Система управления версиями GIT

Д.В. Иртегов

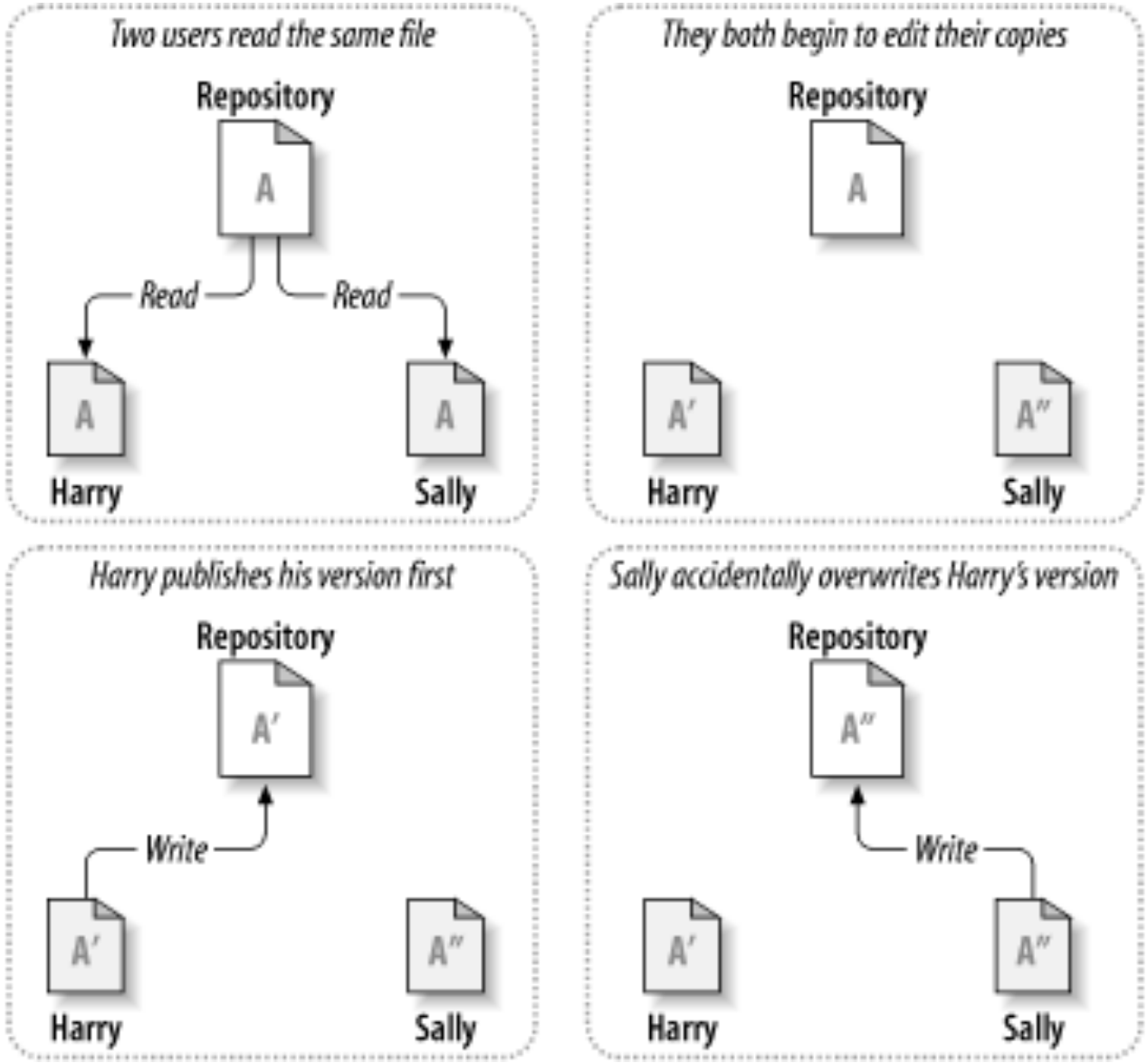
ФИТ/ФФ НГУ

2023

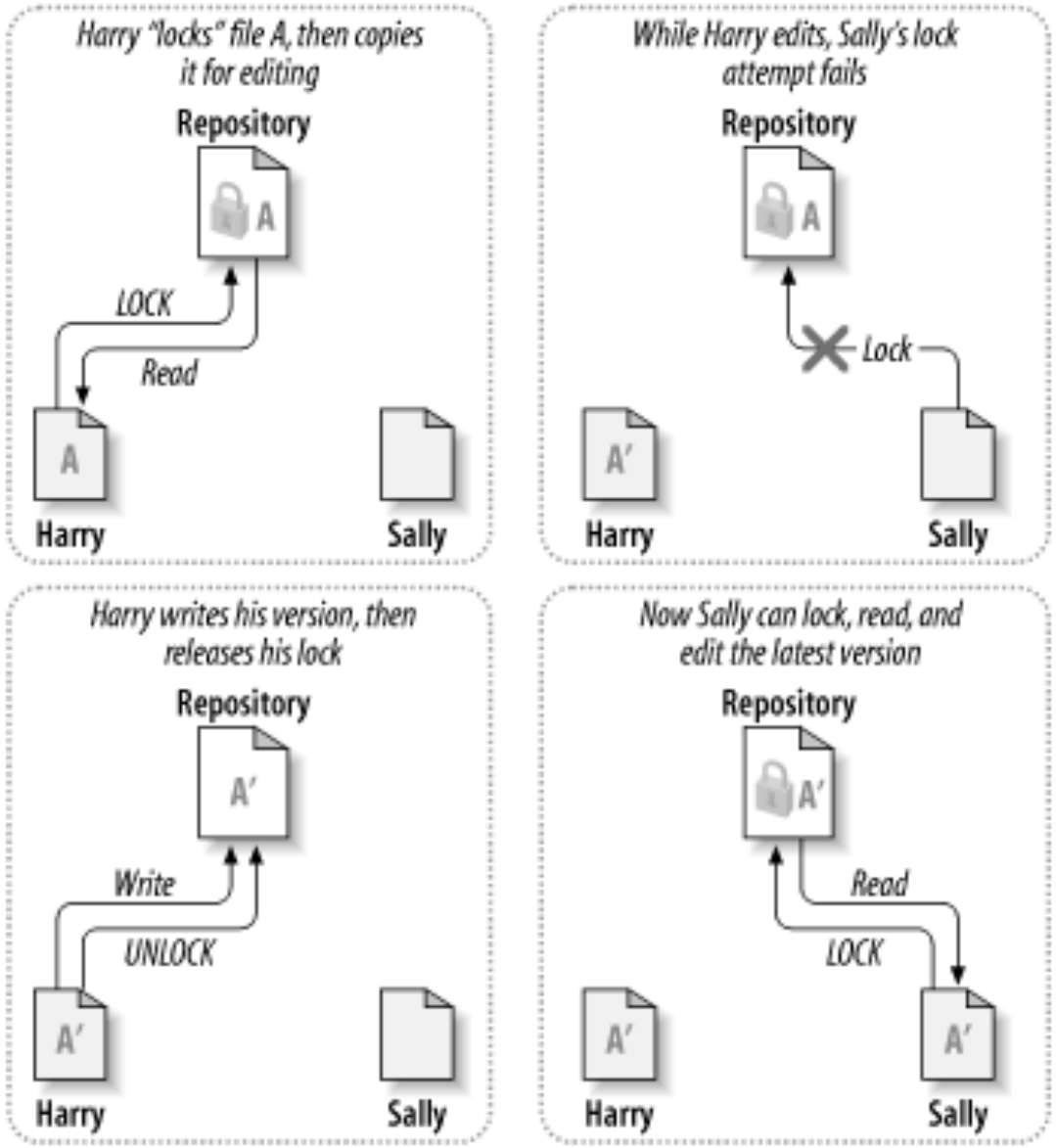
Что такое git

- Version Control System или Revision Control System
- Первые системы такого типа появились в 80е годы
- Главное назначение – управление исходным кодом программного проекта, над которым работает много разработчиков
- На сегодня, git - одна из самых популярных VCS
- Она не лишена недостатков, но для наших целей они не очень существенны

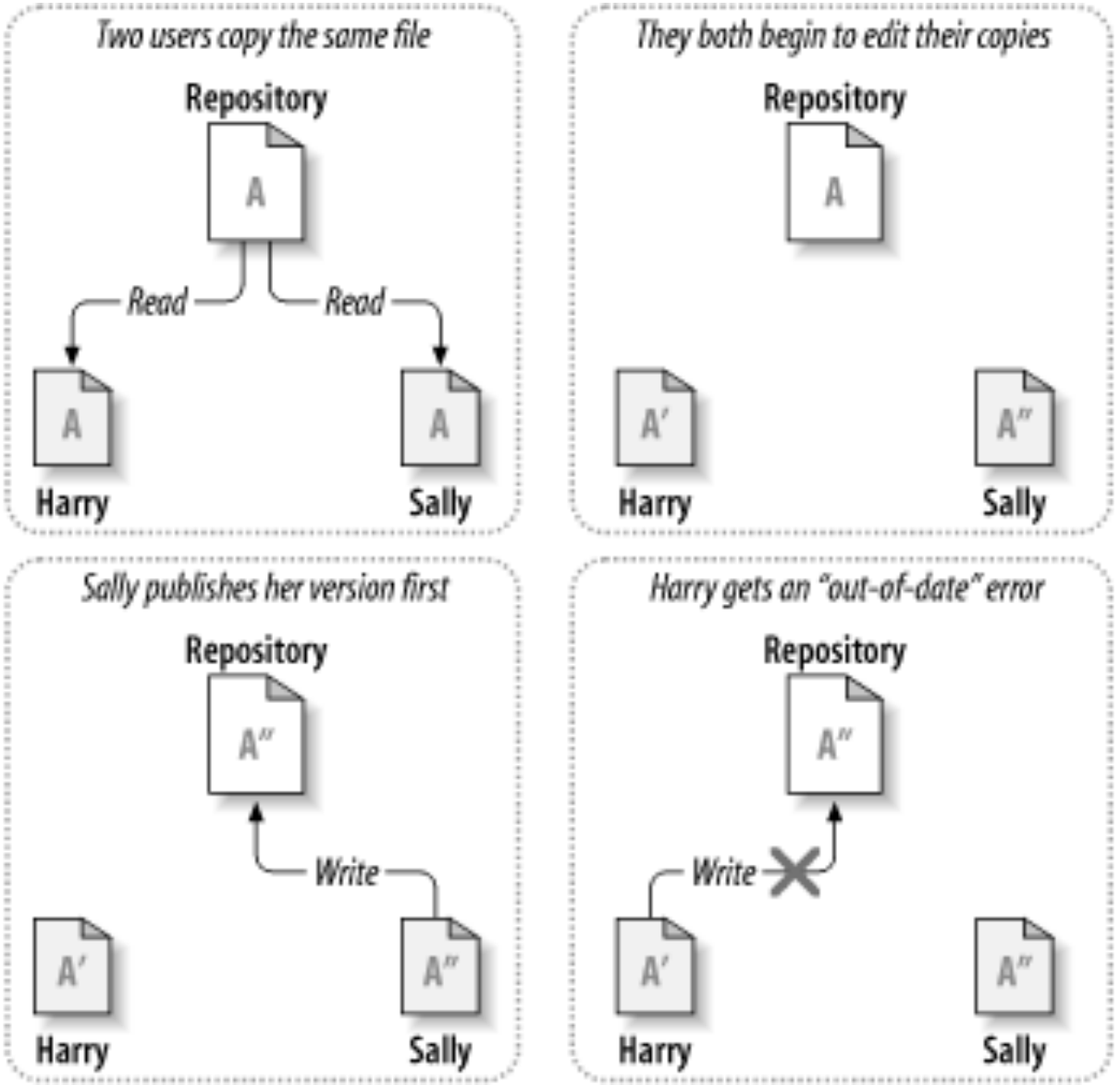
Главная проблема при работе команды



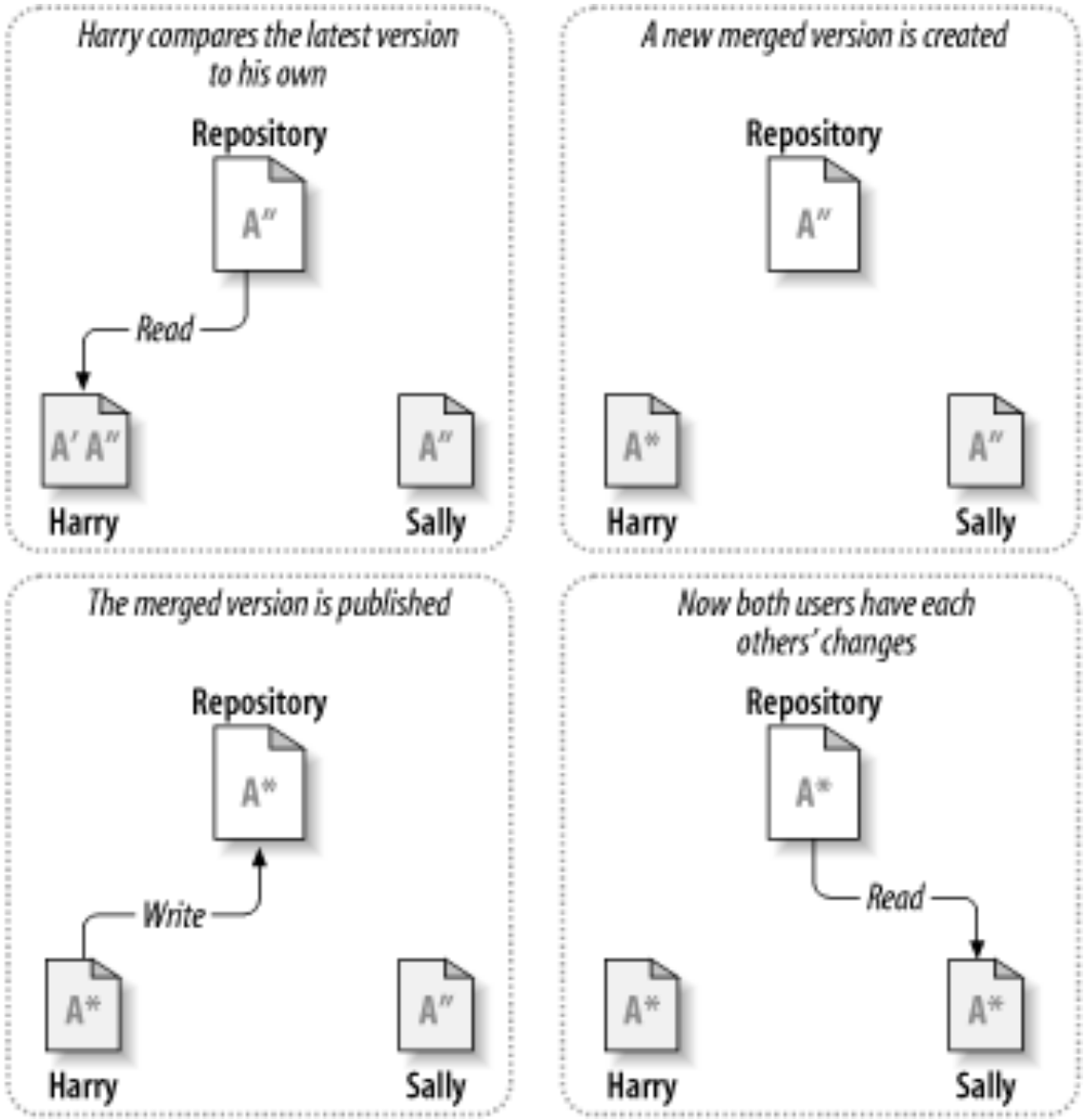
Блокирование-изменение



Копирование-слияние (копирование)



Копирование-слияние (слияние)



Основные понятия общие для всех* VCS

- * ну, для всех, какие я знаю
- Правка (commit)
  - Логическая группа из одного или нескольких изменений
  - Добавление файла, удаление файла, редактирование файла (в любых сочетаниях)
  - Также, содержит метаинформацию: кто, когда и зачем это коммитил, то есть автора, временной штамп и сообщение (описание)
- Репозиторий
  - Место (обычно, ресурс на центральном сервере), где хранятся коммиты
  - Репозиторий может приложить все или некоторые коммиты к проекту и показать состояние файлов проекта на некоторый (соответствующий выбранному коммиту) момент времени

Основные понятия VCS (продолжение)

- Рабочая (локальная) копия
  - Место на компьютере разработчика или машине CI/CD, где лежат файлы проекта в виде обычных файлов. Может соответствовать состоянию проекта на момент любого коммита.
  - Рабочая копия также содержит метаданные (с каким репозиторием она связана, какому коммиту соответствует, даты модификации или хэши файлов и др.)
  - Рабочая копия может быть синхронизована с репозиторием (точно совпадать с каким-то коммитом) или не синхронизована (содержать незакоммиченные изменения)
  - Рабочая копия может содержать файлы, которых нет и даже не должно быть в репозитории (конфигурацию среды разработки, генерируемые при сборке файлы, тестовые данные, заметки разработчика для себя)

Основные понятия VCS (продолжение)

- Ветка (branch)
  - Группа коммитов, организованных в линейную историю
    - Поскольку ветки могут сливаться, история ветки не всегда тождественна хронологии внесения изменений
  - Ветки могут использоваться для:
    - Хранения разрабатываемой версии продукта + поддерживаемые версии
    - “Feature branch” - разработчик отводит ветку для работы над какой-то фичей, чтобы не коммитить незаконченный (возможно, вообще неработоспособный) код в релизную ветку. Когда работа заканчивается, ветку сливают с основной (например, релизной)
    - Других целей
- Голова (HEAD)
  - Последний коммит в ветке
- Тег (tag)
  - Вообще говоря, произвольный коммит в произвольной ветке, которому вы зачем-то захотели дать человеко-читаемое имя
    - Например, минорный релиз продукта или состояние релиза (например, code freeze)
    - Или состояние ветки, которое вы хотите передать на ревью
    - От тега потом можно отвести ветку

Основные понятия VCS (продолжение)

- Слияние (merge)
  - Перенос всех или некоторых коммитов из одной ветки в другую
  - Во многих VCS синхронизация рабочей копии с центральным сервером реализована как частный случай слияния
- Конфликт
  - Одна и та же строка или соседние строки одного файла изменены в разных коммитах
  - В линейной истории более поздний коммит считается сильнее
  - В нелинейной истории (слияние веток или одна ветка измененная в разных рабочих копиях) это правило не работает
  - VCS имеют эвристики, позволяющие автоматически сливать конфликты, но они не всегда работают
- ***ВНИМАНИЕ!***
- Даже если VCS все автоматически слила, это не означает, что у вас получился правильный (или даже синтаксически допустимый) код
- Это верно для всех существующих VCS (не только для тех, с которыми я работал)
- Пару раз нарветесь на это, наверное поймете почему это так

Теперь собственно про git

- «Распределенная» VCS
- Рабочая копия содержит полную реплику репозитория в папке .git
- Коммит и синхронизация с центральным репозиторием – это две разные операции (commit и push соответственно)
- Вы можете сделать много коммитов, не синхронизуясь с сервером
  - Можно какое-то время работать без связи с сервером
  - Переключение с ветки на ветку и обращение к другим веткам происходит очень быстро (все данные локальные)
  - Можно ставить опыты, в том числе делать слияния с защищенными ветками, не боясь повредить данные на сервере
- Неудобно работать с большими репозиториями
  - На ваш компьютер скачивается большая база
  - Нельзя запретить разработчику смотреть некоторые ветки или папки
  - В каждый момент, локальная копия содержит только одну ветку
  - Но в базе они лежат все,
    поэтому в частных репозиториях админ периодически бегает и кричит «сотрите ненужные ветки»
- Вы не можете синхронизовать локальные копии между собой, это нужно делать через центральный сервер
  - Поэтому слово «распределенная» в первом пункте взято в кавычки

Центральные серверы Git

- Github.com – публичный бесплатный сервер для размещения ваших репозиториев
  - В настоящее время принадлежит компании Microsoft
  - Имеет ряд дополнительных фич, например пулл реквесты
  - Имеет платные фичи
- Gitlab – сервер с открытыми исходниками, который вы можете развернуть у себя
  - Довольно тяжелый и по ресурсам, и по трудоемкости поддержки
  - Аналоги многих платных фич гитхаба доступны бесплатно (ну, по стоимости труда сисадмина/девопса по их правильной настройке)
- Да тыща их

Основные операции git

- git clone *URL*
  - Создать локальную копию на основе удаленного репозитория
  - HTTPS URL: https://github.com/dmitry-irtegov/NSU-OS-2023.git
    - Требует имя/пароль, можно их запомнить в локальном конфиге (небезопасно)
  - SSH URL: git@github.com:dmitry-irtegov/NSU-OS-2023.git
    - Требует загрузить на сервер ваш публичный ключ ssh
    - На Solaris и машинах в терминальных классах, лучше, чтобы парный приватный ключ ssh был с паролем
    - Приватный ключ ssh вы также можете использовать для подписи коммитов
    - (GPG настраивать не обязательно)
- git status
  - Посмотреть состояние локальной копии
  - На какой вы ветке/коммите, какие локальные файлы изменены, есть ли конфликты и т.д.

Основные операции git

- git add *files*
  1. Поставить файл под контроль гита (будет коммититься и синхронизоваться с репозиторием)
  2. Добавить измененный файл к списку файлов, которые будут включены в следующий коммит
  3. Подтвердить, что вы завершили ручное слияние конфликта
- git commit [--amend] [-m *message*] [*files*]
  - --amend – редактировать предыдущий коммит (изменить сообщение, изменить автора, подписать)
  - Если не указать files, коммитит только то, чему вы сделали add
- git checkout [-b] *branch*
  - Переключиться на существующую ветку
  - -b – создать новую ветку на основе текущей

Основные операции git

- git log [*branch*]
  - Посмотреть историю коммитов
- git blame *file*
  - Посмотреть, кто ~~кинул сапогом в пульт~~ менял конкретные строчки в файле
- git diff *branch/commit/*. *branch/commit/*.
  - Посмотреть разницу между двумя коммитами или коммитом и текущим каталогом
  - Выдает файл в формате patch, который можно приложить командами patch или git apply
  - Среды разработки часто имеют аналогичную операцию с графическим интерфейсом

Основные операции git

- git merge [--squash] *branch/tag*
  - Переносит все коммиты, которых нет в текущей ветке, из выбранной
  - Создает дополнительный merge commit
  - С ключом --squash схлопывает их все в один коммит и не создает merge commit
  - Если вам дороги жизнь и рассудок, не применяйте эту операцию к неродственным или далеко разошедшимся веткам
  - Может обнаружить конфликты
  - Останавливается и требует ручного слияния при каждом отдельном коммите, вызывающем конфликт
- git cherry-pick *commit*
  - Переносит отдельный коммит из произвольной ветки в текущую.

Основные операции git

- git pull
  - Выкачать изменения с сервера и попытаться применить их к локальной копии
  - Может привести к обнаружению конфликтов
- git fetch
  - Выкачать изменения с сервера но НЕ применять их к локальной копии
- git push
  - Закачать закоммиченые изменения из локальной копии на сервер
  - Может привести к обнаружению конфликтов
  - Останавливается при конфликтах
  - Рекомендуется сначала сделать pull, разрешить конфликты (если они есть), только потом пушить

Не столь основные операции git

- git rebase
  - Реализует широкий набор операций по переписыванию истории
  - Обычно старается создать иллюзию что оно всегда так было
  - Позволяет сливать (синхронизовать) ветки без создания merge commit
    - Конфликты при этом все равно надо править руками
  - Позволяет удалять нежелательные коммиты, в том числе из глубины истории (например, вы по ошибке закоммитили фотку себя в голом виде)
  - Позволяет схлопывать коммиты, в том числе идущие не по порядку, в один (merge --squash умеет это делать только с линейными последовательностями)
  - Может использоваться для разного рода вандализма,
    в том числе непреднамеренно
  - Если вам дороги жизнь и работа, после ребейза, перед тем как это пушить, ***всегда внимательно смотрите что у вас получилось***,
    как в истории, так и в текущей копии
  - Обычно требует push -f

Наиболее популярные формы rebase

- git rebase *branch*
  - «Читерское» слияние без создания merge commit
  - Может использоваться для «перепрививки» ветки на другую родительскую ветку
  - Рекомендуется делать перед мержем, чтобы заранее выявить конфликты
- git rebase -i *commit*
  - Формирует и открывает на редактирование текстовый файл со списком всех коммитов в текущей ветке после указанного
  - В комментариях есть довольно развернутая справка по командам
  - Редактируя этот файл, вы можете сказать что делать с каждым из этих коммитов: сохранить, сохранить и схлопнуть с предыдущим, сохранить и изменить сообщение, удалить

Технические детали

- Метаинформацию коммита, в первую очередь автора, нужно указывать локально
- git config --global user.name “Vasya Pupkin”
  git config --global user.email v.pupkin@g.nsu.ru
- Эта информация не обязана совпадать с настройками вашей учетной записи в репозитории, но в коммите будет указана именно она, и в таком виде и запушится
- Если вы пушите не в свой репозиторий, админ репозитория вам после этого может открутить руки
- Если вы ведете разные проекты с разными учетными записями, вы можете захотеть вместо --global написать --local

Пулл реквесты

- Фича гитхаба
- Позволяет пушить изменения в репозитории, в которые вы вообще-то пушить не можете
- Точнее, запрашивать возможность запушить (решение о пуше принимает владелец репозитория)
- Создаются по относительно сложной процедуре

Процедура создания пулл реквеста (первый реквест)

- В веб-интерфейсе гитхаба, заходите в целевой репозиторий (в случае сдачи задач по ОС, в мой репозиторий NSU-OS-2023)
- Там нажимаете кнопку fork
- Если вы зарегистрированы на гитхабе, в вашей учетной записи создастся копия (fork) моего репозитория
- Не забудьте ***выключить*** галочку «fork only main branch», вам же нужно будет создавать пулл реквесты в ветку вашего семинариста
- Коммитите ваши изменения в ваш репозиторий, лучше сразу в ветку, созданную от prepod/accepted
- Когда все отладите и протестируете, схлопываете все в один коммит, подписываете и пушите в ваш форк
- В веб-интерфейсе гитхаба, открываете вашу ветку и нажимаете кнопку “create pull request”. По умолчанию она предложит именно тот репозиторий, от которого вы форкались
- Если преподаватель запросит изменения, вы можете их просто пушить или даже форс пушить в ту ветку, с которой создавали запрос, они отобразятся в пулл реквесте автоматически

Процедура создания пулл-реквеста (второй и следующие запросы)

- Принятое в моем курсе соглашение по именованию файлов в ветках задач позволяет избежать большинства неприятностей.
- Тем не менее, делать пулл реквест из репозитория, который сильно отстал от родительского – операция не для слабонервных
- Перед созданием следующих пулл реквестов, синхронизуйте ваш форк (репозиторий) с моим. Само по себе это не происходит.
- Если вы уже что-то коммитили в ветку с новым решением, необходимо перебазировать эту ветку на текущее состояние prepod/accepted (ваши изменения при этом сохранятся). Это можно сделать через веб-интерфейс гитхаба или из локальной копии командами
- git fetch upstream
  git checkout your-branch
  git rebase upstream/prepod/accepted
  # убедитесь, что ваше решение не повредилось
  git push -f

Уточненные правила создания пулл реквестов

- В начальный момент пулл реквест должен содержать один коммит
  - Если у вас помимо вашей воли добавляются другие коммиты, например от других задач, синхронизуйте ветку по процедуре, описанной на предыдущем слайде, или создайте новую ветку от синхронизованного состояния prepod/accepted
- В коммите не должно быть посторонних файлов, включая бинарные и объектные файлы, только исходные файлы сдаваемой задачи и makefile, если он есть
- Автор коммита должен иметь человеческое имя и ваш университетский email. Эти настройки не обязаны совпадать с настройками вашей учетной записи github.
- Коммит должен быть подписан, не обязательно ключом GPG, можно просто использовать ваш ключ ssh