Praktikum angewandte Systemsoftwaretechnik (PASST)

Versionskontrollsysteme / Aufgabe 3

04. November 2019

Tobias Langer, Michael Eischer
und Florian Schmaus

Lehrstuhl für Informatik 4
Friedrich-Alexander-Universität Erlangen-Nürnberg

Lehrstuhl für Verteilte Systeme
und Betriebssysteme

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT

Rückblick

Rückblick

- Verbindung mit virtueller Maschine per SSH
  - → Verwendung von öffentlichen/privatem Schlüsselpaar
  - → Datentransfer per `scp` und `sshfs`
- `diff` zum ermitteln von Änderungen in Textdateien
  - → `unified diff` als „standardisiertes“ Format

Motivation

Motivation

- Entwicklungsprozesse sind selten linear
  - Experimentelle Features werden getestet
  - Änderungen führen Fehler ein
  - Änderungen müssen zusammengeführt werden

- Versionierung unterstützt Entwicklungsprozesse
  - Dokumentation von Änderungen
  - Dokumentation von Versionsständen
  - Verfahren zum Zusammenführen von Versionsständen

$\Rightarrow$ Manuelle Versionierung ist mühsam & fehleranfällig

Versionierung

Typische Aufgaben eines Versionskontrollsystems sind:

- Sichern alter Zustände
- Zusammenführen paralleler Entwicklungszweige
- Dokumentieren von Änderungen
- Transportieren von Daten

Idealerweise:

- Keine zentrale Infrastruktur notwendig

Lernziele

Im Anschluss an diese Aufgabe solltet Ihr...

- die grundlegende Funktionsweise des Versionskontrollsystems Git beschreiben
- Änderungen versionieren und mit Anderen teilen
- Änderungen Anderer in Euren Versionsstand einspielen
- Fehler im Linux-Kernel finden & beheben

...können.

Agenda

Fakten & Infos rund um Git

Funktionsweise von Git

Arbeiten mit Git

Arbeiten & Navigieren im Linux-Kernel

Zusammenfassung

Aufgabe 3

Fakten & Infos rund um Git

Das Versionskontrollsystem des Linux-Kernels

Git

- Entwickelung 2005 von Linus Torvalds initiiert
- Ausgelegt zur Unterstützung der Linux Kernel Entwicklung

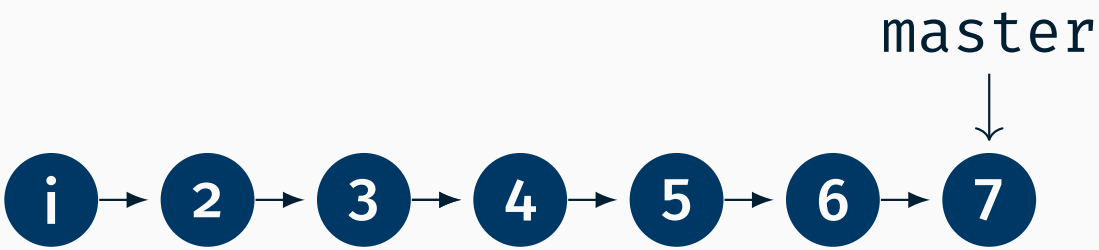
Zentrale Eigenschaften:

- Unterstützung für dezentrale & parallele Entwicklung
- Visualisierung von Entwicklungszweigen
- Ausgelegt für Umgang mit großen Patchmengen

Funktionsweise von Git

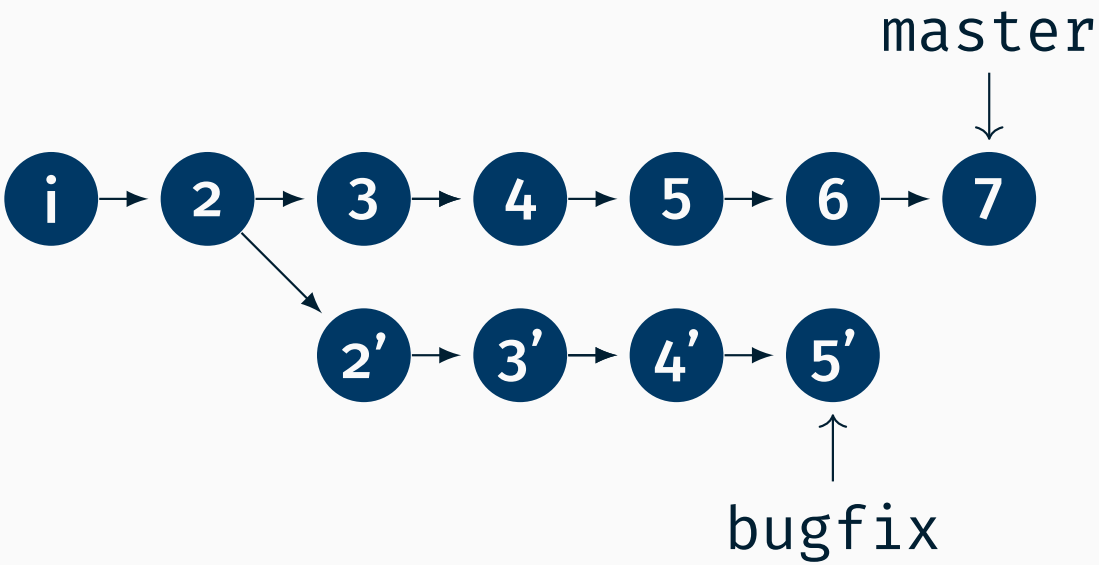
Wie funktioniert Git?

- Sicherung vollständiger Daten jedes Versionsstandes („commit“)
- Eindeutige SHA1-Hashes für jede Version
- Jede Version kennt ihren Vorgänger („parent“)
- Jede Versionsserie („branch“) bekommt einen Namen



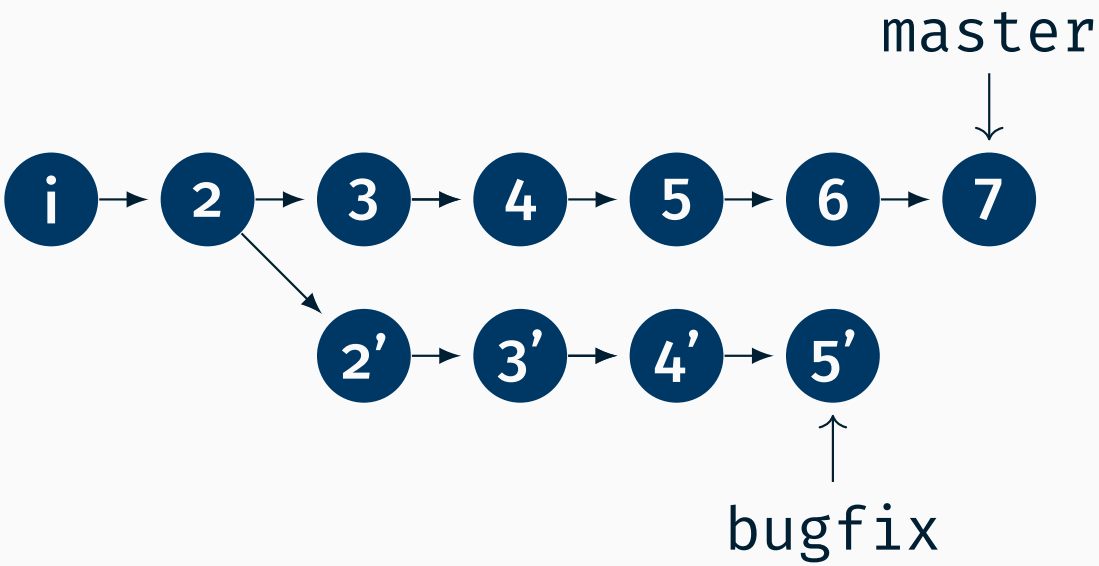
Wie funktioniert Git?

- Sicherung vollständiger Daten jedes Versionsstandes („commit“)
- Eindeutige SHA1-Hashes für jede Version
- Jede Version kennt ihren Vorgänger („parent“)
- Jede Versionsserie („branch“) bekommt einen Namen



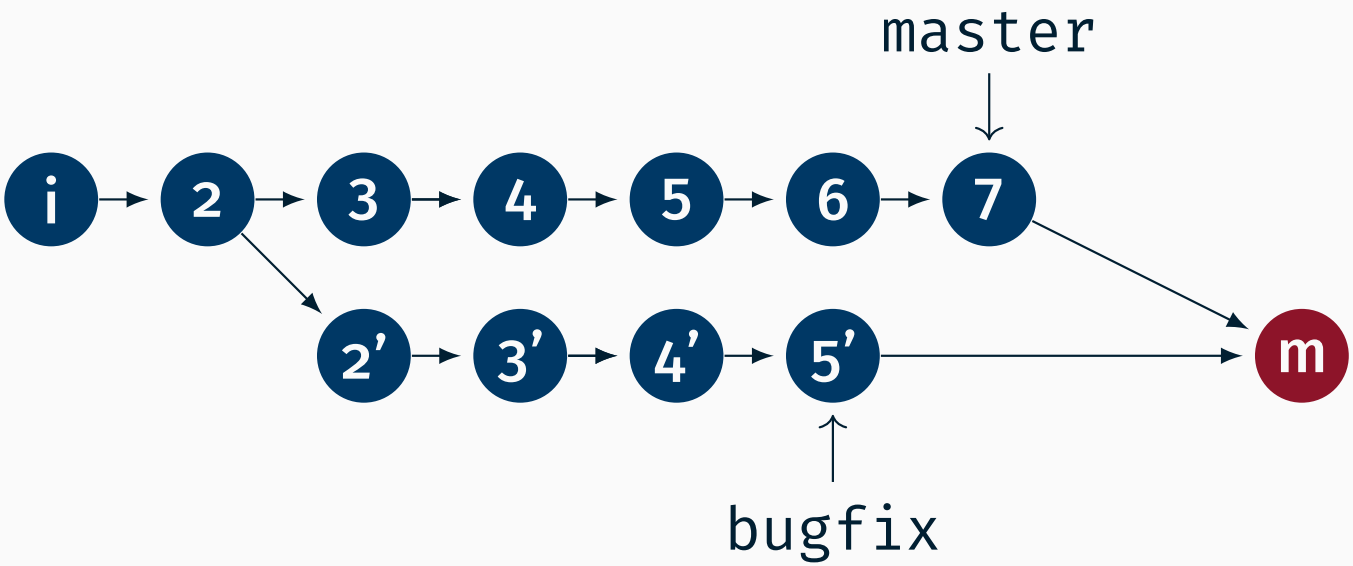
Verzweigungen und Zusammenführungen

Beispiel für parallele Entwicklung (branches und merge):



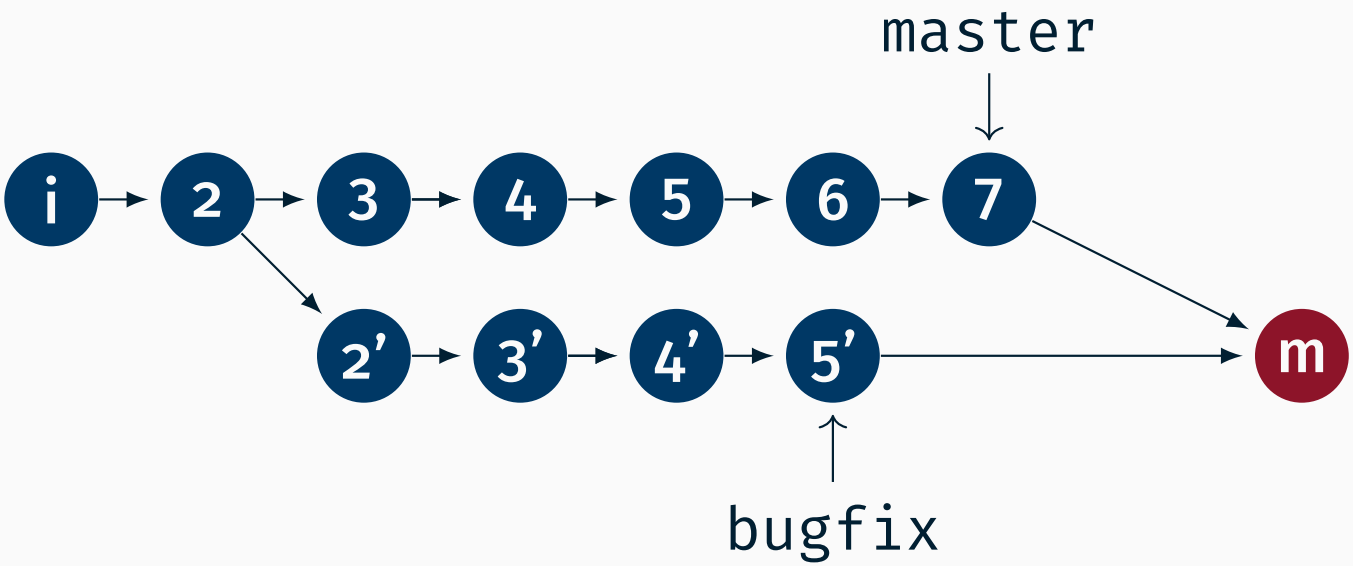
Verzweigungen und Zusammenführungen

Beispiel für parallele Entwicklung (branches und merge):



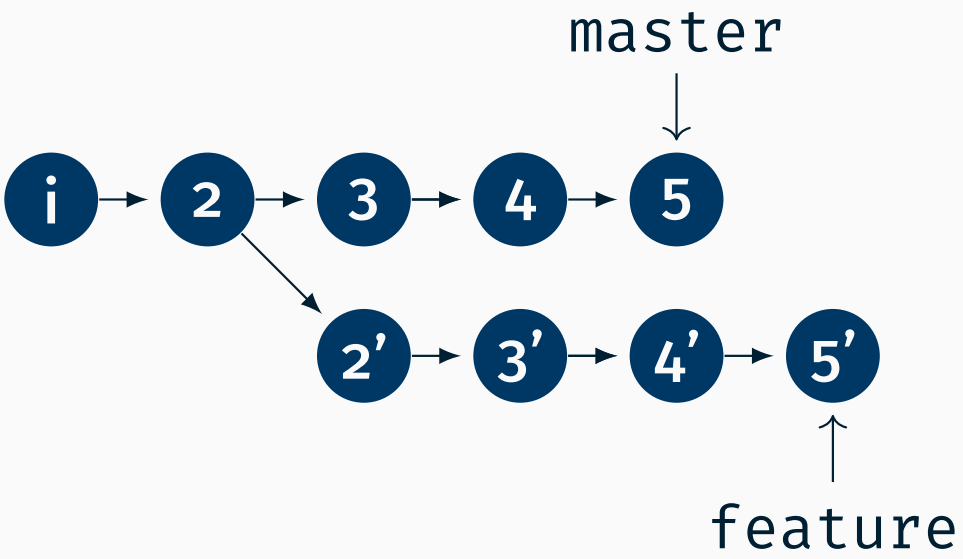
Verzweigungen und Zusammenführungen

Beispiel für parallele Entwicklung (branches und merge):

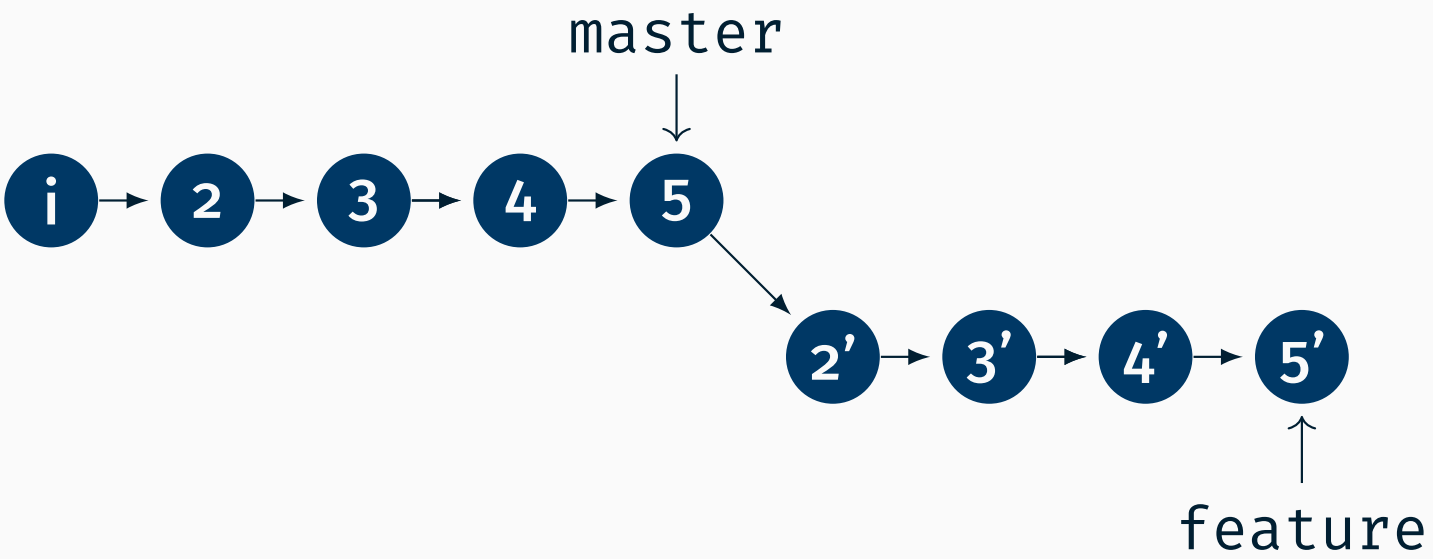


- Git versucht Änderungen automatisch zusammenzuführen
- Nur wenn Änderungen nicht eindeutig:
  - Merge-Konflikt
  - Manuelles eingreifen („resolve“) notwendig

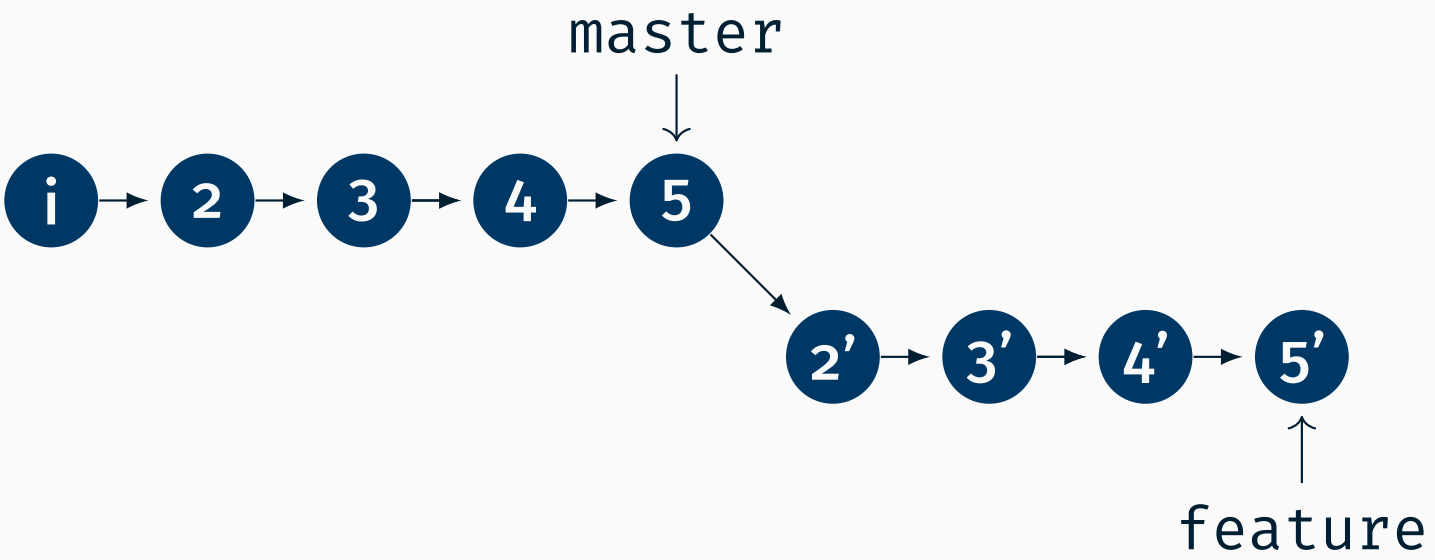
Aufsetzen auf bestehenden Zweigen (rebase)



Aufsetzen auf bestehenden Zweigen (rebase)



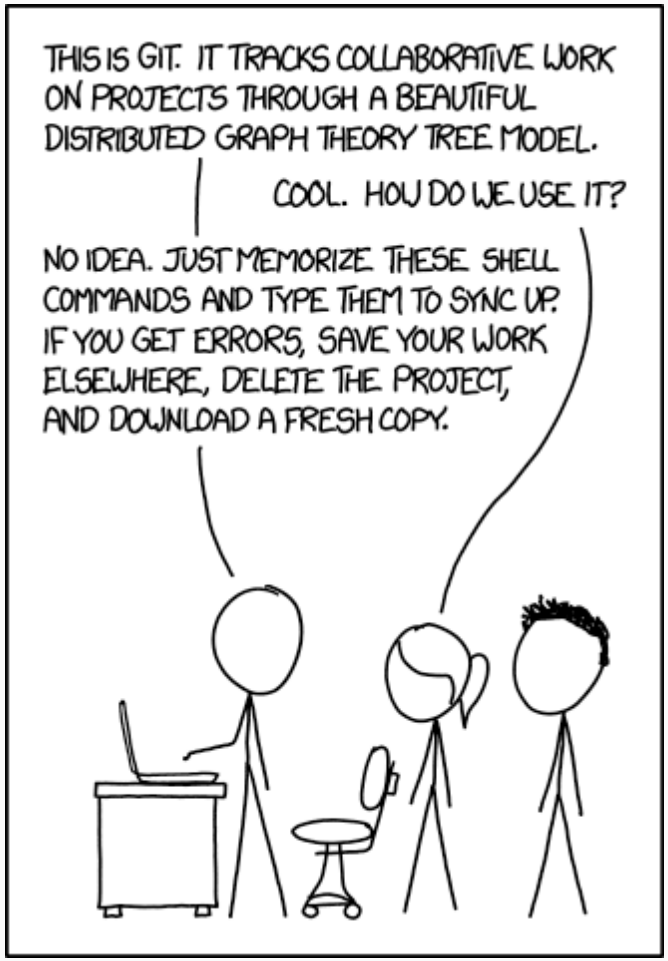
Aufsetzen auf bestehenden Zweigen (rebase)



- „Umbiegen“ des Wurzelknotens eines Branches
- Linearisierung der Historie
- Vorsicht!
  - Verzweigungen werden nicht automatisch angepasst!
  - Nach einem Rebase haben alle Patches neue Hashes

Arbeiten mit Git

Arbeiten mit Git



https://imgs.xkcd.com/comics/git.png

Repositories einrichten und Änderungen einspielen

- Initialisieren einen Repos im aktuellen Verzeichnis

```
git init
```

- Initiales Klonen der Quellen

```
git clone \
git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git
```

- Vormerken einer (versionierten) Datei als Kandiat für nächsten Commit („staging“)

```
git add Datei
```

- Einspielen von eigenen Änderungen in Datei oder aller Änderungen

```
git commit Datei
git commit -a
```

Git interactive rebase (`rebase -i`) (1/2)

Erlaubt die Versionsgeschichte „neu“ zu schreiben, also Commits

- umsortieren
- verschmelzen
- aufteilen
- nachträglich ändern
- …

In der Regel wirken sich die Operationen auf den Hash aus!

Git interactive rebase (rebase -i) (2/2)

```
~ git rebase --interactive HEAD~4

[vim]
pick 0b9bf3812ad1 afs: Split wait from afs_make_call()
pick a690f60a2ba3 afs: Calculate lock extend timer from...
pick 68ce801ffd82 afs: Fix AFS file locking to allow fine...

# Rebase 149e703cb8bf..a9fbcd672883 onto 149e703cb8bf (4 commands)
#
# Commands:
# p, pick <commit> = use commit
# r, reword <commit> = use commit, but edit the commit message
# e, edit <commit> = use commit, but stop for amending
# s, squash <commit> = use commit, but meld into previous commit
...
```

Zustand von Repositories untersuchen

- Anzeige der Differenzen zum Vorgänger (bzw. Anzeige des vorbereiteten [„staged“] Commits)

```
git diff
git diff [--staged|--cached]
```

- Dateizustände (neu, unbekannt, geändert, vorgemerkt) anzeigen

```
git status
```

- Die neuste Änderung untersuchen

```
git show
```

Arbeiten mit entfernten Repositories

- Einspielen von entfernten Änderungen

```
git pull
```

- Weitere entfernte Repositories registrieren

```
git remote add next git://git.kernel.org/.../next/linux-next.git
```

- Registrierte Repositories auflisten

```
git remote -v
```

- Alle Remotes nachladen (aktueller Branch wird nicht verändert)

```
git remote update
# oder
git fetch --all
```

Erstellen und Anzeigen von Branches

- Lokalen Branch aus dem neuem „Remote“ anlegen

  ```
  git checkout -b work next/master
  ```

- Alle registrierten Zweige anzeigen

  ```
  git branch -a
  ```

- Unterschiede zwischen lokalem und entferntem Branch

  ```
  git log ..origin/master
  git log -p ..origin/master
  ```

- Aktuelle Änderungen auf dem entfernten Branch neu aufspielen

  ```
  git pull --rebase
  ```

Navigieren in der Commit Historie

- Commit Historie ist selten linear
  - Merge Commits
  - Ein Commit kann mehr als einen Elterncommit haben
  - Historie ist ein gerichteter azyklischer Graph
- Spezielle Notation zum Zugriff auf Vorgängercommits

```
# ~n zeigt auf nten Vorgängercommit
git show HEAD~1
# Ersten "Großelterncommit", alternativ HEAD~~
git show HEAD~2
# ^n wählt Vorgängercommit ntem Elternbranch
git show HEAD^1
# Vorgängercommit aus zweitem Elternbranch, alternativ HEAD^^
git show HEAD^2
# Kombination von alledem
git show HEAD^4~~
```

Austauschen von Änderungen als Patches

- Die letzten **2** Änderungen als Patch formatieren

```
# bei einem Merge den ersten Vorgänger wählen
git format-patch HEAD~2
```

- Sendeziel für Patchversand via E-Mail vorgeben

```
git config sendemail.to=linux-kernel@i4.cs.fau.de
```

- Patchset mit den letzten **3** Änderungen via E-Mail senden

```
git send-email --compose HEAD~3
```

Sonstige Werkzeuge & Infrastruktur (1/2)

- Vielzahl an freier & kommerzieller, graphischer Werkzeuge
  - gitg
  - gitk
  - tig
  - git gui
  - ...
- Für eine Aufstellung, siehe auch:
  - `https://git.wiki.kernel.org/index.php/Interfaces,_frontends,_and_tools`

Sonstige Werkzeuge & Infrastruktur (2/2)

- Webgestützte Systeme
  - GitHub
  - GitLab (bspw. `https://gitlab.cs.fau.de`)
  - …
- Erweitern Funktionalität teilweise erheblich
  - Projektverwaltung
  - Nutzerverwaltung
  - Continuous Integration, Delivery, Deployment, …

Arbeiten & Navigieren im Linux-Kernel

Debuggen des Linux-Kernels (1/2)

- `(k)gdb` nicht für alle Fehlertypen der beste Ansatz
  - Was tun, wenn der Kernel sehr früh Oopst?
  - Wie vorgehen, die serielle Schnittstelle fehlerhaft ist?
  - Wenn alles „wegoptimiert“ ist?
- Lösung: `printk()`-Ausgaben auf der Konsole und im Kernel-Log

Debuggen des Linux-Kernels (2/2)

Prototyp:

```
int printk(const char *s, ...)
```

Beispiel aus `linux-5.1/init/main.c`:

```
pr_notice("Kernel command line: %s\n", boot_command_line);
```

mit

```
#define pr_notice(fmt, ...) \
printk(KERN_NOTICE pr_fmt(fmt), ##__VA_ARGS__)
```

Meldungen *nachlesen*:

```
passt [~]> dmesg
...
[0.000000] Kernel command line: root=/dev/vda1 console=ttys0
...
```

Debuggen mit `printk()`

- alle Ausgaben haben eine Priorität (<n> am Stringanfang)
- Ausgabe erst bei Kernel Log-Level $n + 1$
- Log-Level wird standardmäßig auf 7 gesetzt
- Anpassen über Kommandozeile (debug, `loglevel`) und `klogd(8)`

Mögliche Prioritäten (`include/linux/printk.h`)

```
#define KERN_EMERG   "<0>" /* system is unusable                */
#define KERN_ALERT   "<1>" /* action must be taken immediately */
#define KERN_CRIT    "<2>" /* critical conditions              */
#define KERN_ERR     "<3>" /* error conditions                 */
#define KERN_WARNING "<4>" /* warning conditions               */
#define KERN_NOTICE  "<5>" /* normal but significant condition */
#define KERN_INFO    "<6>" /* informational                    */
#define KERN_DEBUG   "<7>" /* debug-level messages             */
```

Code-Navigation

- cscope
  - Symbolsuche, insbesondere für Funktionen
  - Aufrufer und Aufgerufene

```
make cscope # cscope Index erzeugen
cscope -d   # Index für modifizierte Dateien nicht neu erzeugen
...         # Verlassen mit CTRL-d
```

- ctags
  - Ebenfalls Symbolsuche
  - Autocompletion

Zusammenfassung

Zusammenfassung

- Versionsverwaltung mit Git
  - Sichert Änderungshistorie eines Repositories als Commits
  - Arbeit an verschiedenen Entwicklungspfaden in Branches
  - Dezentral, d.h. Unterstützung von Remote Repositories
- Debugging des Linux-Kernels
  - `printk` wenn der `(k)gdb` versagt
  - Es existieren viele Werkzeuge zur Navigation in Sourcecode

Aufgabe 3

Fehler finden und beheben

- Vorgegebene Kernelquellen mit injizierten Fehlern:
  `/proj/i4passt/kernel/linux-borked.git`
- Vorgegebene Kernelconfig:
  `/proj/i4passt/kernel/linux-borked.config`
- Verschiedene Fehlertypen, *nicht nur* Systemabstürze
  - „Normale“ Systembenutzung um alle Fehler zu finden
  - kgdb nicht immer das optimale Werkzeug
- Insgesamt zwölf verschiedene Fehler
- Patches an `linux-kernel@i4.cs.fau.de`
  - Achtung, Hannover liest mit!

Bearbeitungszeit bis 18. November

Fragen?