя такого файла может быть размещено в любом месте дерева файловой системы ОС UNIX, при условии, что файловая система поддерживает файлы такого типа (большинство файловых систем Unix, такие, как UFS и ZFS, поддерживают FIFO-файлы, но FAT16/FAT32 (pcfs) их не поддерживают). Это позволяет неродственным процессам взаимодействовать через канал, если они имеют соответствующие права доступа к файлам. Именованные каналы существуют независимо от любых процессов, но, в отличие от файлов, хранящиеся в них данные не переживают перезагрузку системы.

Пример иллюстрирует использование именованных каналов из командной строки shell на двух различных терминалах. Представьте себе такую ситуацию: вы имеете терминальный класс, в котором в качестве одного из терминалов используется телетайп. Первый человек, вошедший в класс, входит в систему и исполняет команды, приведённые в верхней части иллюстрации. Первая команда создаёт в текущей директории именованный канал NP.

Команда mknod(1) использует системный вызов mknod(2) для создания именованного канала. chmod ug+w NP дает права записи в этот файл любому процессу того же пользователя или любого пользователя из той же группы. line < NP читает одну строку из своего перенаправленного стандартного ввода, т.е. из именованного канала NP, и выводит прочитанное на свой стандартный вывод. Если никто ещё не открыл именованный канал на запись, line(1) будет спать, ожидая ввода от других процессов.

Теперь на одном из экранных терминалов командой cat(1) содержимое целого файла копируется в именованный канал. line(1) на печатающем терминале просыпается, как только cat(1) открывает именованный канал на запись; затем она читает первую строку из канала и печатает ее на свой стандартный вывод.

Если в буфере канала находятся данные, это можно обнаружить по тому, что такой канал имеет ненулевую длину. Это можно показать, если вызвать команды line и ls -l из порожденного интерпретатора shell, который получает ввод из NP. Команда ls не использует стандартный ввод; ее присутствие нужно только для того, чтобы показать, что 4 строки, содержащие 108 символов, остаются в канале до тех пор, пока читающий процесс активен. Когда процесс завершится, в именованном канале не будет ни одного байта.

Содержимое файла data таково:

```
$ cat data
ABCDEFGHIJKLMNOPQRSTUVWXYZ
ABCDEFGHIJKLMNOPQRSTUVWXYZ
ABCDEFGHIJKLMNOPQRSTUVWXYZ
ABCDEFGHIJKLMNOPQRSTUVWXYZ
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

на печатающем терминале:

```
$ /etc/mknod NP p
$ chmod ug+w NP
$
$ #  NOTE "line < NP"  reads first line of NP
$ (line; ls -l NP) <NP
ABCDEFGHIJKLMNOPQRSTUVWXYZ
prw-rw----   1 tmm     unixc   108 Sep 19 13:55 NP
$
$ ls -l NP
prw-rw----   1 tmm     unixc     0 Sep 19 13:56 NP
```

на экранном терминале:

```
$ ls -l NP
prw-rw----   1 tmm     unixc     0 Sep 19 13:54 NP
$ cat data >NP
```

+

Дополнительная информация (на случай, если не все обучаемые хорошо знают shell): если вы хотите прочитать все строки из именованного канала, по одной за раз, команда line(1) должна использоваться в цикле, а ввод этого цикла должен быть перенаправлен из именованного канала.

Создание именованных каналов

Системный вызов mknod(2) используется для создания директории, именованного канала, символьного или блочного специального файла или обычного файла. Только именованные каналы могут быть созданы обычным пользователем; файлы остальных типов создаются только суперпользователем.

Аргументы системного вызова mknod(2):

path - указатель на имя создаваемого файла и путь к нему

mode - тип файла и права доступа к нему. Ненулевое значение cmask может исключить некоторые из заданных прав.

dev - зависящие от конфигурации данные для специального файла. Для именованных каналов этот параметр не используется и может быть нулевым.

Возвращаемое значение показывает успех или неудачу.

Обратите внимание, что в параметре mode используется не только младшие 12 бит, как в аналогичном параметре open(2) и chmod(2), а все 16. Старшие 4 бита mode кодируют тип создаваемого инода, как и в поле st_mod структуры stat; open(2) и chmod(2) не позволяют задавать и изменять тип инода, а mknod(2) требует его указать.

Пример: следующий оператор создаёт в текущей директории именованный канал NP с правами доступа rw-rw----. Константа S_IFIFO из <sys/stat.h> имеет значение 010000, что означает файл типа именованного канала, известный также как специальный FIFO-файл. Для использования S_IFIFO, перед <sys/stat.h> нужно включить <sys/types.h>, потому что <sys/stat.h> использует многие из typedef, определенных в этом файле.

```
#include <sys/types.h>
#include <sys/stat.h>

mknod("NP", S_IFIFO | 0660, 0);
```

Именованные каналы могут удаляться с помощью rm(1) или unlink(2).

Именованный канал может быть создан в любой момент до его первого использования и удалён в любое время после этого.

В SVR4 добавлена библиотечная функция mkfifo(3C), эквивалентная mknod(2), где параметр mode включает флаг S_IFIFO.

Не следует путать FIFO-файлы с сокетами Unix (Unix domain sockets). Сокеты Unix, впервые введённые в BSD Unix и поддерживаемые большинством современных Unix-систем, в том числе и SVR4, в ряде отношений похожи на FIFO-файлы. И FIFO-файлы, и сокеты Unix имеют имя в файловой системе, могут использоваться для коммуникации неродственных процессов и выдают файловый дескриптор, с которым можно использовать вызовы read(2), write(2) и select(3C). Также порядок закрытия сокетов и FIFO-файлов во многом аналогичен. Однако сокеты имеют другой тип инода (S_IFSOCK) и процедура их создания и открытия отличается от процедуры создания и открытия FIFO-файлов.

Процедура создания и открытия сокетов Unix описана на странице руководства socket(3SOCKET). На этой же странице описана и процедура создания сетевых сокетов. В нашем курсе сокеты Unix не рассматриваются.

Особенности системных вызовов

Следующие системные вызовы имеют особенности при работе с именованными каналами:

open(2) Именованный канал открывается так же, как и обычный файл, но с дополнительными особенностями. Как правило, если вы открываете именованный канал для чтения, системный вызов open(2) будет ожидать, пока какой-то другой процесс не откроет этот же канал для записи, и наоборот.

Однако, если вы открыватете именованный канал для чтения с установленными флагами O_NDELAY или O_NONBLOCK, open(2) немедленно вернёт вам правильный дескриптор файла, даже если ни один процесс ещё не открыл его для записи. Если вы открываете канал на запись, но никто ещё не открыл его для чтения, open(2) с флагами O_NDELAY или O_NONBLOCK вернет код неудачи.

close(2) Этот вызов используется обычным образом и закрывает именованный канал, так же как и любой файл, когда закончена работа с ним. Ниже описано, как он действует на read(2) и write(2).

read(2) Этот системный вызов читает столько данных, сколько на момент вызова есть в канале. Если количество байт в канале меньше, чем требуется, read(2) возвращает значение меньшее, чем его последний аргумент. read(2) возвращает 0 если обнаруживает, что другой конец канала был закрыт. Если канал пуст, но файловый дескриптор другого конца ещё открыт), read(2) будет заблокирован.

write(2) Этот системный вызов ведёт себя аналогично записи в неименованный канал. Если читающий процесс закроет свой конец канала, пишущий процесс получит сигнал SIGPIPE при попытке записи в этот канал.

lseek(2), mmap(2) Этот системный вызов не работает, так как нет способа перечитать данные из канала; чтение разрушает их.

dup(2) Этот системный вызов практически не используется с именованными каналами

poll(2) и select(3C). Эти вызовы часто используются для мультиплексирования ввода-вывода в каналы, если процессу необходимо работать с несколькими каналами или другими устройствами или псевдоустройствами, работа с которыми может привести к блокировке.

fcntl(2) Ведет себя аналогично неименованным каналам. Используется, главным образом, для изменения флагов O_NDELAY и O_NONBLOCK.

У именованных каналов, флаги O_NDELAY и O_NONBLOCK можно было бы установить при открытии, но у именованных каналов эти флаги перегружены: они влияют как на поведение open(2), так и на поведение read(2)/write(2), причём влияют по разному. Поэтому вы можете захотеть использовать один из этих флагов при open(2), но не при работе с каналом, или наоборот. В обоих случаях, перед чтением и записью, флаг необходимо установить в требуемое значение при помощи fcntl(2).

Именованные каналы - Пример - Схема

Пример на следующей иллюстрации показывает полезное приложение именованных каналов. Пример состоит из процесса файлового сервера и процесса-клиента. Здесь серверный процесс ожидает, пока ему через общедоступный именованный канал не передадут имя обычного файла и имя личного именованного канала, созданного клиентом для получения содержимого требуемого файла. Клиент распечатывает содержимое обычного файла, полученное от сервера.

server.h:

```
1 struct message {
2 char privfifo[15];/* name of private named pipe */
3 char filename[100];/* name of requested file */
4 };
5
6 #define  PUBLIC  "Public"/* name of public named pipe */
7 #define  LINESIZE  512
8 #define  NUMTRIES  3
```

Именованные каналы - Пример - Клиент

Программа-клиент создает личный именованный канал, посылает имена этого канала и требуемого файла файловому серверу через общедоступный именованный канал. Затем она распечатывает содержимое файла, полученное через личный именованный канал.

1-2 Эти файлы описывают различные типы файлов для системного вызова mknod(2).

13-17 Синтезируется имя личного программного канала, и создаётся сам этот канал с правами чтения и записи для всех процессов.

18-23 Общедоступный именованный канал открывается на запись, и в него записываются имена личного канала и требуемого файла.

25-27 Личный именованный канал открывается для чтения.

29-30 Распечатываются данные, полученные из личного канала.

31-32 Личный именованный канал закрывается и удаляется из текущей директории.

Заметьте, что сервер и клиент должны соблюдать соглашение о формате данных, записываемых в общедоступный канал.

Файловый сервер работает таким образом:

```
$ server &
$ client data
ABCDEFGHIJKLMNOPQRSTUVWXYZ
ABCDEFGHIJKLMNOPQRSTUVWXYZ
ABCDEFGHIJKLMNOPQRSTUVWXYZ
ABCDEFGHIJKLMNOPQRSTUVWXYZ
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

Файл: client1.c

ИМЕНОВАННЫЕ КАНАЛЫ - ПРИМЕР - КЛИЕНТ

```
1   #include <sys/types.h>
2   #include <sys/stat.h>
3   #include <fcntl.h>
4   #include <sys/uio.h>
5   #include "server.h"
6
7   main(int argc, char **argv)/* client process */
8   {
9   struct message msg;
10   int n, fdpub, fdpriv;
11   char line[LINESIZE];
12
13   sprintf(msg.privfifo, "Fifo%d", getpid());
14   if(mknod(msg.privfifo,S_IFIFO | 0666,0) == -1) {
15   perror(msg.privfifo);
16   exit(1);
17   }
18   if ((fdpub = open(PUBLIC, O_WRONLY)) == -1) {
19   perror(PUBLIC);
20   exit(2);
21   }
22   strcpy(msg.filename, argv[1]);
23   write(fdpub, (char *) &msg, sizeof(msg));
24
25   if((fdpriv = open(msg.privfifo,O_RDONLY)) == -1) {
26   perror(msg.privfifo);
27   exit(3);
28   }
29   while ((n = read(fdpriv, line, LINESIZE)) > 0)
30   write(1, line, n);
31   close(fdpriv);
32   unlink(msg.privfifo);
33   exit(0);
34   }
```

Именованные каналы - Пример - Файловый сервер

В struct message содержатся имя личного именованного канала вместе с именем требуемого файла.

Программа файлового сервера в этом примере открывает на чтение общедоступный именованный канал. Он спит до тех пор, пока какой-нибудь процесс-клиент не откроет другой конец этого канала. Тогда сервер читает из общедоступного канала имена личного канала и требуемого файла в соответствующие поля структуры. Затем, сервер открывает личный канал и требуемый файл. Содержимое файла копируется в канал, и файл закрывается. Затем, закрыв общедоступный канал, программа возвращается назад, снова открывает общедоступный канал и ждет обращения следующего клиента.

9 Здесь определяется структура сообщения, которое читается из общедоступного именованного канала.

14-17 Общедоступный канал открывается для чтения. Системный вызов open(2) блокируется, если никакой процесс-клиент не открывает другой конец канала.

18-19 Из общедоступного канала читается сообщение, содержащее имена личного канала и требуемого файла.

20-23 Требуемый файл открывается для чтения.

24 Личный канал открывается для записи. Файловый сервер спит, пока клиент не откроет свой конец личного канала. Если возникли проблемы, и клиент не может открыть канал, сервер повиснет внутри этого open(2). Решение этой проблемы будет показано в следующем примере.

28-29 Данные из файла копируются в личный именованный канал.

30-31 Когда копирование закончено, требуемый файл и личный именованный канал закрываются.

33-34 Общедоступный канал также закрывается, и сервер переходит к следующему витку цикла. Причина для закрытия и повторного открытия состоит в том, что хочется, чтобы файловый сервер спал на open(2), ожидая запросов.

Заметьте, что программа никогда не завершается.

Файл: server1.c

ИМЕНОВАННЫЕ КАНАЛЫ - ПРИМЕР - ФАЙЛОВЫЙ СЕРВЕР

```
1   #include <sys/types.h>
2   #include <fcntl.h>
3   #include <sys/stat.h>
4   #include "server.h"
5
6   main(int argc, char **argv)/* server process */
7   {
8       int fdpub, fdpriv, fd;
9       struct message msg;
10       int n;
11      char line[LINESIZE];
12
13      loop:/* forever */
14      if ((fdpub = open(PUBLIC, O_RDONLY)) == -1) {
15          perror(PUBLIC);
16          exit(1);
17      }
18      while (read(fdpub, (char *) &msg,
19              sizeof(msg)) > 0) {
20          if ((fd = open(msg.filename, O_RDONLY)) == -1) {
21              perror(msg.filename);
22              break;
23          }
24          if ((fdpriv = open(msg.privfifo, O_WRONLY)) == -1) {
25              perror(msg.privfifo);
26              break;
27          }
28          while ((n = read(fd, line, LINESIZE)) > 0)
29              write(fdpriv, line, n);
30          close(fd);
31          close(fdpriv);
32      }
33      close(fdpub);
34      goto loop;
35  }
```

Именованные каналы - Пример - Файловый сервер без блокировки

В этой программе файловый сервер модифицирован так, чтобы решить проблему возможного зависания в случае, если сервер открывает личный именованный канал, а клиент не открывает свой конец этого канала. Для этого личный канал открывается с флагом O_NDELAY. Теперь сервер пытается открыть личный канал несколько раз и сдается после заданного числа неудач.

24-30 Здесь личный именованный канал открывается для записи. Если клиент не открыл свой конец канала для чтения, open(2) возвращает неудачу, и сервер делает следующую попытку после паузы в одну секунду. Цикл завершается, если канал был открыт, или после заданного числа неудач.

31-34 Если открытие личного именованного канала не удалось, программа возвращается к началу цикла.

Файл: server2.c

ИМЕНОВАННЫЕ КАНАЛЫ - ПРИМЕР - ФАЙЛОВЫЙ СЕРВЕР БЕЗ БЛОКИРОВКИ

```
 1   #include <sys/types.h>
 2   #include <fcntl.h>
 3   #include <sys/stat.h>
...
13    loop:/* forever */
14    if ((fdpub = open(PUBLIC, O_RDONLY)) == -1) {
15        perror(PUBLIC);
16        exit(1);
17    }
18    while (read(fdpub, (char *) &msg,
19            sizeof(msg)) > 0) {
20        if ((fd = open(msg.filename, O_RDONLY)) == -1) {
21            perror(msg.filename);
22            break;
23        }
24        for (i = 0; i < NUMTRIES; i++) {
25            if ((fdpriv = open(msg.privfifo, O_WRONLY |
26                    O_NDELAY)) == -1)
27                sleep(1);
28            else
29                break;
30        }
31        if (fdpriv == -1) {
32            perror(msg.privfifo);
33            break;
34        }
35        while ((n = read(fd, line, LINESIZE)) > 0)
36            write(fdpriv, line, n);
37        close(fd);
38        close(fdpriv);
39    }
40    close(fdpub);
41    goto loop;
42  }
```

11. System V IPC

Введение

В UNIX System V появились три новых средства межпроцессного взаимодействия (IPC) — семафоры, разделяемая память и очереди сообщений. В других Unix-системах и в стандарте POSIX существуют и другие средства IPC — семафоры и разделяемая память Xenix, семафоры и мутексы Solaris, семафоры и мутексы POSIX Threads и др. Все перечисленные средства IPC требуют использования различных API. Поскольку обсуждаемые средства появились именно в UNIX System V, для них прижилось название System V IPC. Эти средства стали частью стандарта POSIX и поддерживаются большинством современных Unix-систем.

После принятием стандарта POSIX Threads, средства System V IPC, особенно семафоры, считаются устаревшими (семафоры, мутексы и некоторые другие примитивы POSIX Threads можно использовать для синхронизации не только нитей одного процесса, но и для межпроцессного взаимодействия). Однако System V IPC до сих пор считается более распространенным интерфейсом, и много старых приложений написано с использованием этого API, поэтому знание System V IPC для современных разработчиков всё еще актуально.

Средства System V IPC были разработаны в 1980е годы в ответ на пожелания разработчиков прикладных программ, занимавшихся переносом на Unix приложений из IBM MVS, DEC VAX/VMS и др. Поэтому средства System V IPC не полностью соответствуют основной философии Unix. Это глобальные объекты, не имеющие имен в файловой системе, идентифицируемые, создаваемые и уничтожаемые иными способами, чем файлы.

В этом разделе обсуждаются:

. Свойства, общие для этих средств

. Генерация ключа для получения доступа к каждому из средств

. Использование команд IPC

Для работы с разделяемой памятью, семафорами и очередями сообщений используются схожие системные вызовы. Это сходство обсуждается в первую очередь, чтобы обеспечить лучшее понимание индивидуальных свойств этих средств, которые будут формально описаны далее.

Средства System V IPC

System V IPC включает в себя семафоры, разделяемую память и очереди сообщений.

Семафоры используются для синхронизации процессов и управления ресурсами. Например, семафор может быть использован для управления доступом к устройству, такому как принтер. Семафор может гарантировать, что с принтером в данный момент работает только один процесс. Это защитит от перемешивания вывода нескольких процессов на печать.

Разделяемая память — наиболее быстрый способ IPC. Разделяемая память может принадлежать более чем одному процессу. С момента присоединения, разделяемая память становится частью области данных процесса. Как только этот сегмент памяти модифицируется, новые данные становятся доступны всем процессам, присоединённым к нему. Разделяемая память может использоваться для хранения общей для нескольких процессов информации, такой как таблицы поиска или критерии правильности данных.

Очереди сообщений позволяют процессам посылать дискретные блоки данных, называемые сообщениями. Посылающий процесс присваивает каждому сообщению тип. Получатель может избирательно читать сообщения из очереди, основываясь на этом типе. В частности, он может получить первое сообщение в очереди (независимо от типа), первое сообщение данного типа или первое сообщение любого типа из нескольких.

Структура API System V IPC

В разделах системного руководства (программа man) описываются системные вызовы трех типов: get, ctl и op для каждого из типов средств IPC.

get: Для каждого типа средств IPC в операционной системе существует отдельная таблица дескрипторов. Системные вызовы семейства get (semget, shmget и msgget) используются для выделения индекса в соответствующей таблице. Они возвращают идентификатор, основанный на индексе дескриптора в таблице. Этот идентификатор используется большинством остальных вызовов, работающих с данным средством.

Системные вызовы get аналогичны open(2). Как и open(2), get может быть использован для получения доступа к существующему средству или для создания нового. Если вы создаете новое средство, вы должны задать права доступа, похожие на файловые. Так же как файл имеет хозяина и группу, средство IPC имеет создателя, хозяина (обычно совпадает с создателем) и их идентификаторы групп. id, возвращаемый вызовом get, похож на дескриптор открытого файла. Однако, в отличие от файлового дескриптора, этот id является глобальным идентификатором и может принимать большие значения.

ctl: Системные вызовы семейства ctl (semctl, shmctl и msgctl) имеют в качестве параметра командное слово, которое задает одну из следующих функций:

. получение информации о состоянии средства. Это похоже на stat(2) для файла.

. изменение информации, например хозяина, группы, доступа или других данных, относящихся к этому средству.

Эти атрибуты устанавливаются во время вызова get и могут изменяться позднее вызовами семейства ctl. Изменения может делать только хозяин средства или суперпользователь. Эта функция напоминает chown(2) и chmod(2).

. удаление средства.

Удаление обычно делается, когда ни одному из процессов, использовавших это средство, оно больше не нужно. Удаление средства должно производиться процессом с тем же эффективным идентификатором пользователя, что и у хозяина средства. Если средство не удалено, оно будет существовать до перезагрузки системы, уменьшая тем самым доступное количество ресурсов этого типа. Команда ipcrm(1), обсуждаемая ниже, может использоваться для удаления средства, если оно не было удалено использовавшей его программой. Удаление похоже на unlink(2) для файла.

op: Справочное руководство Solaris содержит разделы для semop, shmop и msgop. Существует системный вызов под названием semop(2), но вызовов с именами shmop и msgop нет. В разделе shmop(2) описаны вызовы shmat(2) и shmdt(2). Раздел msgop(2) содержит описание вызовов msgsnd(2) и msgrcv(2). Вызовы семейства op служат собственно для передачи данных между процессами. Для семафоров и очередей сообщений, op напоминает read(2) и write(2).

Общие свойства средств IPC

Средства IPC имеют следующие общие свойства с файлами:

1. Средство IPC должно быть создано, прежде чем его можно будет использовать. Средство может быть создано за несколько дней (если за эти дни система не перезагружалась) или часов до использования, или же это может сделать первая из программ, которая будет его использовать. Средства IPC могут также создаваться во время загрузки, перед переходом системы в многопользовательский режим.

2. Во время создания для средства устанавливаются хозяин и права доступа. Эти атрибуты могут быть изменены позже.

3. Изменения, сделанные процессом в средстве IPC сохраняются после завершения процесса. Это похоже на содержимое изменённого файла. Сообщения, помещённые в очередь, будут храниться там, пока какой-то процесс не извлечёт их оттуда. По умолчанию, значения семафоров сохраняются, когда процесс, использовавший эти семафоры, завершается; однако можно установить режим, когда результат операций над семафорами отменяется после завершения процесса (об этом будет рассказано более подробно, когда будут обсуждаться семафоры).

4. Каждое средство IPC является ресурсом. Существуют ограничения на количество средств IPC, выделенных пользователю, и общее количество таких средств в системе. Созданное средство IPC изымается из числа возможных ресурсов для своего создателя и для других пользователей. Только после удаления этот ресурс вновь становится доступным для других целей. Это похоже на то, как блоки данных файла становятся свободными только после удаления последней ссылки на инод этого файла. Средство IPC может быть удалено его создателем или его хозяином командой ipcrm(1) или соответствующим системным вызовом из программы.

Общие свойства средств IPC - (продолжение)

В разделе руководства (команда man) intro(2) содержится информация о структурах данных и правах доступа, лежащих в основе управления средствами IPC.

Показанная ниже struct ipc_perm является общим элементом упоминавшихся выше дескрипторов средств IPC. Эта структура и некоторые общие препроцессорные макросы описаны в <sys/ipc.h>. Этот файл использует несколько операторов typedef, находящихся в <sys/types.h>. Для каждого из средств IPC в директории /usr/include/sys имеются отдельные файлы: sem.h, shm.h и msg.h

```
struct ipc_perm {
        uid_t   uid;/* owner's user id */
        gid_t   gid;/* owner's group id */
        uid_t   cuid;/* creator's user id */
        gid_t   cgid;/* creator's group id */
        mode_t  mode;/* access modes */
        ulong   seq;/* slot usage sequence number */
        key_t   key;/* key */
        long    pad[4]; /* reserve area */
};
```

Поля seq и key управляются системными вызовами. Вызов get устанавливает значения для полей структуры ipc_perm, а ctl может быть использован для того, чтобы изменить их.

Видно, что, в отличие от файла, средство IPC имеет два идентификатора пользователя — владельца и создателя. Владелец может быть изменен, идентификатор создателя — нет. При этом, как владелец, так и создатель могут менять права и удалять средство. Собственно права кодируются младшими девятью битами поля mode, которые аналогичны правам доступа для файлов: чтение-запись-исполнение для хозяина-группы-остальных. Ни одно из средств доступа System V IPC не использует право исполнения, но соответствующие биты в маске зарезервированы, так что формат маски совпадает с файловой.

С каждым из средств IPC связаны параметры настройки, определяющие размеры таблиц и системные стартовые значения. В старых версиях Solaris, эти параметры устанавливались в файле /etc/system (system(4)) и для их изменения необходима была перезагрузка системы.

В Solaris 10 появилась возможность более гибкого управления квотами ресурсов, в том числе ресурсов System V IPC, при помощи утилиты prctl(1). Обсуждение этого вопроса выходит за рамки данного курса.

Замечание: в Solaris, средства System V IPC обслуживаются модулями ядра, загружаемыми по требованию. Поэтому, если с момента загрузки ОС средства System V IPC ни разу не использовались, в выводе sysdef(1M) соответствующие параметры могут быть не видны.

*get - основные сведения

Для использования средства IPC достаточно знать идентификатор и иметь соответствующие права доступа. Системные вызовы семейства get возвращают идентификатор id для соответствующего средства.

Первый аргумент всех вызовов семейства get имеет тип key_t. Этот параметр аналогичен имени файла, но представляет собой целое число. Перед созданием средства IPC, пользователь должен выбрать значение ключа. Рекомендованный способ генерации значений ключей, снижающий вероятность конфликтов с другими пользователями или приложениями, рассматривается далее в этом разделе. Для ключей зарезервировано специальное значение IPC_PRIVATE. В системе одновременно может присутствовать несколько однотипных средств с ключом IPC_PRIVATE. Ключи с остальными значениями являются уникальными идентификаторами средства IPC, то есть в системе не могут существовать несколько семафоров с одинаковыми значениями ключей. При этом, однако, могут существовать набор семафоров и сегмент разделяемой памяти с одинаковым ключом.

Вызовы *get возвращают идентификатор средства, id. Этот идентификатор также представляет собой целое число, но, в отличие от ключа, его значение выбирается системой. Диапазон значений id зависит от ОС; обычно это большие числа. id представляет собой уникальный идентификатор средства в системе; ни при каких обстоятельствах одновременно не могут существовать однотипные средства с одинаковыми id. Также, важно отметить, что id представляет собой глобальный идентификатор. Если вы получите id средства в одном процессе, то любой другой процесс, имеющий права, сможет работать с этим средством, используя этот id. Значение id может быть передано другому процессу любым способом, например, через разделяемый файл, через аргументы exec(2), через трубу и т. д. Таким образом, id аналогичен тому, что в Win32 называется global handle (для сравнения, файловые дескрипторы в Unix привязаны к процессу и, таким образом, аналогичны local handle Win32).

Системный вызов *get имеет параметр flg, аналогичный параметру flags вызова open(2). Если в flg задан флаг IPC_CREAT (этот флаг определён в <sys/ipc.h>) и процесс указал несуществующий ключ или значение IPC_PRIVATE, get пытается создать новое средство. Совместно с флагом IPC_CREAT в параметре flg следует включить (побитовым ИЛИ) восьмеричное число, задающее права доступа к вновь создаваемому средству. Как и для файлов, можно задать различные права для хозяина, группы и других пользователей.

Если необходимо обратиться к уже существующему средству, обычно устанавливают параметр flg в 0. Если два процесса выполняют get с установленным флагом IPC_CREAT, первый из них станет создателем средства, и его uid будет записан как cuid (uid создателя). Второй процесс не получит ошибки от get, но получит тот же id, что и создатель. Если процесс хочет устранить возможность доступа к уже созданному средству, в параметр flg необходимо включить IPC_EXCL (также из <sys/ipc.h>). IPC_EXCL похож на флаг O_EXCL, используемый в open(2). Если указан этот флаг и средство с указанным ключом существует, *get, вместо открытия существующего средства, вернёт ошибку.

Идентификаторы пользователя и группы создателя, так же как и права доступа, заданные get при создании средства, записываются в структуру ipc_perm для этого средства. Поле mode этой структуры содержит права доступа.

Для семафора, право чтения разрешает процессу получать состояние семафора вызовом semctl(2) и ожидать, пока семафор не станет нулевым. Право изменения (записи) для семафора разрешает процессу устанавливать или изменять значение семафора.

Для разделяемой памяти, право чтения разрешает чтение из разделяемого сегмента, а записи - запись в него.

Для очередей сообщений, право чтения требуется для выполнения msgrcv(2), а записи - для

msgsnd(2).

Параметры настройки системы и, возможно, административные квоты определяют ограничения числа средств для всей системы и для каждого из пользователей. Вызов get, пытающийся выйти за эти пределы, возвратит ошибку.

Получение ключа IPC

Для использования средств межпроцессного взаимодействия, пользователь должен задать ключ в качестве первого параметра get. Этот ключ должен быть уникален для средства IPC. Все процессы, желающие использовать то же средство IPC, должны задать тот же ключ.

ftok(3C) генерирует ключ, основываясь на имени доступного файла или директории и дополнительном символе ch. Этот системный вызов часто используется, когда неродственные процессы должны взаимодействовать через IPC. Каждый процесс вызывает ftok(3C) с теми же аргументами и получает тот же ключ. Затем каждый процесс использует этот ключ в системном вызове get.

Параметр path указывает на имя файла или директории. Предполагается, что ch уникально в данном проекте. Возвращается значение типа key_t. Это значение содержит байт ch в качестве старшего байта, младший байт номера устройства, на котором находится заданный файл, в качестве следующего байта, и номер инода файла в качестве двух младших байтов.

Параметр path является указателем на строку символов - имя файла, к которому пользователь может иметь доступ. Доступность предполагает право поиска в директории, содержащей файл, но не обязательно право читать или писать в сам файл.

Параметр ch представляет собой одиночный символ. Все, что делает ftok(3C) - это генерация значения key_t для дальнейшего использования в get. Это значение может быть получено другим способом. Какой бы метод не был выбран, ключи, используемые для разных средств не должны совпадать.

*ctl - основные сведения

Системные вызовы семейства ctl используют id, полученный при вызове *get, в качестве первого аргумента. Они предоставляют три функции, общие для всех средств IPC.

Значение параметра cmd показывает, что должно быть сделано. Эти действия, определенные в <sys/ipc.h>, таковы:

IPC_STAT получает информацию о состоянии, содержащуюся в дескрипторе для соответствующего id. Для выполнения IPC_STAT требуется право чтения.

Замечание: Только хозяин или создатель средства IPC могут выполнять действияи IPC_SET и IPC_RMID.

IPC_SET изменяет хозяина, группу или права доступа, заданные при создании.

IPC_RMID удаляет средство.

Существуют и другие действия, различные для разных типов средств IPC. Системный вызов semctl(2) имеет несколько таких действий.

*op - основные сведения

Системные вызовы семейства op, кроме shmdt(2), используют id, полученный ранее от get.

Для операций над семафорами и очередями сообщений, по умолчанию, если операция была неудачной, процесс приостанавливается, пока сохраняется условие блокировки. Это похоже на поведение по умолчанию при чтении из пустого программного канала или при записи в заполненный. При работе с очередями сообщений блокировка происходит, если посылающий процесс (msgsnd(2)) обнаружил, что очередь заполнена, или получатель (msgrcv(2)) — что в очереди нет сообщений запрашиваемого типа.

Операции над семафорами включают прибавление к и вычитание целых чисел из значения семафора, с условием что это значение не может стать отрицательным. Операции над семафорами вызовут блокировку, если процесс пытается сделать значение семафора меньше нуля. Кроме того, процесс может ждать, пока это значение не станет нулевым.

Блокировка снимается при одном из следующих условий:

. Операция успешна

. Процесс получил сигнал

. Средство IPC было удалено

Операции могут быть сделаны неблокирующимися, т.е. немедленно возвращающими -1, если требуемая операция не удалась. Для этого в параметр flg должна быть включена побитовым ИЛИ константа IPC_NOWAIT, определенная в <sys/ipc.h>.

Команды ipcs(1) и ipcrm(1)

Команды ipcs(1) и ipcrm(1) используются для операций со средствами IPC из командной строки. Они очень полезны при отладке приложений, работающих с System V IPC.

Команда ipcs(1) распечатывает список всех средств IPC, используемых в системе. Показываются также средства, созданные другими пользователями. Вывод ipcs, вызванной без параметров, показан ниже:

```
$ ipcs
IPC status from /dev/kmem as of Mon Dec 23 15:27:05 1985
T     ID     KEY        MODE       OWNER    GROUP
Message Queues:
Shared Memory:
m   5800 0x00000000 --rw-rw----      jeg    unixc
Semaphores:
s   3200 0x00000000 --ra-ra----      jeg    unixc
```

Для всех трех средств IPC, в колонке MODE стоит 'r', если средство доступно для чтения пользователю, группе и другим пользователям. Для разделяемой памяти и очередей сообщений 'w' означает право записи, для семафоров 'a' - право изменения.

Опции ipcs, показывающие состояние различных типов средств IPC, таковы:

-q Очереди сообщений

-m Разделяемая память

-s Семафоры

Существуют опции ipcs, показывающие все данные, хранящиеся в дескрипторе. Например, "ipcs -m -o" показывает количество сегментов памяти. Подробнее см. руководство по ipcs(1).

Средства IPC обычно удаляются заданием IPC_RMID в системном вызове ctl. Эффективный идентификатор пользователя должен совпадать с id хозяина или создателя ресурса. Обычно ресурс удаляется тем же процессом, который создал его. Если средство не было удалено вызовом ctl, для этого можно использовать ipcrm(1). Иначе оно будет существовать до перезагрузки системы. Средства IPC могут удаляться с заданием либо ключа, либо id. Буквы нижнего регистра q, m и s используются для удаления средства по идентификатору. Буквы верхнего регистра используются для задания ключа. Хотя ipcs показывает ключ в шестнадцатиричном виде, ключ заданный ipcrm в командной строке должен быть десятичным.

Следующие команды удаляют сегмент разделяемой памяти и семафор, показанные выше в выдаче ipcs. Напоминаем, только хозяин или создатель могут удалить средство IPC. После ipcrm, команда ipcs показывает, что средства были удалены.

```
$ id
uid=503(jeg) gid=21(unixc)
$ ipcrm -m5800 -s3200
$ ipcs
IPC status from /dev/kmem as of Mon Dec 23 15:27:26 1985
T     ID     KEY        MODE       OWNER    GROUP
Message Queues:
Shared Memory:
Semaphores:
```

Очереди сообщений

Очереди сообщений позволяют процессам обмениваться данными. Эти данные передаются дискретными порциями, которые называются сообщениями. В отличие от блоков данных в программном канале, сообщения из очереди считываются только целиком, нельзя оставить в очереди непрочитанную часть сообщения. Все сообщения имеют тип. При чтении можно либо читать первое сообщение любого типа, либо отбирать сообщения по типам.

Для работы с очередями сообщений предназначены следующие системные вызовы:

. msgget

. msgsnd

. msgsrv

. msgctl

В этом разделе сначала описывается, как создавать очередь и получать к ней доступ, а затем — как передавать данные через нее.

Структура очередей сообщений

Очередь сообщений создается вызовом msgget(2) с командой IPC_CREAT. При создании очереди создается ее дескриптор msgid_ds. Эта структура данных используется системными вызовами, которые посылают и получают сообщения.

Создав очередь, процессы могут ставить в нее сообщения (msgsnd(2)) и получать их оттуда (msgrcv(2)).

Сообщение представляет собой набор байтов. Его размер может меняться от нуля до максимума, определяемого системой. Содержимое может быть любым - ASCII или двоичными данными. Оно может иметь любой формат, например, массив символов или структура.

Когда сообщение ставится в очередь, отправитель метит его типом. Тип представляет собой длинное целое.

Управляющая структура данных содержит два указателя. Один указывает на первое сообщение в очереди, а другой - на последнее. Очередь образуется связанным списком заголовков сообщений. Каждый заголовок содержит тип сообщения, его размер и адрес, и указатель на следующий заголовок в очереди.

Получатель может избирательно обрабатывать сообщения в очереди. В частности, существуют три способа, которыми получатель может запрашивать сообщение.

. Первое сообщение в очереди.

. Первое сообщение заданного типа.

. Первое сообщение с типом из диапазона значений [1:n]

Доступ к очереди сообщений

Для получения доступа к очереди используется системный вызов msgget(2). Его аргументы:

В качестве ключа key может быть использовано любое длинное целое. Ключ может быть получен использованием ftok(3C) или другим способом, обеспечивающим его уникальность. Также можно использовать значение ключа IPC_PRIVATE. Каждый вызов msgget с этим ключом создает новую очередь сообщений.

msgflg управляет созданием и правами доступа очереди. Его значение получается побитовым ИЛИ следующих констант:

. IPC_CREAT - если не существует очереди с этим ключом или ключ равен IPC_PRIVATE, создает новую очередь.

. IPC_EXCL - только вместе с IPC_CREAT. Очередь создается тогда и только тогда, когда ее не существует. Иными словами, когда заданы IPC_CREAT | IPC_EXCL, и уже существует очередь с заданным ключом, системный вызов возвратит неуспех.

. Девять младших бит msgflg используются для задания прав доступа при создании очереди.

Право чтения определяет возможность получать сообщения, а право записи - посылать их.

В системе обычно действуют конфигурационные и административные ограничения на количество очередей, а также на суммарное количество сообщений в очередях и их общий объем. Обычно лимит количества очередей составляет несколько десятков. В Solaris 10 текущее значение административного лимита можно посмотреть командой:

```
prctl -n project.max-msg-ids $$
```

Вызов msgget(2) возвратит неуспех, если вы попытаетесь выйти за установленный предел количества очередей..

Управление очередью сообщений

Собственность и права доступа для очереди устанавливаются при вызове msgget(2) с флагом IPC_CREAT. Эти атрибуты могут быть изменены системным вызовом msgctl(2). Его аргументы:

msgid - идентификатор очереди, полученный от успешного вызова msgget(2).

cmd может принимать одно из следующих значений:

. IPC_STAT - Записывает состояние очереди в структуру, на которую указывает buf.

. IPC_SET - Используется для установки владельца, группы и права чтения/записи для пользователя, группы и других. Значения берутся из структуры, на которую указывает buf.

Также, при помощи IPC_SET можно изменять поле msg_qbytes, обозначающее максимальный размер очереди. В Solaris 10, значение по умолчанию и максимально допустимое значение размера очереди можно посмотреть командой

```
prctl -n process.max-msg-qbytes $$
```

Обычные пользователи могут только уменьшать размер очереди; увеличивать размер может только супервизор и/или обладатель привилегии PRIV_SYS_IPC_CONFIG.

. IPC_RMID - Удаляет очередь и ее дескриптор.

```
 1  struct msqid_ds {
 2  struct ipc_perm msg_perm;/* operation permission */
 3  struct msg*msg_first;/* ptr to first message on q */
 4  struct msg*msg_last;/* ptr to last message on q */
 5  ulong msg_cbytes;/* current # bytes on q */
 6  ulong msg_qnum;/* # of messages on q */
 7  ulong msg_qbytes;/* max # of bytes on q */
 8  pid_t msg_lspid;/* pid of last msgsnd */
 9  pid_t msg_lrpid;/* pid of last msgrcv */
10  time_t msg_stime;/* last msgsnd time */
11  long msg_stimfrac;/* reserved for time_t expansion */
12  time_t msg_rtime;/* last msgrcv time */
13  long msg_rtimfrac;
14  time_t msg_ctime;/* last change time */
15  long msg_ctimfrac;
16  long pad[4];/* reserve area */
17  };
```

msgctl(2) - Пример

. объявления Переменная msgid должна содержать идентификатор очереди, полученный при вызове msgget(2). Структура данных ds будет содержать информацию о состоянии заданной очереди. Структура msqid_ds определена в <sys/msg.h>.

. удаление Очередь должна удаляться только тогда, когда она больше не нужна всем использовавшим ее процессам. Когда задается команда IPC_RMID, последний аргумент может быть любым.

Замечание: Перед тем, как изменять хозяина, права доступа и т.д. для средства IPC, для него должно быть определено текущее состояние.

. изменение хозяина После копирования информации в структуру типа struct msgid_ds, поле msg_perm.uid может быть установлено равным числовому идентификатору пользователя нового хозяина, и хозяин очереди будет изменен системным вызовом msgctl(2) с командой IPC_SET.

Группа может быть изменена аналогичным образом. Библиотечные функции getpwnam(2) и getgrnam(2) соответственно отображают символические имена пользователя и группы в числовые идентификаторы. Эти функции можно использовать, если вы знаете только имя пользователя или группы.

. изменение прав доступа Права определяют, кто может посылать и получать сообщения из заданной очереди, и могут быть изменены вызовом msgctl(2).

. изменение ограничения на размер Используя msgctl(2) можно изменить максимальный размер очереди, т.е. количество байтов, которые она может содержать (msg_qbytes).

Замечание: Shell-команда ipcs -q -b показывает текущее значение этого ограничения для каждой очереди.

msgctl(2) - ПРИМЕРЫ

. объявления переменных

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgid;
struct msgid_ds ds;
```

. удаление

```
msgctl(msgid, IPC_RMID, NULL);
```

. изменение хозяина

```
msgctl(msgid, IPC_STAT, &ds);
ds.msg_perm.uid = 51; /*new uid*/
msgctl(msgid, IPC_SET, &ds);
```

. изменение прав доступа

```
msgctl(msgid, IPC_STAT, &ds);
ds.msg_perm.mode = 0660;
msgctl(msgid, IPC_SET, &ds);
```

. изменение ограничения на размер

```
msgctl(msgid, IPC_STAT, &ds);
ds.msg_qbytes = 3000;
msgctl(msgid, IPC_SET, &ds);
```

Формат сообщения

Сообщение состоит из тидентификатора ипа сообщения, за которым следует содержимое. Тип сообщения представляет собой длинное целое. Длина сообщения может быть любой, от нуля до заданного в системе максимального значения. В старых версиях SVR4 этот максимум был настраиваемым параметром, по умолчанию равен 2048, и не мог быть больше, чем 64 килобайта. В Solaris 10 соответствующие параметры были удалены; ядро Solaris 10 принимает любые сообщения, при условии, что они меньше msg_qbytes соответствующей очереди.

Формат сообщения выглядит так:

```
long mtype;   /* message type */
char mtext[]; /* message text */
```

Массив без размерности указывает, что за типом сообщения может следовать любое количество байтов. Эти байты не обязаны быть массивом символов. Они могут быть любого типа - структура, массив, числа с плавающей точкой или целое и т.д.

Этот формат ассоциирован с типом struct msgbuf в <sys/msg.h>. Структура показана на следующей странице.

Сообщение, определенное struct msgbuf, представляет собой просто область памяти, где за типом сообщения (длинное целое) следует ноль или более байтов содержимого.

Операции - очереди сообщений - msgsnd(2)

Операции над очередью состоят в посылке (постановке в очередь) сообщений и извлечении их оттуда. Это делается системными вызовами msgsnd(2) и msgrcv(2)

msgid задает очередь, в которую нужно поместить сообщение. msgid – это значение, возвращенное вызовом msgget(2).

msgp указывает на объект типа struct msgbuf, которая только что обсуждалась. msgsz - число байтов, которые нужно послать.

Посылающий задает тип, длинное положительное целое, для каждого посылаемого сообщения. Это первые четыре байта struct msgbuf. В очереди сообщения хранятся в том порядке, в котором были посланы. Получатель может разделять сообщения по типу, и, возможно, получать сообщения в порядке отличном от того, в котором они были посланы.

msgflg задает действия на случай, если сообщение не может быть поставлено в очередь. Как правило, процесс при попытке поставить сообщение в заполненную очередь приостанавливается. Это происходит, если достигнуто максимальное количество байтов, которые могут стоять в очереди (msg_qbytes или MSGMNB), или общее число сообщений в системе достигло заданного в системе максимального значения (MSGTQL). Если такая блокировка нежелательна, вы должны установить в слове msgflg IPC_NOWAIT. msgsnd(2) в таком режиме будет возвращать код неуспеха -1 в ситуациях, когда бы обычная операция была заблокирована.

В традиционных Unix-системах, количество очередей и сообщений в системе, а также максимальный размер сообщения, регулировались параметрами настройки ядра. В Solaris 10 эти параметры регулируются управлением ресурсами на уровне проектов (см. страницы руководства project(4), prctl(1)).

Операции - очереди сообщений - msgrcv(2)

msgrcv - получить сообщение

msgid определяет очередь, из которой будет получено сообщение.

msgp указывает на область памяти, в которую сообщение будет записано. Эта область памяти должна содержать long для типа сообщения, за которым должен следовать буфер, достаточный для самого большого сообщения, которое процесс хотел бы получить.

msgsz показывает максимальный размер сообщения, которое процесс хотел бы получить. Это значение должно быть меньше или равно размеру буфера, который следует за long. Иными словами, как и у msgsnd(2), общий размер буфера должен составлять msgsz+sizeof(long)

msgflg может иметь установленными биты IPC_NOWAIT и MSG_NOERROR.

Параметр msgtyp задает тип сообщения, которое нужно получить. Он может принимать следующие значения:

0 Будет получено первое сообщение в очереди.

n Будет получено первое сообщение типа n.

-n Первое сообщение наименьшего типа, находящееся в очереди, с типами в диапазоне от 1 до абсолютного значения n.

Как правило, процесс будет приостановлен, если в очереди нет сообщений заданного типа. Процесс будет приостановлен , пока либо в очереди не появится требуемое сообщение, либо не будет перехвачен сигнал, либо очередь не будет удалена. Если такая приостановка нежелательна, в msgflg нужно установить IPC_NOWAIT.

msgsz задает максимальное количество байтов, которое может быть получено (размер приемного буфера). Количество, которое будет в действительности получено, зависит от msgsz и msgflg. Предположим, что size_sent - это число байтов в сообщении, посланном вызовом msgsnd(2). Если size_sent <= msgsz (т.е., если было послано меньше, чем максимум,заданный получателем), будет получено size_sent байтов.

Если size_sent > msgsz, посланное сообщение больше, чем получатель хотел бы. Возникнет ошибка или нет, определяется значением msgflg. Если установлен MSG_NOERROR, ошибки не возникнет. Первые msgsz байтов будут получены, а остаток отброшен.

Если же флага MSG_NOERROR нет, msgrcv(2) возвращает -1, если размер сообщения больше, чем msgsz. В этом случае сообщение не будет потеряно. Если получатель задаст MSG_NOERROR в следующем обращении к msgrcv(2), или использует больший буфер, возможно, выделенный malloc(3C), то сообщение может быть прочитано.

В случае успеха, msgrcv(2) возвращает количество прочитанных байтов. Этот вызов не имеет средств для определения id процесса или id пользователя процесса-отправителя. Эту информацию можно передавать в теле сообщения, или даже в типе, если он не используется для других целей.

Пример сообщений - отправитель

Этот пример состоит из двух отдельных процессов — отправителя сообщений и их получателя.

1-7 Необходимые include-файлы.

12Объявить переменную для значения ключа.

13 Объявить переменную для идентификатора очереди сообщений.

14-17 Сообщение представляет собой структуру. Первое поле должно быть длинным целым. Остаток сообщения формируется пользователем.

24-27 Создать ключ, основываясь на текущей директории.

28 Создать очередь сообщений.

37-44 Послать три сообщения с типом равным 1L. Длина отправляемых данных включает нулевой байт, завершающий строку.

45 Ожидать получения сообщения о завершении (по договоренности, типа 99L) от получателя, перед тем, как завершиться. Тип сообщения Done определен в строке 9. Это может быть любое значение, кроме 1L.

50 Удалить очередь сообщений.

ПРИМЕР СООБЩЕНИЙ - ОТПРАВИТЕЛЬ

```
 1  #include <sys/types.h>
 2  #include <string.h>
 3  #include <unistd.h>
 4  #include <stdlib.h>
 5  #include <stdio.h>
 6  #include <sys/ipc.h>
 7  #include <sys/msg.h>
 8  #define MAX_SEND_SZ  30
 9  static const long Done = 99L;
10  main()
11  {
12     key_t key;
13     int mid;
14     struct msgbuf {
15        long mtype;
16        char mtext[40];
17     } buf;
18     static char *strings[3] = {
19        "hello",
20        "how are you",
21        "good-bye"
22     };
23     int i, rtn;
24     if((key = ftok(".", 'a')) == -1){
25        perror("Can't form key");
26        exit(1);
27     }
28     mid = msgget(key, IPC_CREAT | 0660);
29     if (mid == -1) {
30       perror("Sender can't make msg queue");
31       exit(2);
32     }
36     buf.mtype = 1L;
37     for(i=0; i < 3; i++){
38       strcpy(buf.mtext,strings[i]);
39       if(msgsnd(mid,&buf,strlen(buf.mtext)+1,0)
40       {
41         perror("Sender can't msgsnd");
42         exit(3);
43       }
44     }
45     rtn=msgrcv(mid,&buf,MAX_SEND_SZ,Done,0);
46     if( rtn == -1 ){
47       perror("Sender can't msgrcv");
48       exit(4);
49     }
50     msgctl(mid, IPC_RMID, NULL);
51     return(0);
52  }
```

Пример сообщений - получатель

15-18 Сообщение представляет собой структуру. Первое поле в структуре должно быть длинным целым. Остаток структуры определяется пользователем.

22 Создает ключ на основе текущей рабочей директории. Как msend, так и mrecv должны запускаться в одной и той же рабочей директории.

31-36 Получать и распечатывать сообщения в цикле, пока не получено good-bye.

37-38 Послать сообщение, состоящее только из типа (без содержимого), обратно к msend. При получении этого сообщения, msend удалит очередь и завершится.

Файл: mrecv.c

ПРИМЕР СООБЩЕНИЙ - ПОЛУЧАТЕЛЬ

```
 1  #include <sys/types.h>
 2  #include <unistd.h>
 3  #include <string.h>
 4  #include <stdio.h>
 5  #include <stdlib.h>
 6  #include <sys/ipc.h>
 7  #include <sys/msg.h>
 8  #define MAX_RCV_SZ60
 9  static const long Fifo = 0L;
10  static const long Done = 99L;
11  static const int Zero_len = 0;
12
13  main()
14  {
15     struct msgbuf {
16        long mtype;
17        char mtext[MAX_RCV_SZ];
18     } buf;
19     key_t key;
20     int mid, rtn;
21
22     if((key = ftok(".", 'a')) == -1){
23        perror("Can't form key");
24        exit(1);
25     }
26     mid = msgget(key, 0);
27     if (mid == -1) {
28        perror("Receiver can't access msg queue");
29        exit(2);
30     }
31     while(1) {
32        rtn = msgrcv(mid,&buf,MAX_RCV_SZ,Fifo,0);
33        printf("rtn=%d buf.mtype=%ld buf.mtext=%s\n",
34           rtn, buf.mtype, buf.mtext);
35        if(!strcmp(buf.mtext,"good-bye"))break;
36     }
37     buf.mtype = Done;
38     msgsnd(mid, &buf, Zero_len, 0);
39     return(0);
40  }
```

Пример сообщений - вывод

Программа msend запускается первой в фоновом режиме. Она создает очередь и посылает в нее сообщения. Программа mrecv последовательно читает их, распечатывает количество полученных символов, тип сообщения и его содержимое.

ПРИМЕР СООБЩЕНИЙ - ВЫВОД

```
$ msend&
7080
$ mrecv
rtn=6 buf.mtype=1 buf.mtext=hello
rtn=12 buf.mtype=1 buf.mtext=how are you
rtn=9 buf.mtype=1 buf.mtext=good-bye
```

Семафоры

Все средства работы нескольких процессов или потоков с разделяемыми данными, в особенности с разделяемой памятью, нуждаются в дополнительных средствах, позволяющий координировать доступ к этим данным. Без средств координации, один из процессов может привести данные в несогласованное состояние и нарушить работу других процессов. Например, если разделяемые данные представляют собой список свободных мест в самолёте или автобусе, отсутствие координации может привести к тому, что билет на одно место может быть продан два раза.

System V IPC включает средство для координации доступа к разделяемым данным.

Это средство называется семафорами. Вы изучите системные вызовы, которые создают семафоры, управляют ими и совершают операции над ними.

Каждый из вас встречал семафоры в повседневной жизни. Например, перекресток представляет собой разделяемый между двумя улицами ресурс. В каждый момент через перекресток может проходить только один транспортный поток. Светофор или регулировщик играют роль «семафора», блокируя один транспортный поток и позволяя проходить другому. Автомобили, приближающиеся к заблокированному перекрестку, выстраиваются и формируют очередь. Семафор System V IPC похож на дорожный светофор не только выполняемой функцией, но и тем, что он не ставит физических препятствий перед процессами, пытающимися нарушить координацию. Семафор указывает процессу, что ему не следует обращаться к ресурсу, защищаемому семафором, но не мешает процессу сделать это.

Изобретение семафоров приписывается Дейкстре, который предложил этот механизм для координации доступа программ к разделяемому ресурсу.

В операционной системе UNIX семафор представляет собой семафор-счетчик или, что то же самое, семафор Дийскстры общего вида. Это беззнаковое короткое целое, разделяемое несколькими программами. Программисты связывают со значениями этого числа определенный смысл, используя семафоры при описании взаимодействия процессов.

Над семафором определены три операции: добавление и вычитание целочисленных значений и ожидание, пока значение семафора не станет нулевым. При вычитании действует ограничени, что значение семафора не может стать отрицательным. Если программа попытается сделать это, она будет приостановлена.

Обычно семафор инициализируется значением, равным числу программ, которые могут иметь одновременный доступ к ресурсу. Если, например, таким ресурсом является принтер, то значение семафора должно быть 1, потому что принтер, как правило, используется в каждый момент только одной программой. Если программа хочет получить доступ к принтеру, она уменьшает значение семафора на 1 операцией WAIT. Другая программа, пытающаяся печатать на том же принтере, также попытается выполнить WAIT и приостановится, так как значение семафора при этом стало бы отрицательным. Второй процесс становится в очередь к семафору и ждет, пока значение семафора не станет большим или равным 1. Это случится, когда первая программа закончит печать и выполнит над семафором операцию POST, увеличив его на 1. Теперь второй процесс сможет выполнить WAIT и использовать ресурс, так как можно вычесть 1 из семафора, и он не станет отрицательным.

Наборы семафоров

В System V IPC, семафоры создаются в виде наборов. Если вам необходим только один семафор, вам следует создать набор из одного семафора. Наборы полезны, если программе нужно изменить значение нескольких семафоров одновременно.

Семафоры обычно используются для обозначения доступности одного или более ресурсов следующими способами:

. закрытие/открытие - каждый ресурс может одновременно использоваться только одним процессом. Процесс, требующий использования ресурса, ждет, пока значение семафора не покажет, что ресурс свободен. Когда процесс получает ресурс, он должен закрыть семафор, чтобы не позволить другим процессам доступ к ресурсу.Когда процесс освобождает ресурс, он должен открыть семафор. При таком использовании, операции WAIT и POST также называют, соответственно, LOCK и UNLOCK.

Замечание: При создании семафор получает по умолчанию значение 0. Обычно он инициализируется в 1 уже после создания. Закрытие состоит в вычитании 1 из значения семафора, а открытие — в добавлении 1. Можно использовать и другие арифметические схемы.

. производитель-потребитель. Процессы обмениваются порциями данных, при потребитель должен получать сигнал, что готова новая порция данных, а производитель — что потребитель обработал предыдущую. Классическое решение этой задачи на двух семафорах состоит в том, что один семафор используется для оповещения производителя, а другой — потребителя. В начальном состоянии, потребитель заблокирован в WAIT на своем семафоре. Производитель генерирует очередную порцию данных, делает POST на семафор потребителя и засыпает в WAIT на своем семафоре; потребитель просыпается, обрабатывает данные и делает POST на семафор производителя, и т. д. Это решение несложно обобщить на сценарии нескольких производителей или потребителей или ситуацию, когда несколько порций данных могут ждать обработки.

. подсчет. Семафор может быть использован для разделения беззнакового целого между процессами. Он не обязан быть ассоциирован с разделяемым ресурсом.

Управление доступом к разделяемому ресурсу работает только тогда, когда все программы выполняют соглашение о семафоре для этого ресурса.

Системные вызовы для работы с семафорами

Ниже приведен обзор системных вызовов для работы с семафорами:

semget(2) Этот системный вызов получает набор из одного или более семафоров. semget(2) возвращает идентификатор набора семафоров. Семафор однозначно определяется этим идентификатором и начинающимся с нуля индексом в наборе.

semctl(2) Этот системный вызов служит следующим целям:

. Получает значение одиночного семафора из набора или всех семафоров в наборе.

. Устанавливает значение одного или всех семафоров в наборе.

. Получает информацию о состоянии набора семафоров.

. Определяет число процессов, ожидающих, пока семафор из набора не станет нулевым.

. Определяет число процессов, ожидающих, пока семафор в наборе не увеличится по сравнению с его текущим значением.

. Определяет процесс, который выполнял последнюю операцию над семафором.

. Изменяет права доступа к набору семафоров.

. Удаляет набор семафоров. Наборы семафоров, так же как файлы и очереди сообщений, должны удаляться явным образом.

semop(2) Этот системный вызов оперирует с одним или несколькими семафорами в наборе. В действительности, это набор операций над набором семафоров. Каждая операция позволяет увеличить значение семафора на заданную величину (POST или UNLOCK), уменьшить (WAIT или LOCK(, или ожидать, пока значение семафора не станет нулевым. Уменьшение значения семафора может заблокировать процесс, если вычитаемая величина меньше его текущего значения.

Набор операций выполняется атомарно, в том смысле, что при проверке возможности всех операций никакие другие операции над семафорами набора не выполняются. Если какая-то из операций приводит к блокировке, то ни одна операция из набора не выполняется и весь набор операций блокируется. Когда какой-то другой процесс изменит семафоры, снова проверяется возможность всех операций в наборе и т. д. Это исключает возможность мёртвой блокировки, но может привести к так называемой «проблеме голодания», когда набор операций блокируется на неограниченное время, при том, что для каждой отдельно взятой операции существуют интервалы времени, в течении которого она возможна.

Получение доступа к набору семафоров

Системный вызов semget(2) используется для создания или получения доступа к набору из одного или нескольких семафоров. При успехе, он возвращает идентификатор набора семафоров. Аргументы semget(2):

key Ключ доступа к набору. Похож на имя файла. В качестве ключа может использоваться любое целое значение. Различные пользователи набора должны договориться об уникальном значении ключа. Ключ может быть создан библиотечной функцией ftok(3). Если необходим приватный ключ, может быть использовано значение IPC_PRIVATE.

nsems Количество семафоров в наборе. Это значение должно быть больше или равно 1. Семафор задается идентификатором набора и индексом в этом наборе. Индекс меняется от нуля до nsems-1.

semflg Биты прав доступа и флаги, используемые при создании набора. Девять младших битов задают права доступа для хозяина, группы и других пользователей. Для набора семафоров определены права чтения и изменения. Флаги таковы:

IPC_CREAT Если этот флаг установлен и набор не существует, или задан ключ IPC_PRIVATE, будет создан новый набор. Если же набор с таким ключом уже существует и не задан флаг IPC_EXCL, то semget(2) возвратит его идентификатор.

IPC_EXCL Этот флаг используется только вместе с IPC_CREAT. Он используется для того, чтобы создать набор только тогда, когда такого набора еще не существует. Этот флаг похож на O_EXCL при создании файлов.

Следующие системные параметры, просматриваемые prctl(1) ограничивают вызов semget(2):

process.max-sem-nsems
- максимальное количество семафоров в наборе

project.max-sem-ids
и zone.max-sem-ids
- максимальное количество наборов семафоров в проекте или зоне, соответственно.

Получение доступа к семафору - Пример

Эта программа показывает использование семафора для доступа к одиночному разделяемому ресурсу. В этом примере разделяемый ресурс представляет собой стандартный вывод — экран вашего терминала. Запускаются две параллельные копии программы; это можно сделать при помощи запуска в фоновом режиме из shell (для этого нужно добавить символ & в конец командной строки).

Каждый процесс получает исключительный доступ к терминалу для вывода неразорванной текстовой строки.

Замечание: текст этой программы используется в нескольких следующих примерах для демонстрации работы системных вызовов с семафорами.

Фрагмент программы работает следующим образом:

20 Функция ftok(3) создает ключ доступа к набору семафоров. Было бы полезно проверить успешность создания ключа, сравнив полученное от ftok(3) значение с -1.

21-24 Выполняется попытка создать семафор. Если она успешна, переменной creator присваивается 1.

25-31 Иначе, семафор может быть уже создан, и делается попытка получить к нему доступ. Если это не выходит, программа печатает сообщение об ошибке и завершается.

... Отсутствующий код описан в следующих примерах.

Файл: semdemo.c

ПОЛУЧЕНИЕ ДОСТУПА К СЕМАФОРУ - ПРИМЕР

```
 1  #include <unistd.h>
 2  #include <stdlib.h>
 3  #include <sys/types.h>
 4  #include <sys/ipc.h>
 5  #include <sys/sem.h>
 6  #include <stdio.h>
 7  #define  DUMMY  0
 8  #define  COUNT  4
 9
10  main(int argc, char *argv[])
11  {
12     key_t ipckey;
13     int semid, pid, creator, i;
   ...
20     ipckey = ftok(argv[0], 's');
21     if ((semid = semget(ipckey, 1,
22           IPC_CREAT|IPC_EXCL|0666)) != -1) {
23        creator = 1;
24     }
25     else {
26        if((semid=semget(ipckey,1,0))==-1){
27          perror(argv[0]);
28           exit(1);
29        }
30        creator = 0;
31    }
   ...
62  }
```

Управление семафорами

semctl(2) выполняет действия по управлению наборами семафоров и одиночными семафорами из набора. Аргументы semctl(2):

semid идентификатор, полученный от semget(2)

semnum индекс семафора в наборе. Первый семафор в наборе имеет индекс 0.

cmd команда. Возможные значения этого аргумента обсуждаются на следующей странице.

arg тип этого параметра зависит от команды cmd. Это может быть:

. Целое число, задающее новое значение семафора

. Указатель на массив беззнаковых коротких целых, используемый для установки и получения значения всех семафоров в наборе.

. Указатель на информационную структуру semid_ds для набора семафоров.

```
sys/sem.h:
struct semid_ds {
   struct ipc_permsem_perm;/* operation permission struct */
   struct semsem_base;/* ptr to first semaphore in set */
   ushortsem_nsems;/* # of semaphores in set */
   time_tsem_otime;/* last semop time */
   longsem_otimfrac;/* reserved for time_t expansion */
   time_tsem_ctime;/* last change time */
   longsem_ctimfrac;
   longpad[4]; /* reserve area */
};
```

intro(2) содержит дополнительную информацию о структурах данных, используемых для работы с семафорами. Кроме того, можно получить справки в файлах <sys/ipc.h> и <sys/sem.h>.

semctl(2) - Примеры

В прототипе semctl(2) последний параметр указан как union. Это означает, что тип последнего параметра зависит от значения команды cmd. На следующей странице приведены примеры использования различных значений cmd. Ниже показано, какие типы arg используются с различными командами:

. int val;

SETVAL Эта команда устанавливает значение отдельного семафора в наборе.

. struct semid_ds *buf;

IPC_STAT Эта команда копирует состояние набора семафоров в буфер buf.

IPC_SET Эта команда устанавливает значения хозяина, группы и прав доступа для набора семафоров.

. ushort *array;

GETALL Эта команда получает значения всех семафоров в наборе и помещает их в массив, на который указывает array.

SETALL Устанавливает все семафоры из набора в значения, которые хранятся в массиве array.

. arg не используется

GETVAL Эта команда получает значение семафора с индексом semnum.

GETPID Эта команда получает идентификатор процесса, который совершал последнюю операцию над семафором с индексом semnum.

GETNCNT Эта команда получает количество процессов, ожидающих увеличения значения семафора по сравнению с текущим.

GETZCNT Эта команда получает количество процессов, ожидающих, пока значение семафора не станет 0.

IPC_RMID Эта команда удаляет набор семафоров и его идентификатор. Если какие-то процессы были заблокированы в операциях с этим набором, во всех этих процессах соответствующие вызовы semop(2) вернут ошибку EIDRM.

semctl(2) - ПРИМЕРЫ

. описания и системный вызов для создания набора семафоров

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#define ANY 0
int semid, rtn;
struct semid_ds ds;
ushort us[5], init_us[5] = { 0, 6, 7, 1, 4 };
...
semid = semget(key, 5, IPC_CREAT | 0660);
```

. инициализировать один семафор из набора

```
    semctl(semid, 2, SETVAL, 7);
```

. получить значение одного семафора из набора

```
  /* такой же формат для GETNCNT, GETZCNT, GETPID */
    rtn = semctl (semid, 2, GETVAL, ANY);
```

. инициализировать все семафоры в наборе

```
    semctl (semid, ANY, SETALL, init_us);
```

. получить все значения семафоров в наборе

```
    semctl (semid, ANY, GETALL, us);
```

. изменить хозяина

```
    /* также изменяются права доступа */
    semctl (semid, ANY, IPC_STAT, &ds);
    ds.sem_perm.uid = 51; /* new uid */
    semctl (semid, ANY, IPC_SET, &ds);
```

. удалить набор семафоров

```
    semctl (semid, ANY, IPC_RMID, ANY);
```

Инициализировать и удалить семафор - Пример

Это тот же пример, что и выше, но здесь показан код для инициализации семафора и его удаления. Этот фрагмент программы работает так:

32-37 Если эта программа создала набор семафоров, инициализировать его в 1. Как правило, набор семафоров должен быть удален перед выходом из программы, если его инициализация не удалась, но это здесь не показано.

54-60 Создатель набора семафоров может удалить его. В нашей программе, по соглашению, удалением семафоров занимается тот же процесс, который их создал. Обратите внимание на использование "пустых" аргументов в тех местах, где соответствующие параметры системных вызовов не используются. Функция sleep(3C) задерживает удаление семафора на 5 секунд, так что другая программа может продолжать работу с ним.

Файл: semdemo.c
ИНИЦИАЛИЗИРОВАТЬ И УДАЛИТЬ СЕМАФОР - ПРИМЕР

```
 1 #include <unistd.h>
 2 #include <stdlib.h>
 3 #include <sys/types.h>
 4 #include <sys/ipc.h>
 5 #include <sys/sem.h>
 6 #include <stdio.h>
 7 #define  DUMMY  0
 8 #define  COUNT  4
 9
10 main(int argc, char *argv[])
11 {
   ...
32  if (creator) {/* initialize semaphore */
33  if (semctl(semid, 0, SETVAL, 1) == -1) {
34  perror(argv[0]);
35  exit(2);
36  }
37  }
   ...
54  if (creator) {
55  sleep(5);
56  if(semctl(semid,DUMMY,IPC_RMID,DUMMY) == -1) {
57  perror(argv[0]);
58  exit(5);
59  }
60  }
61  return(0);
62 }
```

Операции над семафорами

Системный вызов semop(2) производит операции над одним или более семафорами из набора. Операции увеличивают или уменьшают значение семафора на заданную величину, или ожидают, пока семафор не станет нулевым.

Аргументы этого системного вызова таковы:

semid Идентификатор набора семафоров.

sops Адрес массива с элементами типа struct sembuf. Каждый элемент задает одну операцию над одним семафором. По этому адресу должно размещаться nsops таких структур. Эта структура определена следующимобразом:

```
sys/sem.h:
struct sembuf {
    ushort sem_num; /* semaphore */
    short sem_op;   /* semaphore operation */
    short sem_flg;  /* operation flags */
};
```

Поля этой структуры имеют следующий смысл:

sem_num Индекс семафора, над которым производится операция.

sem_op Если это значение положительно, оно добавляется к текущему значению семафора (POST).

Если sem_op отрицательно, и его абсолютное значение больше текущего значения семафора, операция обычно блокируется. Иначе, из семафора просто вычитается абсолютное значение этого поля (WAIT).

Если sem_op равен нулю, операция блокируется, пока семафор не станет нулевым.

sem_flg Это поле задает дополнительные флаги. Нулевое значение означает, что не задано никаких флагов.Допустимы следующие флаги, объединяемые побитовым ИЛИ:

IPC_NOWAIT Если установлен этот флаг, и операция должна была бы привести к блокировке, semop(2) возвращает неуспех. Этот флаг может быть использован для анализа и изменения значения семафора без блокировки.

SEM_UNDO Если этот флаг установлен, при завершении процесса (как по exit(2), так и по сигналу), операция будет отменена. Это защищает от посмертной блокировки ресурса процессом, который ненормально завершился, не успев освободить ресурс.

Использование SEM_UNDO отменяет операцию над семафором, но не обеспечивает, что ресурс, защищаемый этим семафором, остался в согласованном состоянии после завершения процесса.

Кроме того, нужно иметь в виду, что отмена операций реализована в виде счетчика, в котором накапливаются все значения, добавляемые и вычитаемые процессом при операциях над данным семафором. При завершении процесса этот счетчик просто вычитается из текущего значения семафора. Из этого описания должно быть ясно, что если вы используете флаг SEM_UNDO, его необходимо использовать при всех операциях над этим семафором. Если вы указываете SEM_UNDO только при операциях LOCK, но не при UNLOCK, счетчик отмены может просто переполниться, а его применение к семафору может привести к нежелательным результатам.

При выполнении операции над несколькими семафорами, значения семафоров не меняются, пока не окажется, что все требуемые операции могут быть успешно выполнены. При этом, поведение системы в случае, когда набор операций содержит несколько разных операций

над одним и тем же семафором, не стандартизовано. Разные реализации POSIX могут интерпретировать такой набор по разному.

nsops Количество структур в массиве, на который указывает sops. nsops должен всегда быть больше или равен 1. Параметр prctl(1) process.max-sem-ops ограничивает максимальное количество операций в одном наборе.

Блокировка ресурса - Пример

Здесь показан фрагмент программы для захвата и освобождения ресурса. Разделяемым ресурсом является возможность вывода на терминал. Обычно, если два процесса одновременно пишут на терминал, то вывод обоих процессов смешивается на экране.

В этом примере программа использует семафоры, чтобы получить исключительный доступ к терминалу и напечатать текстовую строку.Если эта программа параллельно исполняется двумя процессами, то процессы будут использовать терминал по очереди.

Этот фрагмент программы работает следующим образом:

14-17 Эти командные структуры задают операцию захвата (вычесть 1) и операцию освобождения (добавить один). Флаг SEM_UNDO используется для восстановления предыдущего значения семафора, если программа, изменившая его, ненормально завершится. Один из способов вызвать ненормальное завершение программы - нажать клавишу BREAK или DEL на терминале.

... Отсутствующий код содержит вызов semget(2) для создания набора из одного семафора и semctl(2) для установки его значения в 1 командой SETVAL.

41-44 Здесь семафор уменьшается на 1, блокируя (захватывая) ресурс.

45 Печатается сообщение о том, что ресурс заблокирован. Заметьте, что вывод не завершается переводом каретки, то есть это будет еще не вся строка вывода. В пропущенном коде необходимо выключить буферизацию stdio, чтобы неполные строки выводились на устройство.

46 sleep(3C) изображает использование терминала. Кроме того, это освобождает процессор на время, пока процесс спит.

47-51 Печатается сообщение, что ресурс освобождается. Затем ресурс действительно освобождается увеличением значения семафора на 1.

Ниже приведена выдача двух экземпляров этой программы, запущенных в

фоновом режиме. Это может быть сделано одной командной строкой.

```
$ semdemo & semdemo &
12423
12424
$ [12423] locking[12423] unlocking
[12424] locking[12424] unlocking
[12423] locking[12423] unlocking
[12424] locking[12424] unlocking
[12423] locking[12423] unlocking
[12424] locking[12424] unlocking
[12423] locking[12423] unlocking
[12424] locking[12424] unlocking
```

Заметьте, что каждый процесс по очереди пишет полную строку на стандартный вывод, являющийся сейчас разделяемым ресурсом.

Файл: semdemo.c

БЛОКИРОВКА РЕСУРСА - ПРИМЕР

```
 1      #include <unistd.h>
 2      #include <stdlib.h>
 3      #include <sys/types.h>
 4      #include <sys/ipc.h>
 5      #include <sys/sem.h>
 6      #include <stdio.h>
 7      #define  DUMMY  0
 8      #define  COUNT  4
 9
10       main(int argc, char *argv[])
11       {
   ...
14       static struct sembuf lock =
15       { 0, -1, SEM_UNDO };
16       static struct sembuf unlock =
17       { 0, 1, SEM_UNDO };
   ...
39       pid = getpid();
40       for (i = 0; i < COUNT; i++ ) {
41       if (semop(semid, &lock, 1) == -1) {
42       perror(argv[0]);
43       exit(3);
44       }
45       printf("[%d] locking\t", pid);
46       sleep(1);/* terminal output being used */
47       printf("[%d] unlocking\n", pid);
48       if (semop(semid, &unlock, 1) == -1) {
49       perror(argv[0]);
50       exit(4);
51       }
52       }
   ...
62       }
```

Набор семафоров - Использование

Этот пример иллюстрирует использование набора семафоров, содержащего более чем один семафор. Этот набор используется для управления ресурсами печати. Существует три отдельных ресурса, и каждому из них соответствует собственный семафор в наборе. Эти ресурсы суть все принтеры вместе, механический принтер и лазерный принтер. Заметьте, что семафор 0, первый в наборе, инициализирован значением 2, т.к. в общем принтерном ресурсе есть два принтера. Пунктирные линии показывают связи между тремя ресурсами и тремя семафорами.

Семафор с индексом 1 управляет доступом к файлу устройства, обозначенному переменной среды PRINTER1. Семафор с индексом 2 управляет устройством, на которое ссылается PRINTER2.

Первые четыре буквы имени команды определят, должен вывод идти на принтер, связанный с PRINTER1 или с PRINTER2. Если имя команды начинается с line, должен быть использован PRINTER1. Если оно начинается с lase, нужно использовать PRINTER2.

Создание набора семафоров - Пример

Этот пример использует набор семафоров для управления всеми принтерами, трактуемыми как общий ресурс печати. Этот код показывает, как создать и инициализировать набор из более чем одного семафора.

Эта программа использует следующий файл заголовка:

printer.h
```
1       #define   DUMMY   0
2       #define   NUMPR   2/* number of printers */
3       #define   ACQUIRE     -1
4       #define   RELEASE      1
```
Фрагмент программы работает так:

15 Объявление массива, содержащего инициализационные значения всех семафоров в наборе.

... Переменные среды PRINTER1 и PRINTER2 содержат имена "принтеров", ассоциированных с индексами 1 и 2 набора семафоров. Эти "принтеры" могут быть как файлами, так и устройствами.

31 Формирование ключа на основании имени программы и буквы s. Так как два имени программы laserprnt и lineprnt представляют собой ссылки на один и тот же файл, ftok(3), используя эти имена в качестве параметров, выдает одно и то же устройство и номер инода.

32-35 Здесь создается набор из NUMPR + 1 семафоров. Дополнительный семафор с индексом 0 управляет ресурсом печати в целом.

45-54 Первый семафор в наборе инициализируется количеством принтеров в общем ресурсе печати. Остальные семафоры инициализируются в 1. Заметьте, что второй аргумент semctl(2) не используется.

Файл: printer.c

СОЗДАНИЕ НАБОРА СЕМАФОРОВ - ПРИМЕР

```
 1 #include <stdlib.h>
 2 #include <string.h>
 3 #include <unistd.h>
 4 #include <fcntl.h>
 5 #include <sys/types.h>
 6 #include <sys/ipc.h>
 7 #include <sys/sem.h>
 8 #include <stdio.h>
 9 #include "printer.h"
10
11 main(int argc, char *argv[])
12 {
 ...
14    key_t ipckey;
15    ushort initial[NUMPR + 1], prntnum;
 ...
31    ipckey = ftok(argv[0], 's');
32    if ((semid = semget(ipckey, NUMPR + 1,
33     IPC_CREAT | IPC_EXCL | 0666)) != -1) {
34       creator = 1;
35    }
 ...
45    if(creator) { /* initialize semaphore set */
46       initial[0] = NUMPR;
47       for (n = 1; n <= NUMPR; n++)
48           initial[n] = 1;
49       if(semctl(semid,DUMMY,SETALL,initial) == -1) {
50           sprintf(errmsg,"%s - SETALL", argv[0]);
51           perror(errmsg);
52           exit(3);
53       }
54    }
 ...
```

Операции над набором семафоров — Пример (продолжение)

Код, приведенный в этом примере, представляет собой часть той же программы, что и предыдущий пример. Он демонстрирует, как оперировать с двумя семафорами в наборе одновременно.

Фрагмент программы работает так:

... Имя команды определяет принтер, который нужно использовать.

56-61 Две командные структуры установлены так, чтобы уменьшить на 1 первый семафор и семафор, связанный с принтером с номером prnnum. Заметьте, что используется флаг SEM_UNDO. Вспомните, что номер семафора, над которым должна быть выполнена операция, содержится в командной структуре.

62-66 Здесь выполняется операция над двумя семафорами одновременно. Эта операция захватит принтер, который должен быть использован. Другой процесс, исполняющий этот же код, будет заблокирован до тех пор, пока значение семафора не увеличится.

79-85 Командные структуры для обоих семафоров изменены так, чтобы увеличить значение семафоров на 1. Оба семафора изменяются "одновременно", т.е. ни один другой процесс не получит управления в промежутке между их изменениями.

Эта программа демонстрируется так:

```
$ ln printer lineprntr
$ ln printer laseprntr
$ PRINTER1=/dev/tty05
$ PRINTER2=/tmp/xyz
$ export PRINTER1 PRINTER2
$ lineprntr data & lineprntr data & wait
3909
3910
   1ABCDEFGHIJKLMNOPQRSTUVWXYZ
   2ABCDEFGHIJKLMNOPQRSTUVWXYZ
   3ABCDEFGHIJKLMNOPQRSTUVWXYZ
   4ABCDEFGHIJKLMNOPQRSTUVWXYZ
   1ABCDEFGHIJKLMNOPQRSTUVWXYZ
   2ABCDEFGHIJKLMNOPQRSTUVWXYZ
   3ABCDEFGHIJKLMNOPQRSTUVWXYZ
   4ABCDEFGHIJKLMNOPQRSTUVWXYZ
$ ipcs -s
IPC status from /dev/kmem as of Tue Mar  3 11:18:53 1987
T    ID    KEY       MODE       OWNER    GROUP
Semaphores:
s    100 0x7304001a --ra-ra-ra-      jeg    unixc
$ ipcrm -s 100
```

Файл: printer.c

ОПЕРАЦИИ НАД НАБОРОМ СЕМАФОРОВ - ПРИМЕР

```
 1  #include <stdlib.h>
 2  #include <string.h>
 3  #include <unistd.h>
 4  #include <fcntl.h>
 5  #include <sys/types.h>
 6  #include <sys/ipc.h>
 7  #include <sys/sem.h>
 8  #include <stdio.h>
 9  #include "printer.h"
10
11   main(int argc, char *argv[])
12   {
 ...
17       struct sembuf operation[2];
 ...
56       operation[1].sem_num = prntnum;
57       operation[1].sem_op = ACQUIRE;
58       operation[1].sem_flg = SEM_UNDO;
59       operation[0].sem_num = 0;
60       operation[0].sem_op = ACQUIRE;
61       operation[0].sem_flg = SEM_UNDO;
62       if(semop(semid, operation, 2) == -1) {
63          sprintf(errmsg,"%s - ACQUIRE",argv[0]);
64          perror(errmsg);
65          exit(4);
66       }
 ...
79       operation[1].sem_op = RELEASE;
80       operation[0].sem_op = RELEASE;
81       if(semop(semid, operation, 2) == -1) {
82          sprintf(errmsg,"%s - RELEASE",argv[0]);
83          perror(errmsg);
84          exit(7);
85       }
 ...
87   }
```

Разделяемая память

Разделяемая память System V IPC позволяет двум или более процессам разделять память и, следовательно, находящиеся в ней данные. Это достигается помещением в виртуальное адресное пространство процессов одной и той же физической памяти.

Того же эффекта можно достичь, отобразив в память при помощи mmap(2) доступный на запись файл в режиме MAP_SHARED. Как и в случае mmap(2), разделяемые сегменты не обязательно будут отображены на одни и те же адреса в разных процессах.

Главным практическим отличием разделяемой памяти System V IPC от mmap(2) является то, что для mmap(2) нужен файл, а память System V IPC ни к какому файлу не привязана. Кроме того, использование mmap(2) с флагом MAP_SHARED приводит к тому, что система синхронизует содержимое памяти с диском, что может снизить общую производительность системы. Если вам нужно сохранить содержимое разделяемой памяти после завершения работы всех процессов приложения, mmap(2) оказывается удобнее, но на практике такая потребность возникает довольно редко. Поэтому разделяемая память System V IPC до сих пор широко применяется многими приложениями.

В последние годы, разделяемая память вытесняется многопоточными программами. С определенными оговорками, с точки зрения программиста, потоки можно рассматривать как процессы, у которых вся память разделяется.

Этот раздел показывает, как создавать и использовать сегмент разделяемой памяти. Это выполняется следующими системными вызовами.

. shmget

. shmat

. shmdt

. shmctl

Разделяемая память

Это средство IPC позволяет двум или более процессам разделять память.

Такая память используется для хранения данных, которые нужны нескольким процессам. Это могут быть поисковые таблицы, критерии правильности данных и т. д. Разделяемая память представляет собой самый быстрый способ обмена данными между процессами.

Как правило, один процесс создает сегмент разделяемой памяти вызовом shmget(2) с флагом IPC_CREAT, отображает его в свое адресное пространство и инициализирует. Отображения разделяемого сегмента в адресное пространство процесса называют присоединением сегмента. Затем другие процессы могут присоединять этот сегмент и использовать его содержимое. Создатель задает права чтения/записи для хозяина/группы/других пользователей. Обычно сегмент удаляется тем же процессом, который создал его.

Конфигурация системы определяет минимальный и максимальный размеры сегмента, а также максимальное общее количество и общий объем сегментов в системе. Если система использует страничную подкачку, общий объем сегментов может превышать объем физической памяти, но крайне нежелательно, чтобы он превышал объем файла подкачки. Также ограничено количество сегментов, присоединенных к одному процессу.

Через разделяемую память могут взаимодействовать неродственные процессы. Все, что нужно процессу для присоединения разделяемого сегмента, это соответствующие права доступа для своих идентификаторов пользователя и группы.

Когда сегмент больше не нужен процессу, он отсоединяет его вызовом shmdt(2). Если это не было сделано явным образом, это произойдет при завершении процесса (как по exit(2), так и по сигналу) или при exec(2) . Отсоединение разделяемой памяти похоже на закрытие файла. Отсоединение не удаляет разделяемый сегмент, оно только логически исключает этот сегмент из виртуального пространства данных программы.

В отличие от остальных средств IPC, у разделяемой памяти происходит отложенное удаление. Вызов shmctl(2) с командой IPC_RMID приводит только к тому, что новые процессы не могут присоединиться к этому сегменту, но не к отсоединению этих сегментов у процессов, которые их используют. Запрос на удаление помечает сегмент, но само удаление происходит только когда количество присоединившихся процессов станет равно 0. Флаг ожидания удаления виден как D в выводе команды ipcs(1).

Родительский и порожденные процессы также могут использовать сегменты разделяемой памяти. После fork(2) порожденный процесс наследует присоединенные разделяемые сегменты. Обычные сегменты памяти данных после fork(2) копируются при записи, а сегменты разделяемой памяти остаются общими, так что родитель и потомок видят изменения, вносимые другим процессом, и могут взаимодействовать через них. При этом, в родителе и в потомке сегмент будет отображен на одни и те же виртуальные адреса.

При использовании модифицируемой разделяемой памяти нужны средства координации доступа, позволяющие гарантировать целостность данных и защитить структуры данных, целостность которых временно нарушается. Для такой координации можно использовать семафоры System V IPC, а также семафоры и/или мутексы POSIX Threads.

Создание/получение разделяемой памяти

Для этой цели используется вызов shmget(2). Его параметры:

key — ключ. Значение key может быть получено с использованием ftok(3) или установлено в IPC_PRIVATE.

size - размер сегмента в байтах. На самом деле, размер сегмента округляется вверх до целого числа страниц.

shmflg управляет созданием и правами доступа к сегменту. Допустимые установленные биты:

. IPC_CREAT - если идентификатора разделяемого сегмента с таким ключом нет, или если ключ равен IPC_PRIVATE, создается новый сегмент.

. IPC_EXCL - только совместно с IPC_CREAT. Разделяемый сегмент создается тогда и только тогда, когда он не существует, т.е. этот бит задает исключительное право создания.

. Девять младших бит shmflg задают права доступа. Для разделяемого сегмента это могут быть права чтения и записи.

В Solaris 10, системный вызов shmget(2) ограничен следующими параметрами prctl(1):

project.max-shm-memory
, zone.max-shm-memory
- максимальный суммарный объем сегментов разделяемой памяти в проекте или зоне, соответственно

project.max-shm-ids
, zone.max-shm-ids
- максимальное количество сегментов разделяемой памяти.

Управление разделяемой памятью

Системный вызов shmctl(2) имеет следующие параметры

shmid - это идентификатор разделяемого сегмента.

Команда cmd может принимать одно из следующих значений:

IPC_STAT Копирует информацию о состоянии разделяемого сегмента в структуру, на которую указывает buf.

IPC_SET Устанавливает хозяина, группу и права чтения/записи для хозяина, группы и других пользователей. Значения должны находиться в структуре, на которую указывает buf. Эту операцию могут выполнять только хозяин, создатель и суперпользоватедь.

IPC_RMID Удаляет идентификатор разделяемого сегмента и, возможно, освобождает сам сегмент. Только хозяин, создатель и суперпользователь могут удалить разделяемый сегмент.

Эта команда не делает немедленного освобождения ресурсов, она скорее запрещает дальнейшие shmget(2) и shmat(2). Когда все присоединившиеся процессы закончат работу с сегментом (shm_nattch станет равным 0), сегмент будет удален. Если удаления не сделать, разделяемый сегмент будет существовать до перезагрузки системы, даже если к нему никто не присоединен.

SHM_LOCK Закрепляет разделяемый сегмент в памяти, то есть запрещает его сброс в файл подкачки. Эта команда может привести к исчерпанию физической памяти, поэтому она быть выполнена только суперпользователем.

SHM_UNLOCK Разрешает страничную подкачку для сегмента. Эта команда может быть выполнена только процессом с эффективным ID пользователя, равным суперпользователю.

Дескриптор разделяемого сегмента имеет следующую структуру:

```
struct shmid_ds {
	struct ipc_perm shm_perm;  /* operation permission struct */
	int shm_segsz;  /* size of segment in bytes */
	struct anon_map *shm_amp;  /* segment anon_map pointer */
	ushort shm_lkcnt;  /* number of times it is being locked */
	char pad[2];
	pid_t shm_lpid;  /* pid of last shmop */
	pid_t shm_cpid;  /* pid of creator */
	ulong shm_nattch;  /* used only for shminfo */
	ulong shm_cnattch;  /* used only for shminfo */
	time_t shm_atime;  /* last shmat time */
	long shm_atimfrac;  /* reserved for time_t expansion */
	time_t shm_dtime;  /* last shmdt time */
	long shm_dtimfrac;  /* reserved for time_t expansion */
	time_t shm_ctime;  /* last change time */
	long shm_ctimfrac;  /* reserved for time_t expansion */
	long pad1[4];  /* reserve area  */
};
```

Операции над разделяемой памятью

Операции над разделяемой памятью состоят только в присоединении и отсоединении сегмента. Запись и считывание данных выполняются обычными методами доступа к памяти.

shmat(2) отображает разделяемый сегмент в адресное пространство пользователя.

Аргументы:

shmid является идентификатором разделяемого сегмента.

shmaddr, вместе с shmflg, определяет адрес, по которому будет помещен присоединенный сегмент. Если shmaddr равен нулю, система определяет адрес сама. Если shmaddr не равен нулю, его значение будет взято в качестве адреса. Это должен быть адрес, который система могла бы выбрать для этой цели. В данном курсе мы не изучаем сведений, необходимых для корректного выбора этого адреса. Предоставление выбора адреса системе улучшает переносимость вашей программы.

Разумеется, при этом вы не можете обеспечить, чтобы в разных процессах сегмент был отображен на одни и те же виртуальные адреса, но это вообще сложно обеспечить, ведь в других процессах на требуемые адреса может быть уже отображен другой сегмент или, например, разделяемая библиотека. Единственный способ гарантировать, что сегмент будет отображен на одни и те же адреса — это присоединение сегмента в родительском процессе и его передача одному или нескольким потомкам через fork(2).

shmflg, - Набор флагов. При помощи этого параметра можно сделать сегмент доступным только для чтения, указав флаг SHM_RDONLY. Кроме того, в этом параметре можно указывать флаги, управляющие адресом отображения, которые мы в данном курсе не изучаем.

Процесс может присоединить более одного разделяемого сегмента. Главное ограничение состоит в том, что области виртуальных адресов не могут перекрываться — это относится как к сегментам System V IPC, так и к сегментам mmap(2). Многие Unix-системы имеют административные ограничения на количество сегментов, присоединенных к одному процессу.

shmdt(2) отсоединяет разделяемый сегмент, присоединенный по адресу shmaddr. В отличие от munmap(2), сегменты System V IPC можно отсоединять только целиком. Когда процесс отсоединяет сегмент, поле shm_nattch в дескрипторе этого сегмента уменьшается на 1. Если программа не исполнит shmdt(2) явным образом, exit(2) или exec(2) отсоединит все разделяемые сегменты.

Системные вызовы exec(2) и exit(2), а также завершение процесса сигналом, отсоединяют все разделяемые сегменты, присоединенные к адресному пространству процесса. Когда выполняется fork(2), порожденный процесс наследует все разделяемые сегменты.

Разделяемая память - Родительский процесс

Разделяемая память может использоваться неродственными процессами, однако, программы в этом примере исполняются родительским и порожденным процессами. На следующих страницах приведены две программы, использующие разделяемую память. Одна программа создает сегмент разделяемой памяти и инициализирует его в соответствии с количеством мест в классе. Затем, создаются несколько копий порожденного процесса (fork(2) и exec(2)). Эти порожденные процессы (продавцы) периодически продают места в классе. Для того чтобы исключить одновременный доступ к разделяемому сегменту, используются семафоры System V. Каждый из продавцов завершается когда видит, что в классе больше не осталось места. Родительский процесс ожидает, пока все его подпроцессы завершатся, а затем удаляет семафоры и разделяемый сегмент.

Несмотря на то, что в примере используются родственные процессы, эти процессы исполняют exec(2), то есть мы не можем гарантировать, что сегмент будет отображен на одни и те же адреса. Поэтому в таком сегменте нельзя хранить указатели, вся адресация должна осуществляться при помощи индексации в массивах.

Файл заголовка registration.h используется как инициализатором (родителем), так и продавцами (порожденными процессами). Он описывает формат информации о классе.

registration.h

```
1  struct CLASS {
2       char class_number[6];
3       char date[6];
4       char title[50];
5       int seats_left;
6  };
```

Разделяемая память - Родительский процесс

13-16 Структура показывает, что доступно 15 мест.

29 Пользовательская функция, которая создает, присоединяет и инициализирует разделяемый сегмент памяти информацией о классе.

30 Пользовательская функция, создающая набор семафоров и инициализирующая эти семафоры значением 1.

РАЗДЕЛЯЕМАЯ ПАМЯТЬ - РОДИТЕЛЬСКИЙ ПРОЦЕСС

```
 1 #include"registration.h"
 2 #include<string.h>
 3 #include<unistd.h>
 4 #include<stdlib.h>
 5 #include<sys/types.h>
 6 #include<sys/ipc.h>
 7 #include<sys/sem.h>
 8 #include<sys/shm.h>
 9 #include<stdio.h>
10 #include<memory.h>
11 #include<wait.h>

12 static struct CLASS class = {
13    "1001",
14    "120186",
15    "C Language for Programmers"
16 };
17 #define NCHILD 3
18 static pid_t child[NCHILD];

19 static char*shm_ptr;
20 static intsemid, shmid;
21 static charascsemid[10], ascshmid[10];
22 static char pname[14];
23 static void rpterror(char *),shm_init(void),
24       sem_init(void), wait_and_wrap_up(void);
25 main(int argc, char *argv[])
26 {
27     int i;
28     strcpy(pname,argv[0]);
29     shm_init();
30     sem_init();
```

36-49 Создает три подпроцесса-продавца (shmc[0-2]).

50 Пользовательская функция ждет, когда продавцы продадут все места, а затем удаляет сегмент разделяемой памяти и семафоры.

54 Создает сегмент разделяемой памяти, размера, достаточного чтобы хранить информацию о классе. Поскольку в качестве ключа задается IPC_PRIVATE, идентификатор разделяемой памяти должен быть передан подпроцессам либо как аргумент командной строки, либо как переменная среды. Только процессы с эффективным пользовательским идентификатором хозяина могут использовать его для чтения и записи.

59 Отображает разделяемый сегмент в виртуальное пространство данных процесса, предоставив системе выбор адреса присоединения. Сегмент присоединяется с правом записи (без флага SHM_RDONLY).

64 Копирует информацию о классе в разделяемый сегмент.

65 Создает ASCII представление для значения shmid.

Замечание: Альтернативой ключу IPC_PRIVATE может быть использование родительским процессом своего идентификатора процесса в качестве ключа. Затем подпроцессы смогут получить ключ вызовом getppid(3C).

```
36  for(i=0; i<NCHILD; i++){
37  child[i] = fork();
38  switch(child[i]){
39  case -1:
40  rpterror("fork-failure");
41  exit(1);
42  case 0:
43  sprintf(pname,"shmc%d",i+1);
44  execl("shmc", pname, ascshmid,
45  ascsemid, (char*)0);
46  perror("execl failed");
47  exit(2);
48  }

49  }
50  wait_and_wrap_up();
51}

52static void shm_init(void)
53{
54  shmid=shmget(IPC_PRIVATE,sizeof(class),0600|I
55  if(shmid == -1){
56  perror("shmget failed");
57  exit(3);
58  }
59  shm_ptr = shmat(shmid, 0, 0);
60  if(shm_ptr == (char *)-1){
61  perror("shmat failed");
62  exit(4);
63  }
64  memcpy(shm_ptr,&class,sizeof(class));
65  sprintf(ascshmid, "%d", shmid);
66}
```

69-80 Создает семафор, инициализирует его в 1 и создает ASCII-представление для его идентификатора.

83-90 Ждет, когда все продавцы обнаружат, что больше нет доступных мест.

93 Отсоединяет разделяемый сегмент.

94-95 Удаляет разделяемый сегмент и семафор.

Файл: shmp.c

```
 69 static void sem_init(void)
 70 {
 71   if((semid=semget(IPC_PRIVATE,1,0600|IPC_CREAT))==-1) {
 72   perror("semget failed");
 73   exit(5);
 74   }
 75   if((semctl(semid,0,SETVAL,1)) == -1){
 76   printf("parent: semctl, SETVAL failed\n");
 77   exit(6);
 78   }
 79   sprintf(ascsemid, "%d", semid);
 80 }

 81 static void wait_and_wrap_up(void)
 82 {
 83   pid_t wait_rtn; int w, ch_active = NCHILD;
 84   while( ch_active > 0 ){
 85   wait_rtn = wait( (int *)0 );
 86   for(w=0; w<NCHILD; w++)
 87   if(child[w]==wait_rtn){
 88   ch_active--;
 89   break;
 90   }
 91   }
 92   printf("Parent removing shm and sem\n");
 93   shmdt(shm_ptr);
 94   shmctl(shmid, IPC_RMID, NULL);
 95   semctl(semid, 0, IPC_RMID, 0);
 96   exit(0);
 97 }

 98 static void rpterror(char *string)
 99 {
100   char errline[50];
101   sprintf(errline,"%s %s", string, pname);
102   perror(errline);
103 }
```

Разделяемая память - Порожденный процесс

Родительский процесс создает три подпроцесса. Каждый из них будет продавать места, подсчитываемые переменной в разделяемой памяти. Семафор позволяет избежать одновременных попыток изменения счетчика.

16-20 Проверяет правильность числа аргументов

22 Напоминаем, что родитель использовал в качестве ключа при создании разделяемого сегмента IPC_PRIVATE. Идентификатор разделяемого сегмента передается как аргумент exec(2). Здесь он должен быть преобразован назад из ASCII в целое число.

23 Отображает разделяемый сегмент в адресное пространство процесса.

28 Получает идентификатор семафора из аргумента командной строки.

29 Продает места, пока счетчик оставшихся мест, хранящийся в разделяемой памяти, не станет нулевым. Вместо такого счетчика можно было бы использовать семафор.

30 Отсоединяет разделяемый сегмент.

РАЗДЕЛЯЕМАЯ ПАМЯТЬ - ПОРОЖДЕННЫЙ ПРОЦЕСС

```
 1 #include"registration.h"
 2 #include<stdlib.h>
 3 #include<unistd.h>
 4 #include<sys/types.h>
 5 #include<sys/ipc.h>
 6 #include<sys/sem.h>
 7 #include<sys/shm.h>
 8 #include<stdio.h>
 9 static struct CLASS*class_ptr;
10 static char    *pname;
11 static intshmid, semid, ret;
12 static struct sembuf lock ={ 0, -1, 0};
13 static struct sembuf unlock ={ 0, 1, 0};
14 static void sell_seats(void),rpterror(char*);

15 main(int argc, char *argv[])
16 {
17 if(argc < 3){
18 fprintf(stderr,"Usage:%s shmid semid\n",argv[0]);
19 exit(1);
20 }
21 pname = argv[0];
22 sscanf(argv[1], "%d", &shmid);
23 class_ptr = shmat(shmid, 0, 0);
24 if(class_ptr == (struct CLASS *) -1){
25 rpterror("shmat failed");
26 exit(2);
27 }
28 sscanf(argv[2], "%d", &semid);

29 sell_seats();

30 ret = shmdt(class_ptr);
31 exit(0);
32 }
```

36-60 Каждые десять секунд или меньше пытается продать место. Вызов semop в строках40 и 53 гарантируют, что только один процесс в каждый момент изменяет разделяемую память (переменную seats_left). Когда не осталось мест, цикл заканчивается. Это приводит к возврату из функциии завершению программы.

61-66 Функция rpterror() добавляет имя процесса к строке, которая затем передается библиотечной функции perror(3C).

Файл: shmc.c

```
 36 static void sell_seats(void){
 37 int all_out = 0;
 38 srand( (unsigned)getpid());
 39 while ( !all_out ){ /*loop to sell all seats*/
 40 if(semop(semid,&lock,1) == -1){
 41 rpterror("semop lock failed");
 42 exit(4);
 43 }
 44 if (class_ptr->seats_left > 0){
 45 class_ptr->seats_left--;
 46 printf("%s SOLD SEAT -- %2d left\n",
 47      pname,class_ptr->seats_left);
 48 }
 49 else{
 50 all_out++;
 51 printf("%s sees no seats left\n", pname);
 52 }
 53 ret = semop(semid,&unlock,1);
 54 if (ret == -1) {
 55 rpterror("semop unlock failed");
 56 exit(4);
 57 }
 58 sleep( (unsigned)rand()%10 + 1);
 59 }
 60 }

 61 static void rpterror(char *string)
 62 {
 63 char errline[50];
 64 sprintf(errline,"%s %s", string, pname);
 65 perror(errline);
 66 }
```

Разделяемая память - вывод

Продавцы продают все свободные места. Каждый завершается, когда видит, что больше мест не осталось. Родительский процесс удаляет разделяемую память и набор семафоров после того, как все продавцы завершились.

РАЗДЕЛЯЕМАЯ ПАМЯТЬ - ВЫВОД

```
$ shmp
shmc1 sold seat -- 14 left
shmc2 sold seat -- 13 left
shmc3 sold seat -- 12 left
shmc2 sold seat -- 11 left
shmc3 sold seat -- 10 left
shmc1 sold seat --  9 left
shmc2 sold seat --  8 left
shmc3 sold seat --  7 left
shmc3 sold seat --  6 left
shmc1 sold seat --  5 left
shmc2 sold seat --  4 left
shmc3 sold seat --  3 left
shmc3 sold seat --  2 left
shmc2 sold seat --  1 left
shmc1 sold seat --  0 left
shmc3 sees there are no seats left
shmc1 sees there are no seats left
shmc2 sees there are no seats left
parent removing shm and sem
$
```

Процесс Си-компиляции - Обзор

Цель этого приложения — описать фазы компиляции программ на языке С и научить настраивать компилятор под ваши нужды. Компилятор С преобразует исходный текст на языке С в кодах ASCII в выполняемый объектный код. Процесс компиляции разделен на четыре фазы:

. Препроцессор:

- Осуществляет вставку исходных текстов из других файлов (#include)
- Раскрывает макроопределения (#define)
- Осуществляет условную обработку исходного файла (#ifdef)
- Уничтожает комментарии

. Транслятор (компилятор)

- Проверяет текст на отсутствие синтаксических ошибок
- Преобразует конструкции языка С в конструкции ассемблера
- Выполняет машинно-независимые и машинно-зависимые оптимизации
- Генерирует отладочную информацию.

. Ассемблер

- Преобразует конструкции языка ассемблера в машинные команды
- Генерирует объектный модуль и списки экспорта и импорта (списки внешних символов)
- У некоторых компиляторов этот этап исполняется той же командой, что и трансляция

. Редактор связей

- Осуществляет сборку объектных файлов в загружаемый модуль
- Просматривает библиотеки для разрешения внешних ссылок

Для Solaris 10 доступны два компилятора: GNU Compiler Collection (GCC) и SunStudio. Оба компилятора доступны бесплатно, но на разных условиях: от компилятора GCC также доступны исходные тексты. Оба компилятора включают в себя компиляторы С (с поддержкой диалектов Kernigan & Ritchie, ANSI C, C99) и C++. Компилятор SunStudio также поддерживает директивы параллельного программирования OpenMP.

Для запуска компилятора языка С используется команда cc (SunStudio) или gcc (GCC). Для запуска компилятора C++ используются команды CC или g++. В дальнейших примерах мы будем использовать название команды cc; когда будут обсуждаться особенности других форм запуска компилятора, это будет оговариваться отдельно.

Команда cc - это управляющая программа, которая последовательно вызывает с помощью fork и exec другие программы, реализующие фазы процесса компиляции. Каждой фазе соответствует свои опции, и у каждой фазы свои сообщения об ошибках. Раздел ФАЙЛЫ на странице Руководства cc(1) указывает, где может быть найдена каждая исполняемая фаза. В общем случае, фазы процесса компиляции не должны вызываться явно. Их вызов осуществляет команда cc(1). Каждая фаза использует файлы или программные каналы для передачи своего вывода следующей фазе.

Формат команды cc

Команда cc имеет формат:

```
cc [опции] file1.c [file2.c …]
```

Обзор поддерживаемых опций будет приведен далее в этом приложении. Команде компилятора необходимо указать один или несколько файлов. В зависимости от расширения файла, компилятор автоматически определяет, что с ним следует делать: файлы с расширением .c обрабатываются всеми фазами компиляции, начиная с препроцессора и транслятора, файлы с расширением .s — начиная с ассемблера, файлы с расширениями .o, .a и .so сразу передаются редактору связей.

Простой способ собрать программу из нескольких модулей исходного текста — это передать компилятору список всех этих модулей. Однако это приводит к тому, что при каждом вызове такой команды все исходные файлы будут компилироваться заново. При разработке и отладке программы обычно ее приходится перекомпилировать много раз; обычно при каждой пересборке меняется только часть файлов, часто даже только один файл. Фаза трансляции занимает много времени, поэтому невыгодно перекомпилировать те файлы исходного текста, которые не менялись.

При компиляции программ, состоящих из большого количества файлов исходных текстов, обычно каждый файл компилируют с ключом -c (этот ключ приводит к тому, что редактор связей не вызывается, зато сохраняется объектный модуль в файле с расширением .o), а затем вызывают редактор связей отдельным вызовом команды cc. Это позволяет сэкономить время компиляции, вызывая компилятор только для тех файлов, которые были изменены, и используя старые объектные модули от тех файлов, которые не изменялись. Обычно для координации такой раздельной компиляции используют программу make(1), которая будет кратко описана далее в этом приложении.

Процесс Си-компиляции - Фаза препроцессора

Первая фаза компиляции — препроцессор языка C. Выходные данные препроцессора — это еще ASCII текст (операторы языка C). Все, что делает препроцессор, это текстовые вставки и замены

Следующие опции позволяют завершить процесс компиляции после препроцессора:

-P - выходные данные записываются в файл name.i.

-E - выходные данные записываются в стандартный вывод.

Замечание: Команда, исполняющая эту фазу, описана на странице cpp(1) Справочного руководства пользователя.

Директивы препроцессора

Все операторы препроцессора - это полные строки. Директивы препроцессора начинаются с символа # и заканчиваются <NEWLINE> или началом комментария. Можно соединить несколько строк вместе, набрав обратную косую черту (\) в конце соединяемых строк. Перед символом # или после него может стоять один или несколько пробелов или табуляций, но не должно быть символов, не являющихся пробелами

Зарезервированные препроцессором слова (например, define или include), расположены непосредственно справа от #, возможно (но не обязательно) отделенные от этого символа одним или несколькими пробелами.

В директиве #include может быть задано абсолютное или относительное путевое имя файла. Такой файл называют вставляемым файлом, файлом заголовка или файлом определений. Обычно имя файла заголовка имеет расширение .h, хотя препроцессор не выдвигает на этот счет никаких требований. В C++ также используют расширения .hpp или файлы заголовка без расширений.

#include "header.h" файл ищется в директории исходного файла, затем в стандартной директории /usr/include

#include <header.h> файл ищется только в /usr/include

#include "/usr/header.h" файл задается путевым именем

Вставляемые файлы обычно содержат директивы #define для констант и макроопределений, а также директивы #include для других файлов, описания типов, структур, классов и шаблонов C++, объявления глобальных переменных и функций. Достаточно часто вставляемые файлы используются для описания в одном месте структур и параметров, используемых в нескольких файлах исходных текстов.

Директивы #define часто используются, чтобы сделать программу более читабельной. Эти имена имеют такую же форму, как идентификаторы языка C. Чтобы выделить эти имена в тексте, их обычно набирают заглавными буквами. Область действия имени — от соответствующего #define до конца файла или до команды #undef, которая отменяет макроопределение.

Макроопределение (макрос) имеет форму:

#define macro_name(param_1,...,param_n) token_string.
Списка параметров может не быть. Не должно быть пробела между macro_name и символом ‘(‘.Макросы и символьные имена могут быть определены в терминах ранее определенных макросов и/или символов. Макроопределения также называют символами препроцессора.

Встретив macro_name в тексте, препроцессор заменяет его на token_string, осуществляя подстановку значений аргументов. Важно понимать, что производится чисто текстовая подстановка без какого либо анализа семантики или синтаксиса с точки зрения языка C. Если количество переданных аргументов не соответствует заданному в объявлении, препроцессор выдает ошибку. Препроцессор не проверяет соответствия выполненной подстановки синтаксису языка C.

Если SYMBOL дважды определяется с помощью #define без #undef между ними, то препроцессор может выводить предупреждение "SYMBOL redefined". При этом действует последнее определение.

Условная компиляция

Во время фазы препроцессора можно проверить какое-либо условие с помощью директив #if, #ifdef или #ifndef. Все три формы завершаются директивой #endif. Дополнительно, между началом и концом конструкции может появиться #else .

Условие оператора #if expression считается истинным, если expression имеет ненулевое значение.

Условие директивы #ifdef identifier считается истинным, если identifier определен при помощи директивы #define (при этом не имеет значения, как именно он был определен) и не был отменен при помощи директивы #undef.

Условие оператор #ifndef identifier считается истинным, если identifier не был определен.

#ifdef identifier — это сокращенная форма записи #if defined(identifier), а #ifndef — соответственно, #if !defined(identifier).

Если условие директивы #if, #ifndef или #ifdef истинно, операторы между #if и соответствующими #else или #endif обрабатываются (передаются на вход компилятору), а операторы между #else и #endif (если существуют) заменяются пустыми строками или директивой #line. Это необходимо, чтобы компилятор правильно определял номера строк исходного файла при выдаче отладочной информации и сообщений об ошибках.

Если проверяемое условие ложно, операторы между #if и соответствующим #else или #endif заменяются пустыми строками. Выполняются, если существуют, операторы между #else и #endif.

Выражение, используемое в #if может содержать в качестве переменных макроопределения (но не переменные языка С) и любые операции языка Си (логические, арифметические, отношения). Помните, что #if SYMBOL и #if SYMBOL != 0 вырабатывают одинаковый результат.

В следующем примере Z получит значение 10, если во время работы препроцессора X больше чем 3, или Y имеет ненулевое значение. X и Y — ранее определенные параметры:

```
#if X > 3 || Y
#define Z 10
#else
#define Z 20
#endif
```

Условная компиляция привносит гибкость в процесс компиляции. С ее помощью во время компиляции могут быть установлены определенные параметры программы, например, внедрен или удален отладочный код, или откомпилированы разные версии кода для разных компиляторов, ОС или аппаратных платформ.

Заранее определенные символы препроцессора

Препроцессор автоматически определяет ряд символов, которые могут быть использованы в директивах условной компиляции. Эти символы определяют

. Тип аппаратной архитектуры, например, i386 или sparc

. Версию ОС

. Версию компилятора и поддерживаемые компилятором диалекты языка

. Имя текущего обрабатываемого файла: __FILE__ заменяется именем текущего исходного файла, заключенным в двойные кавычки.

. Номер обрабатываемой строки: __LINE__ заменяется целым номером текущей строки в формате целого десятичного числа.

При задании некоторых ключей компиляции, например, -mt у компилятора SunStudio или -threads у GCC, препроцессор также может определять дополнительные символы, указывающие, что компилируется многопоточная программа.

Эти заранее определенные символы могут быть использованы, чтобы сделать программы на Си более переносимыми.

Опции препроцессора

Препроцессор вызывается автоматически на первой фазе процесса компиляции. Следующие опции распознаются командой компилятора (cc) и передаются препроцессору.

-P Работает только препроцессор (вывод пишется в файл с расширением .i)

-E Работает только препроцессор (вывод пишется в файл стандартного вывода)

-Dname=def Определяет идентификатор. Эквивалентно директиве #define name def в начале файла. Например: cc -DSIZE=10 -DSYM=5 -DDEBUG prog.c.

Замечание: если =def опущено, это эквивалентно -Dname=1

-Uname Отменяет определение идентификатора name. Используется для отмены автоматически определяемых препроцессорных символов. Например: cc -E -Ui386 -Du370 prog.c.

-Idir Добавляет dir в список директорий, в которых будут искаться вставляемые файлы.

Для директивы #include "file.h" поиск ведется сначала в директории, где хранятся исходные файлы, затем в dir, и в последнюю очередь в стандартной директории /usr/include.

Для директивы #include <file.h> поиск ведется сначала в dir и затем в стандартной директории.

Пример:

```
cc -I$HOME/myproject/include -I..prog.c
```

-C Не удаляет комментарии

Сообщения об ошибках препроцессора

Препроцессор может выдавать сообщения об ошибках. Например:

1. Использование макроса со слишком малым или слишком большим количеством аргументов приведет к выводу сообщения argument mismatch.

2. Неправильное написание ключевого слова препроцессора (например, #inclde) вызовет сообщение undefined control.

3. Использование имени несуществующего файла или неправильно написанное имя файла в #include приведет к сообщению Can't find include file...

4. Отсутствие двойных кавычек или < > вокруг имени файла в #include вызовет сообщение bad include syntax.

Процесс компиляции - Фаза транслятора

Второй фазой процесса компиляции является транслятор или компилятор. Входом для него является программа, обработанная препроцессором. Выход — программа на ассемблере, "родном" для целевого компьютера.

Транслятор выполнит:

. Проверку отсутствия синтаксических ошибок

. Компилятор C++ также выполняет раскрытие шаблонов

. Машинно-независимые (например, оптимизацию арифметических выражений, вынос инвариантов цикла, раскрытие хвостовой рекурсии) и машинно-зависимые (размещение переменных на регистрах, переупорядочение операций для лучшего использования конвейеров процессора) оптимизации

. Если нет фатальных ошибок, преобразует операторы C в команды ассемблера.

Опции транслятора

Современные трансляторы C/C++ имеют большое количество опций, управляющих выдачей предупреждений, используемыми оптимизациями, поддержкой разных диалектов языка, включением инструментации для профилирования, поиска утечек памяти и др.. В зависимости от комплекта поставки, компилятор может включать поддержку нескольких аппаратных архитектур. Так, SunStudio для x86 может генерировать код как для 32-разрядной архитектуры x86, так и для 64-разрядной x64, а также проводить оптимизации под разные модели процессоров соответствующих архитектур. Эти опции различаются у разных трансляторов и перечислены в руководствах по соответствующим компиляторам.

Стандартные опции, поддерживаемые всеми современными компиляторами для Unix, включают:

-g Генерирует таблицу символов для отладки с помощью символьного отладчика dbx(1). Эта опция также передается ассемблеру и редактору связей.

-O Вызывает оптимизатор объектного кода, который уменьшает размер и увеличивает скорость программы посредством перемещения, объединения и удаления кода.

-S Не вызывает ассемблер. Оставляет результат в файле с расширением .s. Этот файл будет содержать команды языка ассемблера, "родного" для целевого компьютера.

-fPIC Генерирует позиционно-независимый код для использования в разделяемых библиотеках

Процесс Си-компиляции - Фаза ассемблера

Фаза ассемблера конвертирует исходные тексты на языке ассемблера в объектный код.

Некоторые компиляторы исполняют эту фазу той же программой, что и фазу трансляции, и/или вместо текста на языке ассемблера передают ассемблеру внутреннее представление кода. Тем не менее, фаза ассемблирования логически присутствует во всех трансляторах.

Сообщения об ошибках на этой стадии выдают только плохо разработанные компиляторы. Ассемблер — машинно-зависимый язык. Следовательно, опции ассемблера тоже машинно-зависимы. Если компилятор поддерживает несколько аппаратных архитектур или несколько разных моделей целевого процессора, опции этапа трансляции, управляющие выбором модели процессора, разумеется, передаются и ассемблеру. Также, ассемблеру передается опция -g, указывающая, что необходимо генерировать отладочную информацию.

Опция этапа ассемблирования, поддерживаемая всеми компиляторами — это опция -c, сигнализирующая, что компиляцию нужно завершить на этапе ассемблирования. При использовании этой опции, не вызывается редактор связей, а объектный файл не удаляется. Вместо этого, созданный объектный модуль размещается в файле с расширением .o.

Объектный файл содержит бинарные данные (таблицы символов, таблицы перемещений, машинный код и, возможно, отладочную информацию) и не может просматриваться текстовым редактором. Для просмотра данных в объектном файле можно использовать утилиту elfdump(1).

Процесс компиляции — Редактор связей

Финальной стадией процесса компиляции является редактор связей или линкер.

Редактор связей создает исполняемый файл, объединяя объектные файлы, выполняя перемещение кода и разрешая внешние ссылки. Например, если один из объектных файлов содержит вызов функции printf(3C), редактор связей будет искать эту функцию по всем объектным файлам и библиотекам и выдаст ошибку сборки, если не найдет.

Объектный код, используемый для создания исполняемой программы, находится в объектных файлах, а также разделяемых и архивных библиотеках.

По умолчанию, кроме пользовательских объектных файлов и библиотек, в состав программы включается файл /usr/lib/crt1.o. Этот файл содержит адрес точки входа программы и код, инициализирующий среду исполнения языка C и вызывающий определенную пользователем функцию main. При сборке программы для исполнения в нестандартном окружении, например, в качестве ядра ОС, может быть необходимо задать другой стартовый файл при помощи соответствующих опций редактора связей.

Стандартная библиотека языка C /lib/libc.so просматривается без явного запроса пользователя. Редактору связей можно задать просмотр других библиотек, кроме libc.so, или отключить просмотр этой библиотеки. Например, при сборке ядра ОС, не следует подключать стандартную библиотеку языка C, потому что её функции содержат системные вызовы, а ядро не может выполнять системные вызовы, во всяком случае, теми же средствами, какими это делает libc.

Архивные библиотечные файлы имеют расширение .a и создаются командой ar(1). По существу, такие файлы представляют собой просто коллекции объектных модулей. При подключении архивной библиотеки, из неё извлекаются только те модули, в которых определены переменные и функции, используемые вашей программой. Эти модули включаются в состав исполняемого файла вашей программы, поэтому сборка с архивной библиотекой называется также статической сборкой.

Архивные файлы имеют формат, описанный на странице руководства ar(4), состоящий из заголовка архивного файла, таблицы символов архива, за которым следуют заголовок и объектный код для каждого элемента.

Файлы разделяемых библиотек имеют расширение .so (shared object). При сборке с такими библиотеками, редактор связей определяет, в какой именно библиотеке находится требуемая функция, и генерирует таблицу PLT (Procedure Linkage Table), включаемую в состав вашей программы. Таблица PLT содержит информацию, что функцию printf надо искать в библиотеке libc.so. При этом, сама библиотека в состав вашей программы не включается, а присоединяется к вашей программе при её запуске. Связь между функцией printf и её кодом в libc.so устанавливается только при первом вызове этой функции. Такая сборка называется динамической сборкой.

Разделяемые библиотеки формата ELF допускают также настоящую динамическую сборку, когда библиотека не подключается редактором связей, а открывается во время исполнения функцией dlopen(3C). Затем, программа может найти адрес нужной ей функции по имени функцией dlsym(3C) и вызывать эту функцию по адресу (в языке C адрес функции называется «указатель на функцию»).

Если при работе редактора связей не случилось ошибок, результат помещается в файл a.out.

Некоторые опции редактора связей

Ниже показаны наиболее часто употребляемые опции редактора связей:

-lname Определяют способ сокращенного задания имени архивного или разделяемого библиотечного файла. Он обозначает подключение библиотеки libname.so или libname.a в зависимости от параметров ключей -d и -B. По умолчанию, директории /lib и /usr/lib просматриваются при поиске библиотеки.

-d y|n Определяет статическую или динамическую сборку. Параметр y, действующий по умолчанию, указывает динамическую сборку. При этом редактор связей сначала пытается подключить разделяемую версию библиотеки (файл .so), а если она не будет найдена, то архивную версию. При параметре n используется статическая сборка и редактор связей ищет только архивные библиотеки.

Внимание: начиная с Solaris 9, в поставку системы не входит архивная версия библиотеки libc, поэтому статически собрать программу на языке C штатными средствами невозможно.

-B static|dynamic Позволяет переключать статический или динамический режим сборки для отдельных библиотек. Может указываться в командной строке несколько раз и действует на все следующие опции -l до конца строки или следующей опции -B.

-Ldir Добавляет директорию dir в начало списка директорий, в которых следует искать библиотеки, задаваемые опцией -l, перед /lib и /usr/lib. Дополнительные директории для поиска библиотек можно также задавать переменной среды LD_LIBRARY_PATH.

-oname Определяет имя name для результирующего исполняемого файла. Имя по умолчанию - a.out.

-s Удаляет отладочную информацию: номера строк исходного кода и информацию, содержащуюся в таблице символов, что позволяет уменьшить размер исполняемого файла, но затрудняет использование dbx(1).

-g Включает отладочную информацию в исполняемый файл, что позволяет использовать символьные отладчики. Отладочную информацию можно удалить из собранного исполняемого файла командой strip(1).

-G Генерация разделяемой библиотеки (файла с расширением .so). Несовместима с ключом -dn. Процедура сборки разделяемых библиотек подробнее описана в странице системного руководства ld(1) и в документе «Linker and libraries guide» на сайте http://docs.oracle.com.

Примеры

Примеры: Три исходных файла компилируются и исполняемый файл переименовывается в prog:

```
cc prog.c f.c g.c -o prog
```

Компоновка объектного файла из заранее скомпилированых файлов .o и функций из /lib/libPW.a:

```
cc -om m.c f1.o -lPW
```

Просмотр текущей и HOME директории для поиска архивного файла libxyz.a:

```
cc p.c -L. -L$HOME -lxyz -o p
```

Утилита make

При разработке и отладке программ, состоящих из большого числа исходных модулей, возникает желание оптимизировать процесс компиляции, перекомпилируя только те файлы, которые изменялись с момента предыдущей компиляции. Для этого предназначена утилита make(1). Для Solaris доступны три версии make: традиционная версия утилиты Unix System V, GNU Make и распределенная версия make (dmake), входящая в поставку пакета SunStudio. Далее будут рассматриваться возможности, общие для всех версий этой утилиты.

make определяет правила, по которым надлежит компилировать файлы, на основе специального текстового файла. Если запустить make без параметров, она пытается взять эти правила из файла Makefile в текущей директории. Если правила размещены в файле с другим именем, то make следует запускать с опцией -f, например make -f my-makefile. Далее в этом разделе мы будем называть входной файл Makefile.

Makefile может содержать определения переменных и правил. Переменные кратко рассматриваются далее.

Правила разделены одной или несколькими пустыми строками. Каждое правило описывает команду для компиляции какого-либо файла и имеет вид:

```
target: dependencies
  command
```

В начале строки перед command обязан присутствовать хотя бы один пробел или табуляция.

target представляет собой имя целевого файла, например, исполняемого или объектного. Dependencies представляет собой список разделенных пробелами имен файлов, от которых целевой файл зависит. Для типичного программного модуля на языке C, целевым файлом будет объектный модуль (.o-файл), а файлами зависимостей — исходный файл на языке C и заголовочные файлы, которые этот файл включает директивой #include. Не следует включать в список зависимостей системные заголовочные файлы, нужно включать только файлы вашего программного проекта, которые могут быть изменены в ходе разработки.

Command — это одна или несколько команд shell, которые нужно исполнить для «делания» файла, то есть для получения целевого файла из файлов зависимостей. Русскоязычные программисты обычно называют «делание» сборкой. Для файлов на языке C сборка, скорее всего, заключается в компиляции с ключом -c, однако в реальных проектах возможны и более сложные ситуации. Например, если ваша программа использует автоматически генерируемый при помощи утилиты yacc(1) синтаксический анализатор, ее Makefile может включать правило для генерации текста на языке C при помощи yacc(1) из файла на входном языке yacc.

Команда может состоять из нескольких строк, каждая из которых исполняется как отдельная команда shell. При успешном завершении сборки, последняя команда должна возвращать код 0. Иначе make решит, что сборка завершилась неудачей и прекратит дальнейшую работу.

Примеры правил и Makefile

Для .o-файла, получаемого из .c-файла, правило может выглядеть так:

```
main.o: main.c functions.h
	cc -c main.c
```

Для исполняемого файла, получаемого из нескольких .o-файлов, правило может выглядеть так:

```
hello: main.o factorial.o hello.o
	cc main.o factorial.o hello.o -o hello
```

Весь Makefile в целом может выглядеть так:

```
hello: main.o factorial.o hello.o
	cc main.o factorial.o hello.o -o hello

main.o: main.c functions.h
	cc -c main.c

factorial.o: factorial.c functions.h
	cc -c factorial.c

hello.o: hello.c functions.h
	cc -c hello.c

clean:
	rm -rf *o hello
```

При сборке с таким файлом, утилита make работает следующим образом: если в командной строке указан параметр с именем цели, например, make clean, утилита пытается собрать указанную цель, в данном случае цель clean. Если цель в командной строке не указана, make пытается собрать первую цель в файле, в данном случае hello.

При сборке каждой цели, make сравнивает дату модификации целевого файла с датами модификации файлов зависимостей. Если целевой файл имеет дату модификации позже, чем файлы зависимостей, считается, что ничего делать не надо. Если же целевого файла нет или какой-то из файлов зависимостей был модифицирован позже целевого файла, соответствующий целевой файл собирается. Если какого-то из файлов зависимостей нет и нет правила, как его сделать, make выдает ошибку. Например, если в текущей директории нет файла functions.h, make выдаст ошибку "Don't know how to make functions.h".

Проверка осуществляется рекурсивно: так, при сборке цели hello, make сначала проверяет, надо ли сделать все его зависимости, то есть файлы main.o, factorial.o и hello.o.

Если команда сборки какой-то из целей выдает ошибку (код завершения, отличный от 0), make останавливает процесс компиляции и не пытается собирать остальные цели. На самом деле, make имеет опции, выключающие это поведение и заставляющие make собрать все цели, какие можно. При сборке из командной строки это не всегда удобно, так как при прекращении сборки последней строкой выдачи, скорее всего, будет сообщение об ошибке, из-за которой сборка остановилась, а при продолжении сборки это сообщение может быть сложно найти в общей выдаче.

Комментарии и переменные

В Makefile, кроме правил, могут также содержаться комментарии и переменные.

Комментарии — это строки, начинающиеся с символа #, заканчивающиеся символом <NEWLINE> и содержащие произвольный текст. Комментарии игнорируются утилитой make.

Есть любители вставлять в начало Makefile магическую последовательность #!/bin/make

Эта строка является комментарием с точки зрения make, но обозначает такой файл как интерпретируемую программу с интерпретатором /bin/make с точки зрения системного вызова exec(2) и, следовательно, с точки зрения shell. Такой Makefile можно сделать исполняемым командой chmod +x и запускать из shell без явного вызова make, командой ./Makefile.

Переменные — это текстовые строки вида NAME=VALUE, похожие на переменные среды или переменные shell. К значениям переменных можно обращаться, используя синтаксис $(NAME). Переменные можно использовать в любых местах Makefile. Значения переменных можно задавать внутри самого Makefile или в командной строке make, передавая в качестве параметра строку NAME=OTHER_VALUE. При этом, значения, заданные в командной строке, «сильнее» значений, заданных в файле.

Также, запустив make с опцией -e, можно заставить его брать значения переменных из переменных среды.

Пример файла с использованием комментариев и переменных:

```
# Если пользователь захочет собирать программу при помощи GCC,
# он может запустить сборку командой make CC=gcc
# или командами export CC=gcc; make -e
CC=cc
CFLAGS=-Wall -O
OBJECTS=main.o factorial.o hello.o

hello: $(OBJECTS)
	$(CC) $(OBJECTS) -o hello

main.o: main.c functions.h
	$(CC) -c $(CFLAGS) main.c

factorial.o: factorial.c functions.h
	$(CC) -c $(CFLAGS) factorial.c

hello.o: hello.c functions.h
	$(CC) -c $(CFLAGS) hello.cpp

# Обратите внимание, что эта версия Makefile гораздо умнее предыдущей
# Она удаляет только объектные файлы, относящиеся к проекту
# а не все объектные файлы в текущем каталоге.
clean:
	rm -rf $(OBJECTS) hello
```

Детальное описание make(1) может быть найдено в системном руководстве man и в многочисленных учебниках и туториалах, доступных в сети Интернет. Подробная документация по dmake доступна на сайте http://docs.oracle.com.

Автоматическая генерация Makefile

make предоставляет ряд средств, упрощающих создание больших Makefile для больших проектов. Самая сложная задача при поддержании Makefile — это правильное отслеживание зависимостей, в первую очередь — отслеживание, какие .c файлы включают какие заголовочные файлы. Для решения этой задачи в рамках проекта GNU была реализована утилита automake (не входит в стандартную поставку Solaris, но может быть собрана из исходных текстов).

Большинство современных интегрированных сред разработки на C/C++, например, SunStudio, включают простые для использования средства автоматической генерации и поддержки Makefile. К сожалению, генерируемые такими средами Makefile не очень-то удобны для последующей ручной модификации.

Приложение. История ОС семейства Unix

В 80-е годы мэйнстримные пользователи предпочитали юниксу VAX/VMS.
Сейчас они предпочитают юниксу Windows NT.
Какая часть этого сообщения вам непонятна?
Реплика из конференции ***USENET*** **alt.comp.os.windows.advocacy**

Обширное и бурно развивающееся семейство Unix оказало огромное идейное влияние на развитие операционных систем в 80-е и 90-е годы XX века. Генеалогия систем семейства опубликована на сайте [**perso.wanadoo.fr**] и слишком обширна для того, чтобы ее можно было полностью привести в книге.

Применения систем семейства крайне разнообразны, начиная от встраиваемых приложений реального времени, включая графические рабочие станции для САПР и геоинформационных систем, и заканчивая серверами класса предприятия и массивно параллельными суперкомпьютерами. Некоторые важные рыночные ниши, например, web-хостинг, передачу почты и другие структурные сервисы Интернета, системы семейства занимают практически монопольно. Еще одна ниша, в которой на 2012 год Unix-системы лидируют с далеким отрывом — это портативные устройства, в первую очередь, смартфоны и планшеты. Безусловными лидерами соответствующих сегментов являются ОС Android и iOS, основанные на ядрах Linux и Apple OS X соответственно.

Родоначальником семейства следует, по-видимому, считать не первую версию Unix, а Multics, совместно разрабатывавшуюся в 1965—1969 годах General Electric и Bell Laboratories. За это время General Electric выделило подразделение, занимавшееся работами над Multics и аппаратной платформой для нее (GE-645), в отдельную компанию Honeywell.

Multics была первой из промышленных систем, предоставлявших:

8. создание процессов системным вызовом `fork`;
9. страничную виртуальную память;
10. отображение файлов в адресное пространство ОЗУ;
11. вложенные каталоги;
12. неструктурированные последовательные файлы;
13. многопользовательский доступ в режиме разделения времени;
14. управление доступом на основе ограниченных ACL (колец доступа).

Multics оказала огромное влияние не только на разработчиков Unix — значительные следы идейного влияния этой системы прослеживаются также в RSX-11 и VAX/VMS фирмы DEC. Последние Multics-системы были доступны в Интернете в 1997 году.

В 1969 году Bell Laboratories отказалась от работ над Multics и начала разработку собственной ОС для внутренних нужд. По-видимому, основной причиной этого шага было осознание несоответствия между амбициозными целями проекта Multics (ОС была весьма требовательна к ресурсам и могла работать только на больших компьютерах Honeywell), в то время как материнской компании Bell Labs — American Telephone & Telegraph — требовалась единая операционная среда, способная работать на различных миникомпьютерах, используемых в подразделениях телефонной сети США.

В Bell Laboratories был объявлен внутренний конкурс на разработку переносимой ОС, способной работать на миникомпьютерах различных поставщиков. К проекту были предъявлены следующие основные требования:

15. многоплатформенность;
16. вытесняющая многозадачность;
17. многопользовательский доступ в режиме разделения времени;

18. развитые телекоммуникационные средства.

Один из участников работ над Multics, К. Томпсон, разработал крайне упрощенное ядро ОС, названное UNIX, для миникомпьютера PDP-7. К 1972 году К. Томпсон и Д. Ритчи переписали ядро системы в основном на языке С и продемонстрировали возможность переноса ОС на миникомпьютеры PDP-11. Это обеспечило выполнение всех требований конкурса, и UNIX была признана основной платформой для вычислительных систем, эксплуатируемых в AT&T.

Легенды доносят до нас более колоритные детали ранних этапов истории новой системы. Так, одна из очень популярных легенд, излагаемая в той или иной форме несколькими источниками, утверждает, что история UNIX началась с разработанной для Multics игровой программы под названием Star Wars (звездные войны — сама эта программа и даже правила игры до нас не дошли). Уволенный из группы разработчиков Multics за разгильдяйство (я привожу наиболее радикальный вариант легенды, не заботясь о его согласовании с историческими фактами), Томпсон занялся "оживлением" неиспользуемой PDP-7, стоявшей в углу машинного зала. Оживление заключалось в написании ядра ОС, реализующего подмножество функциональности Multics, достаточное, для того чтобы перенести и запустить его любимые Star Wars [Бах 1986].

Первые версии UNIX были рассчитаны на машины без диспетчера памяти. Процессы загружались в единое адресное пространство. Ядро системы размещалось в нижних адресах ОЗУ, начиная с адреса 0, и называлось *сегментом реентерабельных процедур*. Реентерабельность обеспечивалась переустановкой стека в момент системного вызова и запретом переключения задач на все время исполнения модулей ядра. На машинах с базовой адресацией выполнялось перемещение образов процессов по памяти и сброс образа процесса на диск (задачный своппинг).

Работа — а особенно разработка программного обеспечения — в режиме разделенного времени на машине с незащищенной памятью — это весьма своеобразное занятие. По воспоминаниям одного из участников команды разработчиков Unix, перед запуском неотлаженной версии программы необходимо было кричать "a out" и делать паузу, чтобы остальные пользователи могли сохранить редактируемые файлы.

В Multics и современных системах Unix `fork` реализуется посредством копирования страниц при модификации. Ранние версии UNIX физически копировали образ процесса. Большая часть (по некоторым оценкам, до 90%) `fork` немедленно продолжается исполнением системного вызова `exec`, поэтому одной из первых оптимизаций, придуманных в университетских версиях системы, было введение системного вызова `vfork`. Порожденный этим вызовом процесс исполнялся на самом образе родителя, а не на его копии. Создание нового образа процесса происходило только при исполнении `exec`. Для того чтобы избежать возможных проблем взаимоисключения (например, при вызове нереентерабельных функций стандартной библиотеки), исполнение родителя приостанавливалось до тех пор, пока потомок не выполнит `exec` или не завершится.

Выделяют [Баурн 1986] следующие отличительные особенности системы.

19. Порождение процессов системным вызовом `fork`, который создает копию адресного пространства и пользовательской области (user area, так в Unix называются структуры данных ядра, связанные с процессом) процесса.

20. Результат завершения процесса хранится в его дескрипторе и может быть считан только родителем. В списке процессов такой дескриптор выглядит как процесс в специальном состоянии, называемом зомби (zombie).

21. Процессы-сироты (продолжающие исполнение после завершения родителя) усыновляются процессом с идентификатором, равным 1.

22. Процесс с идентификатором 1 запускается при загрузке системы (по умолчанию это /bin/init) и запускает все остальные обязательные задачи в системе. Наличие такого процесса иногда объявляют необходимым и достаточным критерием для причисления той или иной системы к семейству Unix.
23. Древовидная структура пространства имен файловой системы: дополнительные ФС монтируются в те или иные точки корневой ФС и идентифицируются затем точкой монтирования, а не именем устройства, на котором расположены.
24. Файлы рассматриваются ОС как неструктурированные потоки байтов и не типизованы на уровне ОС (в частности, на уровне ОС нет деления файлов на записи).
25. Файловая система поддерживает множественные имена файлов в виде жестких и, у более поздних версий, символических связей.
26. Отложенное удаление файлов: если процесс открыл файл, а другой процесс его удалил, то первый процесс может продолжать работу с файлом, и физическое удаление происходит только после того, как первый процесс его закроет.
27. Лозунг "всё — файл" (который, впрочем, последовательно реализован только в экспериментальной системе Plan 9) — в реальных Unix системах практически всегда присутствуют объекты, к которым не применимы файловые операции: переменные среды, структуры данных ядра в ранних версиях (позднее они были оформлены в виде объектов псевдофайловой системы /proc), объекты SysV IPC и примитивы взаимоисключения POSIX Thread Library в современных системах.
28. Своеобразный командный язык, основанный на широком применении переназначения ввода/вывода и конвейеров (последовательностей задач, соединенных трубами).

Некоторые из перечисленных особенностей были позаимствованы другими ОС. Так, переназначение ввода/вывода и конвейеры реализуются командными процессорами ОС семейства CP/M, начиная с MS DOS 3.30. cmd.exe — командный процессор OS/2 и Windows NT/2000/XP — даже правильно понимает конструкцию `cmdline | more 2>&1`. Стандартные командные процессоры UNIX портированы практически на все современные ОС. Функциональный аналог Korn Shell включен в стандартную поставку Windows 2000.

Жесткие связи файлов (но без отложенного удаления) поддерживаются FCS2 (файловой системой VAX/VMS), NTFS и jfs при использовании под OS/2. Точки монтирования (подключение дополнительных ФС в дерево каталогов существующей ФС) реализованы в Windows 2000 и Toronto Virtual File System для OS/2.

Системный вызов `CreateProcess` Windows NT/2000/XP может имитировать семантику `fork`: для этого достаточно передать в качестве имени программы для нового процесса пустой указатель.

Библиотека EMX для OS/2 реализует `fork` с помощью собственного обработчика страничного отказа.

П1.1. Распространение UNIX

Глупый пингвин робко прячет,
умный гордо достает
Неизвестный автор

AT&T в 70-е годы XX века была "естественной" монополией в области телекоммуникаций. Этот статус гарантировался законодательным запретом деятельности других телекоммуникационных компаний на территории США. В обмен на этот статус AT&T вынуждена была подчиняться ряду регуляторных мер, в частности — ей было запрещено выходить на другие, кроме телекоммуникационного, рынки, в том числе на рынок программного обеспечения. Однако разработчики UNIX чувствовали, что их системе суждено гораздо более привлекательное будущее, чем внутренний стандарт крупной компании.

С 1973 года одна из дочерних компаний AT&T, Western Electric, дала разрешение на использование UNIX в некоммерческих целях. Началось распространение системы в университетах США. Наибольший вклад в распространение и развитие университетской версии системы внес университет Беркли, в котором было создано специальное подразделение — BSD (Berkeley Software Distribution).

В BSD Unix было включено множество ценных нововведений, таких как:

29. сегментная (на старших моделях PDP-11) и страничная (на VAX-11/780) виртуальная память;
30. раздельные адресные пространства процессов и выделенное адресное пространство ядра;
31. абсолютные загрузочные модули формата a.out;
32. примитивная форма разделяемых библиотек;
33. усовершенствования механизма обработки сигналов;
34. управление сессиями и заданиями в пределах сессии.

Самое важное нововведение было сделано в начале 80-х годов, когда в рамках работ по проекту DARPA сетевое программное обеспечение ARPANet было перенесено с TOPS/20 на BSD Unix. Вскоре сетевой стек BSD стал референтной реализацией (реализация, на совместимость с которой тестируют все остальные) того, что ныне известно как семейство протоколов TCP/IP.

В 1980 году было решено начать коммерческое распространение системы на несколько необычных принципах: AT&T предоставляла сторонним коммерческим фирмам (естественно, за плату) лицензии на использование исходных текстов ядра и основных системных утилит текущей версии UNIX, а уже эта сторонняя коммерческая фирма (дистрибьютор) строила на основе полученных и самостоятельно разработанных компонентов законченную систему — с инсталляционной программой, системой управления пакетами и т. д. — и занималась ее продажей конечным пользователям и сопровождением. Таким образом была создана специфическая бизнес-модель распространения ОС семейства UNIX, хорошо знакомая пользователям Linux.

Первым из коммерческих распространителей стала фирма Microsoft, продававшая ядро UNIX v7 в составе ОС Microsoft Xenix. Xenix поставлялся почти для всех популярных в то время 16-разрядных миникомпьютеров и микропроцессорных систем [Дейтел 1987]. Как и BSD Unix, Xenix использовал виртуальную память и имел отдельное адресное пространство для ядра. В 1983 году торговая марка Xenix и весь дистрибьюторский бизнес был передан фирме SCO в обмен на долю акций последней.

К середине 80-х воспитанное на университетских версиях UNIX поколение студентов пришло в промышленность. Началось бурное развитие *рабочих станций (workstation)* —

мощных 32-разрядных персональных компьютеров, как правило, оснащенных страничными или сегментными диспетчерами памяти. Лицензия BSD допускала построение на основе кода BSD коммерческих систем без каких-либо ограничений, в том числе и без денежных выплат разработчикам ядра. Благодаря этому, а также благодаря техническому совершенству ядра BSD Unix, последнее оказалось гораздо более привлекательным, чем ядро AT&T, поэтому основная масса поставщиков рабочих станций строили свои ОС на основе BSD Unix. Это привело к быстрому и неконтролируемому размножению систем, называвших себя Unix, и при этом имевших значительное количество несовместимостей — дополнительных или, наоборот, нереализованных системных вызовов, ошибок, "документированных особенностей" и т. д.

В 1984 году AT&T заключила с федеральным антимонопольным комитетом США соглашение, в соответствии с которым компания должна была выделить локальные телефонные сети в отдельные компании, и согласовала планы создания конкурентной среды на рынке междугородней связи и выделения в отдельные компании подразделений, не имеющих отношения к телекоммуникациям. Долгосрочные результаты этого соглашения до сих пор являются предметом горячих дебатов среди юристов и экономистов, но важным с нашей точки зрения является то, что AT&T смогла напрямую заняться продажами и поддержкой программного обеспечения. На рынок вышло ядро Unix System V — первая поддерживаемая версия ядра AT&T UNIX.

В 1987 году вышла версия UNIX System V Release 3, включавшая в себя асинхронные драйверы последовательных устройств (STREAMS), универсальный API для доступа к сетевым протоколам (TLI), средства межпроцессного взаимодействия (семафоры, очереди сообщений и сегменты разделяемой памяти), ныне известные как *SysV IPC,* BSD-совместимые сокеты и ряд других "BSDизмов" [Робачевский 1999]. SVR3 в то время воспринималась как этапная ОС, однако дальнейшее развитие системы вынуждает нас отнести ее, скорее, к переходным версиям.

В этом же году AT&T и Sun Microsystems заключили стратегическое соглашение о разработке перспективного ядра UNIX System VI, которое должно было обеспечить совместимость с System V, BSD Unix и Xenix и, тем самым, консолидировать возникший зоопарк Unix-систем.

Не имея финансовой поддержки со стороны локальных телефонных сетей, AT&T оказалась вынуждена заняться поисками средств для поддержки деятельности по развитию UNIX. Во второй половине 80-х было сделано несколько попыток взыскать лицензионные отчисления с поставщиков коммерческих систем на основе BSD Unix. Нельзя сказать, чтобы эти попытки были особенно последовательными и успешными, но они породили ряд инициатив по разработке "лицензионно чистой Unix-системы".

Среди этих инициатив необходимо назвать следующие.

35. Микроядро BSD Mach.
36. Minix А. Танненбаума.
37. Проект Р. Столлмэна GNU (GNU Not Unix — рекурсивная аббревиатура) [**www.gnu.org**].
38. Консорциум OSF (Open Software Foundation — фонд открытого программного обеспечения).

П1.2. Микроядро

Концепция микроядра с технической точки зрения подробно рассматривается в *разд. 8.3*. С коммерческой (если уместно говорить о коммерческих целях разработки свободно распространяемого ПО) точки зрения BSD Mach был попыткой убить одновременно двух зайцев — совместить переписывание ядра BSD Unix для достижения лицензионной чистоты с изменением архитектуры этого ядра.

Микроядерная архитектура позволила бы избежать самой одиозной черты традиционных систем Unix — однопоточного (или, точнее, кооперативно многозадачного) ядра, и сделала бы систему пригодной для использования в задачах реального времени. Проект Mach не имел полного успеха — основные ветви BSD до сих пор используют традиционное монолитное ядро. Наиболее успешная ОС, основанная на микроядре Mach — это Darwin (Unix-подсистема MacOS X).

Однако идея микроядра и сам термин получили широкое распространение. Микроядерную архитектуру имеет UNIX System V Release 4. Кроме того, на самостоятельно разработанном микроядре основана своеобразная ОС реального времени, часто относимая к семейству Unix — QNX.

Основные работы над ядром BSD UNIX пошли в другом направлении: подсистемы, которые AT&T считал основанием для требования лицензионных выплат, переписывались с нуля, но архитектура системы в целом пересмотру не подвергалась. Этот процесс был в основном завершен к 1994 году, и современные ветви BSD по-прежнему имеют монолитную архитектуру.

П1.3. Minix

Minix был разработан А. Танненбаумом, преподавателем университета Врийе (Vrije University) в Амстердаме [**www.cs.vu.nl** minix]. Это компактная система, созданная для учебных целей, способна работать на 16- и 32-разрядных микропроцессорах, причем не только самостоятельно, но и будучи скомпилирована и запущена в качестве задачи под "нормальной" ОС Unix. Первая версия системы имела очень консервативную (чтобы не сказать — архаичную) архитектуру, очень близкую к архитектуре ранних версий UNIX. Minix 2.0, выпущенный в 1996 году, основан на микроядре и поддерживает страничную виртуальную память на процессорах x86.

Основной целью разработки было создание системы, которая, с одной стороны, была бы работоспособна и могла бы продемонстрировать основные архитектурные концепции современных многозадачных ОС, а с другой — достаточно проста, чтобы студенты могли полностью в ней разобраться. Второе требование фактически исключало возможность доработки ОС до состояния, в котором она могла бы стать коммерчески применима.

Наиболее известен прямой потомок Minix, Linux. Первая версия Linux разрабатывалась путем переписывания ядра Minix модуль за модулем, что значительно упростило Л. Торвальдсу отладку системы. Воспоминаниями об этих временах в современном Linux является поддержка файловой системы minix, название основной ФС — ext2fs (Second Extended File System — расширенная [по сравнению с minix] файловая система, вторая версия) и реликты кода Minix в некоторых модулях.

П1.4. GNU Not Unix

Проект GNU был начат преподавателем Массачусетского технологического института Р. Столлмэном и имел целью разработку полностью свободной операционной системы. "Полная свобода" гарантировалась своеобразным лицензионным соглашением, так называемым *copyleft* — текст современной версии этого соглашения, GPL (General Public License — общая публичная лицензия), размещается в заголовке каждого файла исходного текста программных продуктов, распространяемых в соответствии с данной лицензией [**www.fsf.org**].

Вопросы о необходимости, целесообразности и допустимости этой схемы распространения ПО, а также о моральных, юридических, экономических, социальных и других последствиях ее применения до сих пор являются предметом жарких дебатов [**www.tuxedo.org** homesteading]. Тем не менее, в рамках деятельности FSF (Free Software Foundation — фонд свободного программного обеспечения) было разработано немало высококачественного и полезного ПО, прежде всего — коллекция компиляторов GNU CC (GNU Compiler Collection), включающая компиляторы C/C++, Fortran и Ada, текстовый редактор (и по совместительству интегрированная среда разработки) GNU Emacs, функциональные эквиваленты стандартных утилит UNIX и ряд других программ и утилит. Основной целью проекта объявлялась разработка GNU HURD, весьма амбициозной микроядерной ОС.

В 1996 году публике была представлена крайне сырая альфа-версия системы. К тому времени Linux уже шествовал по планете победным шагом и отвлек на себя всех специалистов, способных участвовать в разработке ядра и согласных распространять результаты своей деятельности на условиях GPL. Наверное, из-за этого HURD не привлек внимания ни разработчиков, ни бета-тестеров. С тех пор до момента публикации этой книги не поступало ни новых версий, ни объявления о прекращении работ. По-видимому, следует признать, что проект HURD завершился провалом.

П1.5. Open Software Foundation

Консорциум OSF, в который вошли DEC, IBM, Hewlett-Packard и ряд менее известных поставщиков рабочих станций и серверов, был создан в 1988 году. Его деятельность началась с принятия и публикации стандарта POSIX.1 (Portable OS Interface based on uniX — переносимый интерфейс ОС, основанный на Unix).

В рамках OSF начались работы по разработке ядра Unix-совместимой ОС, по архитектуре в целом аналогичной UNIX SVR3 (монолитное ядро с поддержкой STREAMS и некоторыми особенностями BSD). К моменту завершения разработки уже был выпущен UNIX System V Release 4.2 (Destiny), достигший всех целей, заявлявшихся в проекте UNIX System VI, и консорциум фактически распался. DEC Unix (известный также как OSF/1) оказался чрезмерно тяжеловесным, не имел коммерческого успеха и, возможно, сыграл немалую роль в судьбе компании DEC. Значительно более счастливой оказалась судьба IBM AIX, лишь частично основанной на коде OSF.

В 1996 году то, что осталось от OSF, слилось с консорциумом X/Open, деятельность которого по стандартизации Unix-систем имела гораздо больший успех.

П1.6. X/Open

Консорциум X/Open [**www.opengroup.org**] был основан в 1990 году и имел гораздо более широкий состав, чем OSF, включая в себя практически всех производителей и поставщиков Unix-систем и ряд образовательных учреждений. Вместо разработки новой версии системы консорциум занялся разработкой стандартов, которым система любого поставщика должна была удовлетворять, чтобы иметь право называться Unix.

Одним из главных достижений в деятельности этого консорциума следует считать разработку и публикацию спецификаций X Window — протокола распределенной графической оконной системы, который стал основой графического интерфейса практически всех Unix-систем.

В 1993 году фирма Novell, которая к тому времени приобрела авторские права и команду разработчиков AT&T, передала консорциуму торговую марку UNIX™. С тех пор консорциум выдавал право носить название UNIX™ системам, которые проходили тесты на совместимость с текущей версией спецификаций. Было также, наконец-то, решено, что торговой маркой является только UNIX (все буквы заглавные), но не Unix. Стандартизация оказала крайне благотворное влияние на рынок Unix-систем и приложений для них, практически устранив различия, существенные для разработки прикладного ПО, между наиболее распространенными системами семейства.

Поскольку сертификация была платной, некоммерческие версии системы, такие как ветви BSD и Linux, ее не проходили. Неожиданным результатом сертификационной политики консорциума стало право OS/390 называться UNIX™ после прохождения тестов в 1998 году [**www.opengroup.org** xu007].

П1.7. UNIX System V Release 4

А больше всего было стыдно Максима и Федора, которые мало того, что пьянь политурная, так еще и смотрят с сочувствием.
В. Шинкарев

Обещанная в 1987 году UNIX System VI вышла на рынок в 1989 году под названием UNIX SVR4. Микроядерная система обеспечивала полную бинарную совместимость с SVR3, бинарную же совместимость с 16- и 32-разрядными Xenix на процессоре x86, и совместимость на уровне исходных текстов с BSD Unix v4.3 [Хевиленд/Грей/Салама 2000]. Заявленная цель консолидации всех основных ветвей Unix в единой системе была полностью достигнута. Sun Microsystems приступила к переводу своих пользователей на Sun OS 5.*x* (ныне известна как Solaris), основанную на ядре SVR4.

Бизнес-модель распространения UNIX SVR4 была похожа на традиционную для AT&T: Unix System Laboratories сохраняла авторские права на ядро системы, но лицензировала право на использование ядра и отдельных компонентов другим компаниям, которые, в свою очередь, строили на основе этих компонентов свои поддерживаемые системы — Sun Solaris, Silicon Graphics IRIX, Novell UnixWare и некоторые менее известные.

Версия SVR4 была этапной — она включала в себя следующие компоненты:

39. многопоточное микроядро;
40. класс планирования реального времени (процессы с этим классом планирования имеют приоритет выше, чем нити ядра);
41. новый формат загрузочного модуля ELF (Executable and Linking Format), обеспечивавший удобную работу с разделяемыми и динамическими библиотеками;
42. динамическое подключение и отключение областей свопинга;
43. динамическую загрузку и выгрузку модулей ядра;
44. многопоточность в пределах одного процесса (так называемые LWP (Light Weight Processes — легкие процессы));
45. псевдофайловую систему /proc, обеспечивающую контролируемый доступ к адресным пространствам других процессов и структурам данных ядра;
46. оптимизирующий компилятор ANSI C, по качеству кода не уступающий GNU C.

В 1991 году подразделение AT&T, занимающееся развитием и поддержкой UNIX, было выделено в отдельное предприятие, USL (UNIX System Laboratories). Дальнейшая история этой организации представляет неплохой сюжет для романа: в 1992 году USL была приобретена фирмой Novell — тогдашний CEO (Chief Executive Officer — главный администратор) компании Р. Нурда пытался сформировать линию продуктов, способную конкурировать со всеми предложениями Microsoft. В 1993 году права на торговую марку UNIX были переданы консорциуму X/Open. В 1995 году акционеры Novell, испуганные перспективой конфронтации с Microsoft, сняли Нурду с поста CEO и стали распродавать его приобретения. В частности, USL и лицензионные соглашения с распространителями UNIX SVR4 (Sun, Silicon Graphics, Microport и др.) были проданы фирме SCO. Нурда основал компанию Caldera, основным бизнесом которой стало распространение и поддержка Linux. 7 мая 2000 года в тексте этой истории была поставлена… ну, скорее всего, не точка, но весьма важный знак препинания: Caldera приобрела компанию SCO вместе со всеми правами на SVR4 [**www.sco.com**].

Впрочем, завершение этой истории, видимо, уже не за горами. В начале XXI века дела Caldera/SCO пошли под гору. Отчасти это было связано с коммерческими неудачами SGI Irix и постепенным уходом Hewlet Packard с рынка RISC-серверов — лицензионные отчисления SGI за SVR4 и HP за код SVR3, использовавшийся в составе HP/UX, составляли значительную часть доходов SCO.

В 2005 году компания Caldera переименовалась в The SCO Group и решила повторить

неудачную попытку Bell Laboratories и взыскать с пользователей Linux деньги за использование кода, якобы заимствованного из System V.

Основные претензии при этом предъявлялись компании IBM, которая получала лицензии на разные версии Unix System V и различные его подсистемы несколькими разными путями. IBM использует довольно много кода Unix System V R3 в IBM AIX. Кроме того, в первой половине 90-х годов IBM приобрела компанию Sequent, которая делала массивно параллельные многопроцессорные системы на основе процессоров x86, работавшие под управлением специализированной версии SCO Unix (также основанной на System V Release 3). По утверждениям SCO Group, сотрудники IBM перенесли часть кода в модули ядра Linux; затем эти модули были включены в основное дерево исходных текстов ядер Linux 2.4 и 2.6.

Судебный процесс был приостановлен в связи с параллельным судебным иском к компании Novell по вопросу о том, какие именно права были переданы SCO в 1995 году. Оба судебных процесса (SCO Group против Novell и SCO Group против IBM) затянулись. В 2007 году компания The SCO Group объявила о реорганизации в соответствии с главой 11 Кодекса США о банкротстве. Это, фактически, останавливало все судебные иски в отношении компании. В августе 2012 года компания SCO Group объявила о ликвидации в соответствии с главой 7 того же Кодекса.

Развитие основной ветви системы (System V Release 4.2) практически прекратилось, активным развитием системы продолжал заниматься только Sun Microsystems.

Sun Solaris, построенный на ядре System V Release 4 поддерживает три основные аппаратные архитектуры — 32битные процессоры SPARC v8, 64-битные процессоры SPARC v9 и x86 в 32-разрядном режиме. На время подготовки второго издания книги к печати каких-либо внятных обещаний по поводу поддержки архитектуры IA-32e (64-разрядное расширение архитектуры x86) не давалось. С каждым релизом Solaris делается попытка прекратить поддержку x86, но под давлением пользователей эта поддержка сохраняется.

Многие из подсистем Solaris были значительно усовершенствованы по сравнению с оригинальной SVR4.

- В Solaris, по сравнению с традиционными Unix-системами, значительно изменен процесс управления драйверами устройств. При загрузке система создает дерево каталогов, соответствующее иерархии периферийных шин и устройств, обнаруженных на этих шинах, а также отдельные каталоги для установленных в системе псевдоустройств. На машинах Sun этот каталог создается на основе информации, предоставляемой загрузочным ПЗУ. На машинах x86 соответствующую информацию собирает вторичный загрузчик, имитирующий также и другие функции ПЗУ Sun. Корень этой иерархии находится в каталоге /devices. На каждую из обнаруженных периферийных шин создается каталог, в котором, в свою очередь, создаются каталоги и файлы, соответствующие устройствам. В каталогах для шин PCI, имена каталогов и файлов устройств соответствуют PCI ID этих устройств. Файлы в традиционном для Unix-систем каталоге /dev представляют собой символические ссылки на иерархию /devices.
- В Solaris 7 была добавлена поддержка сетевого протокола IPv6.
- В Solaris 8 была реализована журнальная версия файловой системы UFS, позволившая администраторам серверов перейти на журнальную ФС без переразметки дисков.
- Также в Solaris 8 были реализованы контрольные точки, сохранявшие состояние файловой системы на определенный момент (этот механизм относительно подробно описывается в *разд. 11.4.3*).
- В Solaris 9 была реализована система "проектов" (projects), которая позволяет выделять ресурсы (например, гарантированную долю процессорного времени или ОЗУ) группам процессов.

- Sun Solaris 8 при работе на серверах семейства Sun Fire поддерживает динамическое подключение, исключение и горячую замену процессорных модулей и модулей ОЗУ, а также одновременную работу нескольких копий Solaris на разных процессорах одного вычислительного комплекса с динамическим перераспределением ресурсов (процессоров, ОЗУ, дисков) между виртуальными системами.
- В дистрибутив Solaris 9 был включен набор из наиболее популярных утилит проекта GNU, таких как командный процессор bash. Лицензия GPL, на основе которой распространяются эти утилиты, не запрещает включать программы в коммерчески распространяемые дистрибутивные пакеты, она лишь не позволяет запрещать кому бы то ни было дальнейшее распространение этих программ.
- В Solaris 10 была реализована новая файловая система ZFS, основанная на принципе write anywhere *(см. разд. 11.5)*.

На всем протяжении 90-х годов, архитектура ядра не подверглась существенным изменениям. Как и MVS полутора десятилетиями раньше, UNIX достиг совершенства в своем роде и нуждается не в новой архитектуре, а только в оптимизации существующего кода (ядро SVR4 несколько тяжеловато по сравнению с монолитными ядрами BSD и Linux) и развитии отдельных подсистем.

В 2005 году Sun Microsystems опубликовала значительную часть исходных текстов ядра Solaris в рамках программы Open Solaris [**opensolaris.org**] на условиях CDDL (Common Development and Distribution License — общая лицензия на разработку и распространение). Обсуждение отличий этой лицензии от других популярных "свободных" лицензий, таких как GPL, увело бы нас далеко от основной темы книги.

В 2010 году компания Sun Microsystems была приобретена компанией Oracle. Вскоре после этого, Oracle объявил о прекращении сотрудничества с сообществом OpenSolaris, об изменении условий лицензирования Solaris 10 и Solaris 11 и о том, что новые версии исходного кода Solaris не будут публиковаться. На 2012 год, Solaris 10 и 11 распространяются на следующих условиях:

На компьютерах производства Sun Microsystems (компания Oracle продолжает производить оборудование под этим брэндом) можно использовать все версии Solaris бесплатно.

На компьютерах других производителей, Solaris 10 и 11 можно использовать при условии приобретения контракта на поддержку у Oracle; при этом сама ОС считается бесплатной.

На сайте Oracle доступны дистрибутивы и образы виртуальных машин Solaris Express. Эти образы можно скачивать и использовать бесплатно при условии регистрации на веб-сайте и согласия с лицензией; лицензия включает в себя условие, что данные ОС могут использоваться только для разработки, тестирования и демонстрации программного обеспечения.

Сайт opensolaris.org на 2012 год продолжает быть доступным, но дерево исходных текстов не обновлялось с 2010 года.

П1.8. Linux

В 1991 году Л. Торвальдс, в тот момент — студент университета Хельсинки, приступил к разработке того, что ныне известно как Linux — полноценной операционной системы, основанной на исходных кодах Minix и распространяемой на условиях GPL [**www.linux.org**].

В 1992 году была выпущена первая публичная версия системы. К тому времени сообщество пользователей и разработчиков freeware уже успело устать от задержек выпуска GNU HURD и обещаний Столлмэна и приняло новый проект с огромным энтузиазмом. Ряд компаний (RedHat, Caldera, SuSe и множество других) начал распространение коммерчески поддерживаемых дистрибутивов ОС на основе ядра Linux, воспроизводя таким образом бизнес-модель распространения AT&T UNIX начала 80-х.

Вышедшее в 1997 году ядро Linux 2.0 имело вполне приемлемую по стандартам коммерческих ОС надежность и почти все наиболее прогрессивные черты других Unix-систем.

47. Загрузочные модули и разделяемые библиотеки формата ELF.
48. Псевдофайловую систему /proc.
49. Динамическое подключение и отключение своп-файлов.
50. Длинные файлы (64-разрядные — длина файла и смещение в нем).
51. Многопоточность в пределах одного процесса (POSIX thread library).
52. Поддержку симметричной многопроцессорности.
53. Динамическую загрузку и выгрузку модулей ядра.
54. Стек TCP/IP, совместимый с BSD 4.4, с поддержкой IPSec, фильтрации пакетов и др.
55. SysV IPC.
56. Бинарную совместимость с UNIX System V на процессорах x86 (iBCS — Intel Binary Compatibility Standard) и, позднее, на SPARC и MIPS.
57. Поддержку задач реального времени (класс планирования реального времени в монолитном Linux невозможен; такие задачи загружаются как модули ядра).

Linux перенесен практически на все 32- и 64-разрядные машины, имеющие диспетчер памяти, начиная от Amiga и Atari и заканчивая IBM System/390 и IBM z/90. Бинарные эмуляторы Linux включены в состав Solaris/SPARC и FreeBSD.

Ядро Linux быстро развивается и еще не достигло той степени "зрелости" и стабильности, которая характерна для SVR4 и ветвей BSD. В частности, поэтому среднее количество опасных ошибок, обнаруживаемых в системе за фиксированный интервал времени, существенно выше, чем в двух указанных ОС; производительность отдельных подсистем также оставляет желать лучшего. Однако положение довольно быстро улучшается и, по-видимому, в обозримом будущем Linux может стать одним из технологических лидеров отрасли.

П1.8.1 Android

ОС для мобильных устройств Android компании Google основана на ядре Linux. На 2012 год, Android представляет самую массовую (по количеству компьютеров, на которых она запущена) ОС семейства Unix, и, возможно, вообще самую распространенную ОС для 32-разрядных процессов.

Компания Android была основана в Калифорнии в 2003 году. Компания занималась разработкой в секрете, сообщая только, что занимается разработкой ПО для мобильных устройств. Разработка финансировалась на личные деньги основателей. В 2005 году, когда продукт был еще не готов, деньги у основателей кончились, и компания вместе с продуктом

была приобретена Google.

В 2007 году Google опубликовал спецификации платформы и объявил о создании консорциума Open Handset Alliance, в который вошли производители сотовых телефонов Samsung и HTC, операторы сотовой телефонии Nextel и T-Mobile, а также производители комплектующих Texas Instruments и Qualcomm. Спецификации платформы включали в себя требования к процессору, загрузочному монитору и периферийному оборудованию, необходимому для загрузки ОС, а также список дополнительного оборудования, которое может поддерживаться платформой (акселерометры и датчики положения, приёмники GPS и др.). Первый телефон на основе Android, HTC Dream, вышел в 2008 году. В 2010 году продажи устройств на основе Android впервые обогнали продажи основного конкурента — iPhone. На 2012 год устройства на основе Android поставляются несколькими десятками компаний, в том числе Samsung, Sony, HTC, Alcatel, Motorola. Платформа лидирует с довольно большим отрывом как по количеству вновь продаваемых устройств, так и по оценкам количества реально используемых телефонов.

ОС основана на компонентах, поставляемых на основе лицензий с открытым исходным кодом. Ядро поставляется на условиях лицензии GPL. Большая часть кода, исполняемого в пользовательском режиме — на условиях Apache License. Некоторые из изменений, внесенных в ядро Android, были внедрены в основную ветвь кода ядра Linux, но всё-таки ядра Linux и Android представляют собой независимые и постепенно все дальше и дальше расходящиеся ветви.

Google не сразу публикует последнюю версию исходных текстов; новые версии системы и обновления сначала становятся доступны членам альянса, так что они могут первыми выйти на рынок с устройствами, поддерживающими новую версию ОС. Доступны независимые от производителя оборудования сборки Android, наиболее известная из которых — CyanogenMod. Установка собственных сборок ОС на устройства представляет некоторую проблему для неподготовленного пользователя, потому что загрузочный монитор не очень-то оптимизирован для установки ОС конечным пользователем; фактически, для перепрошивки телефона нужно иметь настольный компьютер с установленным комплектом для кросс-разработки ПО для Android. Многие производители считают перепрошивку устройства основанием для снятия с гарантии; ряд устройств поставляются с защищенным загрузчиком, делающим установку собственных сборок ОС затруднительной или даже практически невозможной.

Android поддерживает портативные компьютеры (телефоны и планшеты) с процессорами ARM и x86 с сенсорным экраном. На 2012 год, подавляющее большинство Android-устройств на рынке были изготовлены на основе процессоров разных производителей с архитектурой ARM. Состав системы за пределами ядра, то есть набор поставляемых вместе с системой программ, исполняемых в пользовательском режиме, сильно отличается от типичной Unix-системы.

Традиционные Unix-системы используют в качестве графической оболочки среду, основанную на протоколе X 11 (X Window). Android имеет собственную нестандартную графическую оболочку и собственный фреймворк для разработки приложений с событийно-ориентированным графическим интерфейсом. Поскольку прямое портирование приложений с настольных компьютеров на портативные устройства (телефоны и планшеты) затруднительно по целому ряду причин, это решение можно признать разумным. Для портативных устройств нужно новое прикладное ПО, и его логично писать с использованием новых API.

Традиционные системные сервисы Unix, а также командные интерпретаторы и утилиты для работы с файлами в стандартную поставку системы вообще не входят.

Разработчики прикладного ПО могут разрабатывать приложения двух типов:

- «родные» (native), представляющие собой загружаемые файлы формата ELF, содержащие машинный код для целевой платформы. «Родные» приложения имеют доступ ко всем возможностям ядра Android, но привязаны к определенной аппаратной архитектуре. Впрочем, формат пакета допускает включение загрузочных модулей для разных архитектур.
- «управляемые» (managed), представляющие собой байт-код для интерпретатора и/или JiT-компилятора Dalvik.

Dalvik представляет собой виртуальную машину, во многом аналогичную виртуальной машине языка Java. Dalvik предоставляет управляемую память со сборкой мусора и управляемые и ограниченные интерфейсы для вызова «родных» программных интерфейсов, таких, как рисование на экране или доступ к файловой системе. Главным средством генерации байт-кода для Dalvik является компилятор языка, в основном совпадающего по синтаксису с Java. Также, интерфейсы, доступные программисту (функции стандартной библиотеки, фреймворк для разработки графических приложений) очень похожи на язык Java. В технологическом отношении, Dalvik сильно отличается от Sun/Oracle JVM. Он использует другой формат байт-кода, другие технологии JiT-компиляции и сборки мусора и др., но с точки зрения прикладного программиста Google постарался свести отличия от Java к минимуму.

Пока разработчик Java, компания Sun Microsystems, была независимой корпорацией, они поддерживали разработку и развитие платформы, официально заявляя, что это обогащает Java-экосистему. Однако после приобретения Sun Microsystems компанией Oracle, в 2010 году Oracle подал в суд на Google, обвиняя последний в нарушении авторских прав на спецификацию языка и интерфейсов. В 2012 году суд Северного Округа штата Калифорния постановил, что Google несомненно, нарушил авторские права Oracle на функцию проверки границ индекса массива, длина которой (вместе с комментариями) составляет 16 строк. Также, было признано, что Google использовал разработанные Oracle спецификации языка и платформы, но это попадает под определенные в американском законодательстве критерии «честного использования». Oracle объявил, что будет обжаловать это решение в суде более высокого уровня. Автор выражает надежду, что здравый смысл, в конце концов, восторжествует.

Главной проблемой при разработке прикладных программ под Android считается сильная фрагментация рынка. Устройства производятся разными производителями, имеют разный размер и разрешение экрана, разную производительность основного ЦПУ и графического сопроцессора, работают под разными версиями ОС. Многие производители не обновляют ОС на старых (а некоторые даже и не на очень старых) моделях устройств или выпускают обновления через значительное (иногда более полугода) время после выхода версии ОС; возможность установки пользователем собственной (в том числе и самостоятельно собранной) сборки ОС приводит к комбинаторному увеличению количества возможных сочетаний версии ОС и оборудования. Статистика приобретения дополнительного контента заставляет предположить, что многие покупатели телефонов Android, особенно дешевых моделей, используют их, главным образом, в качестве простых телефонов.

П1.9. MacOS X

> ***Гамп:*** *Лейтенант Дэн уговорил меня вложить деньги в какую-то фруктовую компанию. А потом он мне позвонил и сказал, что о деньгах можно больше не беспокоиться. А я сказал, что это хорошо! Одним [поводом для беспокойства] меньше.*
> Э. Рот, к.ф. "Форрест Гамп".

Недавнее и, пожалуй, неожиданное прибавление в семействе Unix — это MacOS X.

Первые версии MacOS — операционной системы для компьютеров Apple Macintosh — представляли собой кооперативно многозадачную ДОС с незащищенной памятью и событийно-ориентированным интерфейсом.

В 1991 году в MacOS 7 была реализована защита памяти, что существенно повысило устойчивость ОС, но система осталась кооперативно многозадачной. В начале 90-х это еще казалось приемлемым, но к середине 90-х, особенно после выхода на рынок Windows 95, всем, в том числе руководству Apple, стало очевидно, что так жить нельзя.

Тем не менее общие сложности перехода от кооперативной к вытесняющей многозадачности привели к тому, что MacOS версий 8 и 9 оставались кооперативными.

Наряду с MacOS, компания Apple поставляла также систему A/UX, основанную на ядре BSD Unix, но предоставлявшую графический пользовательский интерфейс, аналогичный MacOS, обеспечивающую ограниченную совместимость с бинарными приложениями для MacOS. Проблемы с совместимостью приложений, разумеется, сильно сужали сферу применения A/UX, который использовался практически только на серверах.

Историю того, что сейчас известно под названием MacOS X, следует отсчитывать не от основной линии MacOS, а от проекта Стива Джобса NextSTEP.

В 1985 году один из основателей компании Apple Стив Джобс, после конфликта с новым президентом компании Джоном Скалли, основал собственную компанию и назвал ее NeXT. Вместе с Джобсом из Apple ушли несколько ведущих разработчиков Macintosh и MacOS, большинство из которых участвовали также в работах над неудачным проектом Apple Lisa.

Компания NeXT разработала собственную аппаратную платформу на основе микропроцессора Motorola 68030. В качестве интерфейса для подключения периферийных устройств использовалась шина NuBus, та же, что и в Apple Macintosh. Первая версия платформы имела характерный корпус в виде черного куба и так и называлась NeXTcube.

Для этих компьютеров была разработана ОС NEXTSTEP. Ядро ОС было основано на BSD Mach. Наиболее важными особенностями новой ОС были графический пользовательский интерфейс, основанный на технологии Display PostScript, и объектно-ориентированный инструментарий для разработки приложений на основе языка Objective C (объектно-ориентированное расширение C, альтернативное C++).

PostScript — язык описания векторных изображений, первоначально разработанный компанией Adobe в качестве языка управления лазерными принтерами. В действительности, PostScript представляет собой полнофункциональный (т. е. эквивалентный машине Тьюринга с конечной памятью) язык программирования.

Использование PostScript для отрисовки изображений на дисплее, разумеется, должно было сильно упростить разработку приложений WYSIWYG. Впрочем, особенности работы с дисплеем в многооконной графической среде потребовали внесения специфических дополнений и расширений в язык.

Компьютеры NeXT вызвали большой интерес у специалистов в области компьютерной техники, но рынок принял их довольно прохладно. Машины были существенно дороже, чем Apple Macintosh сопоставимой конфигурации и при этом уступали им по производительности. Из-за низкой производительности они не могли конкурировать и с

рабочими станциями на основе RISC-процессоров, так что президент Sun Скотт МакНили ехидно заметил в 1989 году, что это "компьютер с неправильным процессором по неправильной цене". Привлечь значительных разработчиков прикладного ПО на новую платформу не удалось.

Тем не менее, одна из самых значимых прикладных программ XX столетия была впервые разработана именно для NEXTSTEP. В 1991 году сотрудник CERN Тимоти Джон Бернерс-Ли, пытаясь расширить функциональность гипертекстового протокола gopher, разработал язык разметки HTML, протокол HTTP, простой веб-сервер и первый веб-браузер. И сервер, и браузер первоначально были разработаны для NEXTSTEP и, позднее, портированы на другие платформы. Настоящую популярность новая технология приобрела после выхода в 1993 году веб-сервера NCSA HTTPD и браузера NCSA Mosaic, но слава изобретателя WWW всё равно остаётся за Бернерс-Ли. Некоторые источники утверждают, что какие-то элементы кода веб-браузера WWW, разработанного Бернерсом-Ли, в какой-то степени был использован в коде NCSA Mosaic. Поскольку сервер и браузер Бернерса-Ли распространялись на условиях public domain, юридически это вполне возможно, но о каком именно коде идёт речь, по открытым источникам определить затруднительно. Так или иначе, настоящую популярность Веб приобрел именно после того, как серверное и клиентское ПО было портировано из-под NEXTSTEP под более распространенные на тот момент ОС.

В течение первой половины 90-х годов разработчики NeXT предпринимали несколько попыток сохранить жизнь платформе, главным образом путем конвергенции ее с существующими и уже утвердившимися на рынке ОС. Наиболее близкой к удаче попыткой следует признать проект OPENSTEP, разрабатывавшийся совместно с компанией Sun.

Наконец, в 1997 году акционеры Apple уговорили Джобса вернуться на пост CEO компании. При этом компания NeXT была поглощена Apple, и начались работы по конвергенции NEXTSTEP с MacOS. Первые попытки такой конвергенции — проекты Copland и Taligent (разрабатывавшийся совместно с IBM) — анонсировались с большой шумихой, но с технической точки зрения закончились неудачей.

Наконец, в 1998 году была показана публике первая демо-версия того, что двумя годами позже вышло на рынок под названием MacOS X.

Система представляет собой весьма любопытный сплав разнородных компонентов с совершенно различными условиями лицензирования. Важная часть пакета — это Unix-подсистема, известная как Darwin. Это система с открытыми исходными текстами, публикуемыми на условиях BSD License, основанная на BSD Mach 3.0. Ядро реализует внутреннюю вытесняющую многопоточность и обеспечивает интерфейс системных вызовов FreeBSD, однако использует собственную подсистему ввода/вывода (I/O Kit), не имеющую прямых аналогов в системах семейства BSD Unix. Для мультимедийных приложений предоставляется класс планирования реального времени.

Наиболее важное отличие I/O Kit от подсистем ввода/вывода других ОС состоит в том, что традиционные ОС предполагают разработку драйверов на ассемблере или, в более современных системах, на чистом C. Даже в тех ОС, где допускается использование C++, сами точки входа функций драйвера должны использовать соглашения о вызовах C. В то же время понятно, что методика реализации драйверов имеет очевидные параллели с объектно-ориентированными технологиями. Драйвер представляет собой структуру данных (блок переменных состояния устройства) с которым связан набор процедур (точек входа драйвера). Это очень похоже на объект с методами в объектно-ориентированных языках программирования, поэтому идея реализовать "объектно-ориентированную ОС" приходит в голову многим, кто только что изучил ООП и операционные системы.

Darwin I/O Kit воплощает эту идею в жизнь. Он предполагает разработку драйверов на специальном диалекте C++ (так называемом Embedded C++, в котором не поддерживаются

шаблоны, исключения и множественное наследование, а вместо RTTI предоставляется нестандартный API для получения информации о типах объектов).

I/O Kit предоставляет богатый набор базовых классов, обеспечивающий инициализацию драйвера, его регистрацию в реестре доступных устройств, обмен данными с пользовательскими программами и стандартные "продвинутые" возможности современных ОС, такие как автоматическое распознавание устройств, работа с периферийными шинами PCI, ATA, USB, FireWire, SCSI и др, динамическая подгрузка и выгрузка драйверов, горячее подключение, управление питанием устройства (перевод в режим сниженного энергопотребления и вывод из него) и т. д. Над ядром Darwin надстроена полноценная Unix-система с командным интерпретатором Korn Shell, стандартным набором утилит командной строки и некоторыми полезными усовершенствованиями. Так, функции демонов init и cron традиционных Unix-систем (первый отвечает за инициализацию системы, второй обеспечивает запуск задач по расписанию) объединены в демоне launchd.

Основой графической подсистемы MacOS X является Quartz — графический API, базирующийся на стандарте OpenGL для трехмерных изображений и PDF (Portable Document Format) для двумерных изображений. С технологической точки зрения, Quartz 2D является наследником Display PostScript. PDF представляет собой подмножество языка PostScript, разработанное компанией Adobe для электронной публикации документов.

Распространена легенда, что отказ от PostScript и переход к PDF обусловлен слишком жесткими условиями, на которых Adobe предоставляет лицензии на использование PostScript. Впрочем, есть и другие аргументы в пользу такого перехода. Действительно, PostScript представляет собой полнофункциональный язык программирования. Но для таких задач, как отрисовка двумерных изображений на дисплее полная программируемость скорее вредна, чем бесполезна, например потому, что на полнофункциональном языке программирования можно написать бесконечный цикл или программу, иным способом потребляющую слишком много ресурсов. Таким образом, полнофункциональный интерпретатор PostScript представляет собой потенциальную точку для атаки отказа сервиса. Некоторые реализации PostScript, например GhostScript, предоставляющие операции для чтения и записи файлов, могут даже использоваться для внедрения вирусов и других троянских программ.

PDF представляет собой подмножество PostScript, эквивалентное базовому языку по графическим возможностям, но лишенное полной программируемости. Таким образом, поведение интерпретатора PDF гораздо более предсказуемо, как по требованиям к ресурсам, так и по безопасности.

Для совместимости с приложениями MacOS 9 система реализует две подсистемы — Carbon и Cocoa. Carbon представляет собой, грубо говоря, бинарный эмулятор MacOS 9, который позволяет исполнять бинарные модули MacOS 9 под OS X без каких-либо изменений. Cocoa представляет собой API, обеспечивающий легкое портирование приложений для традиционной MacOS в новую среду. Приложения Cocoa могут пользоваться как традиционными API, так и новыми возможностями, предоставляемыми Mac OS и Unix-подсистемой.

В отличие от Darwin, графические подсистемы OS X распространяются с закрытыми исходными текстами. Для защиты от неавторизованного копирования (а также, в определенной мере, и для защиты от внедрения троянского кода) ряд ключевых утилит последних версий OS X распространяются в зашифрованном виде. Для загрузки этих утилит подсистема виртуальной памяти ядра включает специальные криптографические модули.

В поставку MacOS также включена подсистема цифрового управления ограничениями (DRM) iTunes, совместимая с портативным цифровым медиаплеером iPod и реализациями iTunes/QuickTime для других платформ.

Первоначально OS X разрабатывалась для компьютеров PowerMac на основе процессоров IBM/Motorola Power. Впрочем, по некоторым сведениям, ОС сразу разрабатывалась как кроссплатформенная, с параллельной поддержкой Power и x86. В 2005 году Apple объявила о планах оставить платформу Power и перейти к изготовлению компьютеров на основе процессоров Intel CoreDuo с архитектурой x86. Главной причиной этого решения, безусловно, являлся тот факт, что процессоры Intel довольно сильно опередили процессоры Power как по абсолютной производительности, так и по отношению производительность/цена.

Новые компьютеры, известные как Intel Mac, обеспечивают исполнение бинарных загрузочных модулей для PowerMac в режиме бинарной эмуляции. Новые версии компиляторов C/C++ позволяют генерировать дуальные загрузочные модули, содержащие два сегмента кода для каждой из поддерживаемых аппаратных архитектур.

Intel Mac не являются PC-совместимыми. Вместо PC-совместимого BIOS они используют новый стандарт EFI (Extensible Firmware Interface). Из-за этого, загрузка ОС для PC-совместимых компьютеров на Intel Mac требует специального Boot-менеджера. В настоящее время доступна бета-версия загрузчика BootCamp, который обеспечивает установку и загрузку Windows XP.

Аналогично, OS X в стандартной поставке не может быть загружена на PC-совместимом компьютере. Подсистема Darwin легко может быть модифицирована (и реально была модифицирована) для такой загрузки, но Darwin — это далеко не вся система и даже не самая интересная ее часть. Версии OS X с модифицированным загрузчиком доступны в файлообменных сетях, но Apple не предоставляет технической поддержки пользователям этих версий, в том числе не предоставляет драйверов для работы ОС с периферийными устройствами, для которых нет аналогов в стандартных конфигурациях Intel Mac, и даже по мере сил пытается бороться с их распространителями как с "пиратами".

П1.9.1 Apple iOS

Кроме настольных компьютеров и ноутбуков, OS X также лежит в основе iOS, операционной системы Apple для портативных устройств (медиа-плееры iPod, телефоны iPhone, планшетные компьютеры iPad). Отличия включают в себя:

- Другую аппаратную платформу. Портативные устройства Apple основаны на RISC-процессорах с архитектурой ARM.
- Значительно отличающуюся структуру стандартной поставки. В комплект OS X входит более или менее соответствующий стандарту POSIX набор пользовательских утилит — командные процессоры, утилиты для работы с файлами и т. д. В комплекте iOS всего этого нет.
- В OS X владелец системы может установить и запустить практически любой код, в том числе написанный и откомпилированный самостоятельно. В iOS допускается только загрузка подписанных приватным ключом Apple приложений, скачиваемых по сети из онлайн-магазина Apple Store. Эти приложения исполняются в контейнерах с ограниченными привилегиями, которые так и называются «тюрьмой» (jail).
- Разработчикам приложений для iOS доступно лишь подмножество функциональности OS X и unix-подсистемы. Одно из главных ограничений состоит в запрете загрузки и динамической генерации исполняемого кода. Весь код, который приложение может исполнять, должен быть подписан Apple. В частности, это явно запрещает реализацию для iOS интерпретаторов, использующих JiT-компиляцию.

Для того, чтобы затруднить взлом ядра («бегство из тюрьмы», jailbreak) и снятие ограничений на работу приложений, Apple использует ряд мер, включающих в себя

аппаратную защиту от перепрошивки загрузочного монитора, зашифрованный код загрузочного монитора, проверку загрузочным монитором контрольной суммы и подписи загрузчика ОС, проверку загрузчиком ОС контрольной суммы и подписи кода ядра ОС, возможность отзыва подписи на приложении (приложение с отозванной подписью перестает работать на всех устройствах, получивших новую редакцию «черного списка» подписей) и т. д.

Кроме того, по соглашению Apple с разработчиками приложений, приложения не должны содержать не только средств загрузки внешнего кода или JiT-компиляторов, но и не могут содержать интерпретаторов полнофункциональных (полных по Тьюрингу) языков программирования, что исключает возможность реализации для iOS эмуляторов игровых консолей или Adobe Flash Player. Исключение сделано только для языка JavaScript, но и он может исполняться только в браузере Apple.

Большинство моделей устройств взламываются вскоре после поступления в открытую продажу. Так iPhone 5 был взломан в первую неделю после официального начала продаж. Однако взломанные телефоны и планшеты снимаются с гарантийной поддержки; кроме того, установка обновлений ОС на взломанные устройства часто приводит либо к отмене взлома (телефон «возвращается в тюрьму), либо, чаще, к различным отказам, вплоть до невозможности загрузить ОС. По большинству доступных оценок, количество эксплуатируемых взломанных устройств относительно невелико.

Подпись кода даёт некоторые преимущества пользователям и разработчикам ПО, но, конечно, главным получателем преимуществ является компания Apple. Среди преимуществ следует назвать:

- Для пользователей - защиту устройства от вирусов и другого вредоносного кода. Даже если какое-то из приложений Apple Store
- Для разработчиков ПО — техническую защиту от возможности загрузить неоплаченную версию программы, а также невозможность устранения различных средств «принуждения к оплате», таких, как реклама в бесплатной версии приложений.
- Также, разработчики, каким-либо образом обеспечившие себе хорошие отношения с компанией Apple, оказываются защищены от конкуренции: Aplle может не допускать в Apple Store аналогичные приложения.
- Для Apple — возможность осуществлять цензуру ПО, в частности, давая преимущества «хорошо себя ведущим» разработчикам.
- Главное из преимуществ для Apple — это возможность реализовать эффективное цифровое управление ограничениями не только для приложений, но и для файлов данных. Apple Store включает в себя не только возможность покупки приложений, но и возможность покупки звуковых, видео- и текстовых файлов — музыки, фильмов и книг. Оплаченные данные, например, аудиофайлы, привязаны к конкретному пользователю Apple Store и не могут быть проиграны на устройствах, принадлежащих другому пользователю. Разумеется, для того, чтобы принудить пользователей к исполнению этого ограничения, необходимо максимально затруднить как модификацию кода встроенной программы-проигрывателя аудиофайлов, так и возможность загрузки на устройство альтернативных программ-проигрывателей.

На 2012 год, устройства iOS представляют собой наиболее коммерчески успешный пример закрытой программно-аппаратной платформы, основанной на подписи кода и цифровом управлении ограничениями. Устройства на основе Android имеют более высокие продажи (как по количеству устройств, так и по деньгам), но пользователи iOS покупают больше дополнительного контента — это относится как к приложениям, так и к медиафайлам.