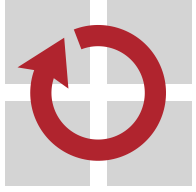
Praktikum angewandte Systemsoftwaretechnik (PASST)

bisect / upstream / Aufgabe 4

6. Juni 2019

Tobias Langer, Stefan Reif, 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

- Versionsverwaltung mit Git
  - Grundlegende Versionsverwaltungskonzepte mit Git
  - Arbeiten am eigenen Repository
  - Kollaboratives Arbeiten mit anderen
- Werkzeuge zur Arbeit am Linuxkernel
  - Navigation im Linuxkernel
  - Ausgabe von (Fehler-)meldungen

Motivation

Motivation

- FLOSS Software erlaubt Änderungen am Sourcecode
  - eigenständige Behebung von Fehlern
  - implementieren eigener Featurewünsche
  - ... idealerweise teilen der Änderungen
- Kollaboration erfordert einheitliches Entwicklungsmodell
  - Diskussion über Änderungen & langfristige Richtungsentscheidungen
  - Erhalten von Qualitätsstandards
  - Umgang mit Lizenzfragen

Lernziele

Lernziele

Im Anschluss an diese Aufgabe solltet Ihr...

- die Funktionsweise des Bisektionsalgorithmus zur Fehlersuche erklären
- Fehler mithilfe von `git bisect` finden und beheben
- die Werkzeuge zur Entwicklung des Linuxkernels beschreiben
- den Entwicklungsprozess des Linuxkernels beschreiben
- eigene Patches für den Linuxkernel in den Entwicklungsprozess einbringen und akzeptiert bekommen

...können.

Agenda

Rückblick

Motivation

Lernziele

Fehlersuche mittels Bisektion

Linux Entwicklung

Zusammenfassung

Aufgabe 4

Fehlersuche mittels Bisektion

git bisect

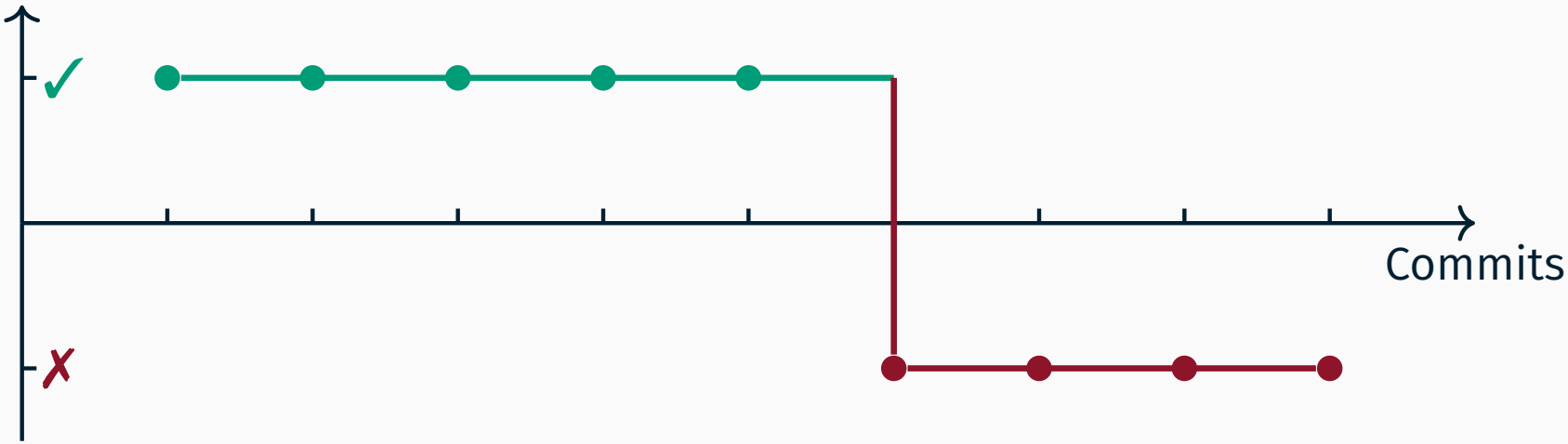
- Werkzeug zur Fehlersuche in einem Git-Repository
- „Welche Revision hat den Fehler eingeführt?“
- Voraussetzungen:
  - Eine Revision in der der Fehler auftritt
  - Eine Revision in der der Fehler (noch nicht) auftritt
  - Manueller oder automatischer Test für den Fehler
  - Möglichst viele testbare (d.h. übersetzbare) Revisionen

Bisektionsalgorithmus

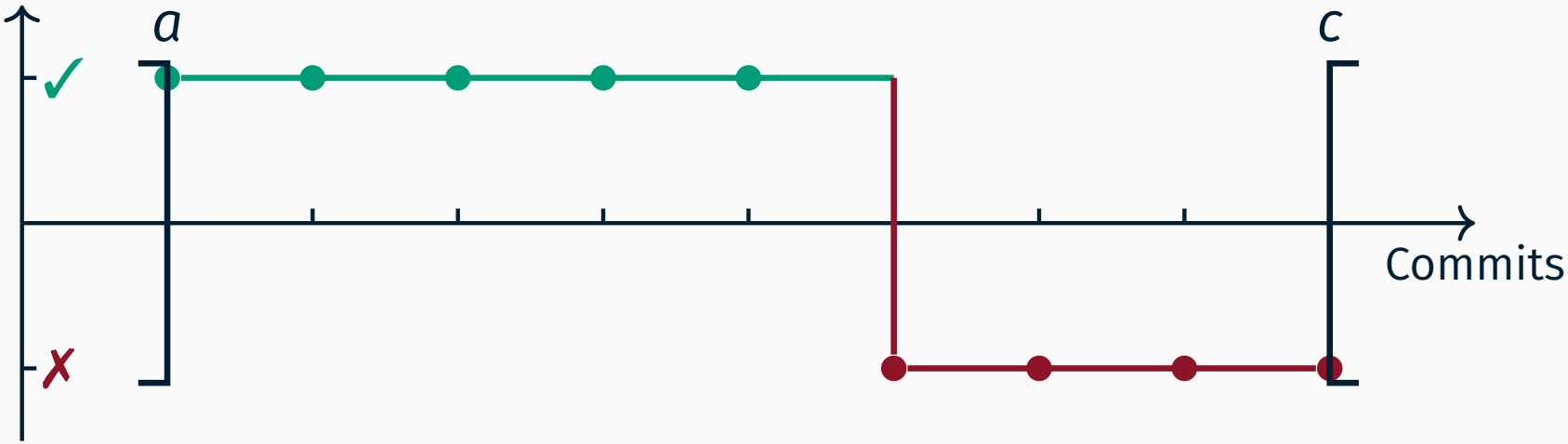
Verfahren (beinahe binäre Suche)

1. wähle Intervall $]a; c[$, so dass Fehler in c, kein Fehler in a
2. wähle b „in der Mitte“ zwischen a und c
3. prüfe b auf Fehler
   - Fehler: Wiederhole mit mit Intervall $]a; b[$
   - kein Fehler: Wiederhole mit mit Intervall $]b; c[$
4. fertig, wenn Intervall leer

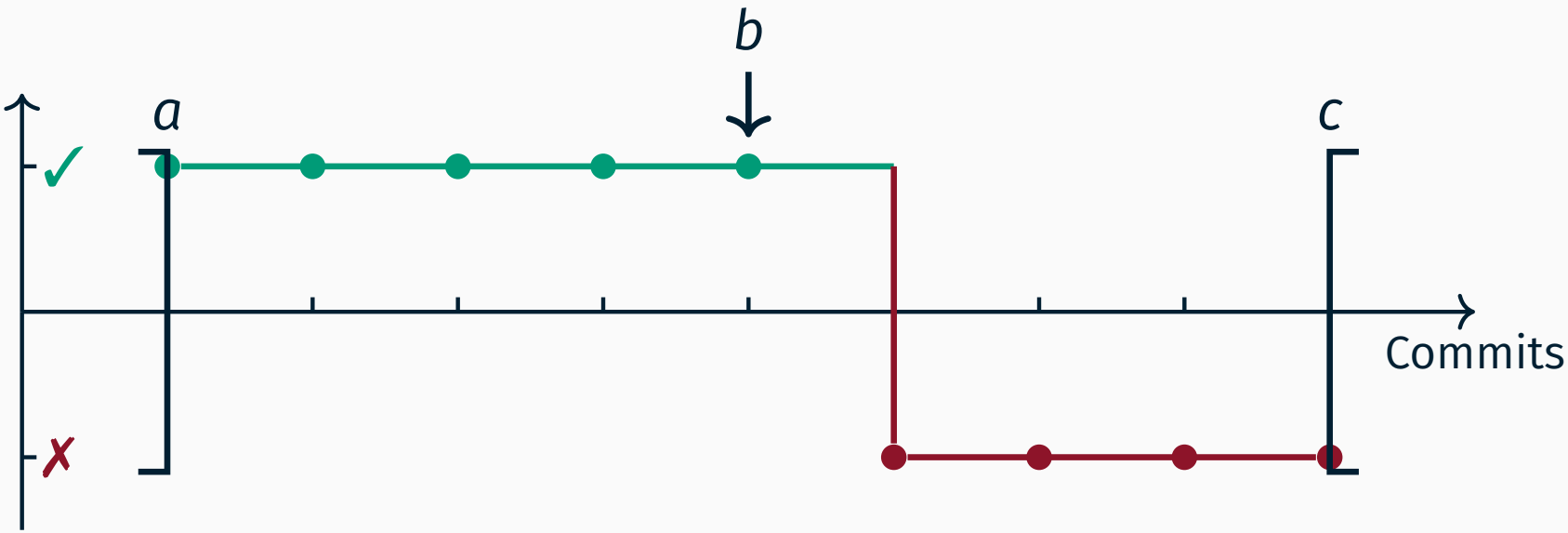
Bisektionsalgorithmus am Beispiel



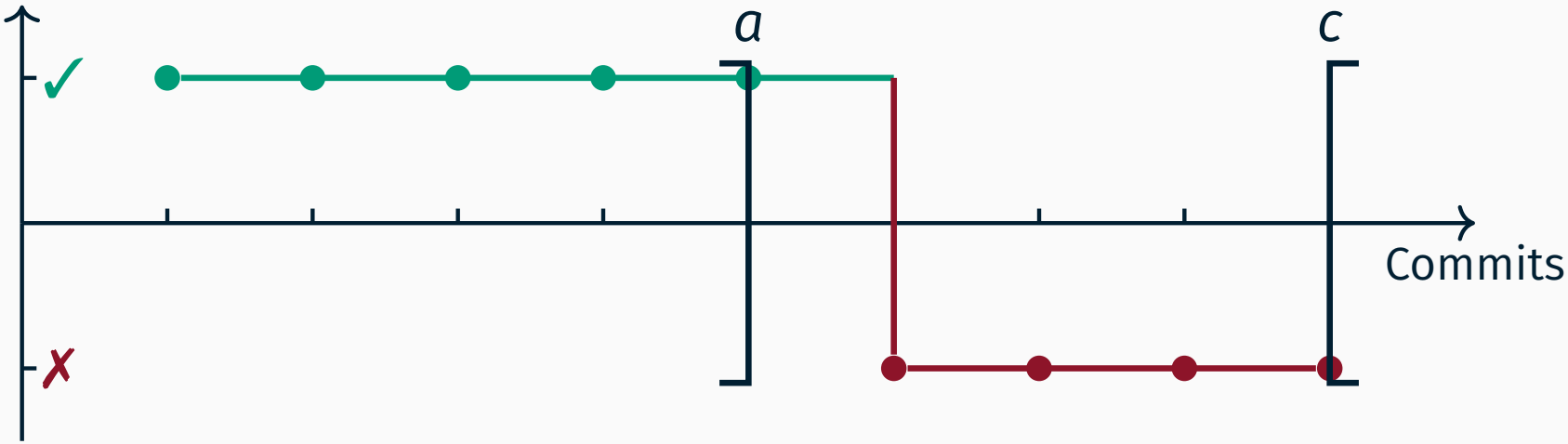
Bisektionsalgorithmus am Beispiel



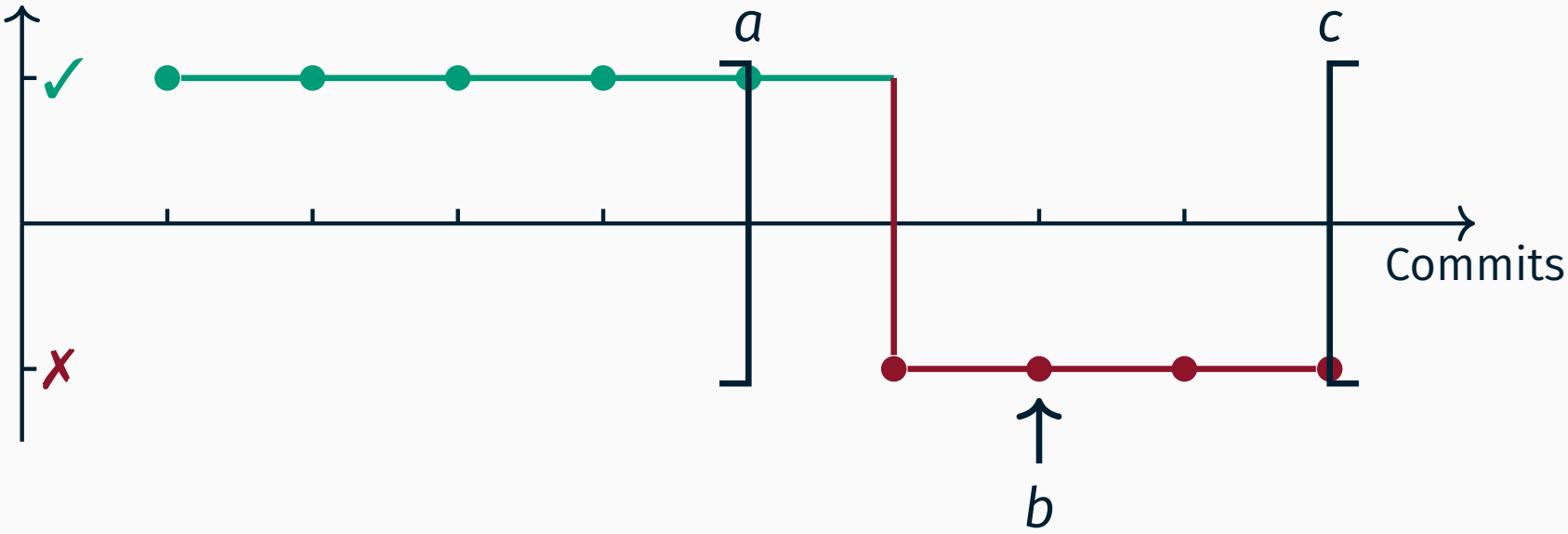
Bisektionsalgorithmus am Beispiel



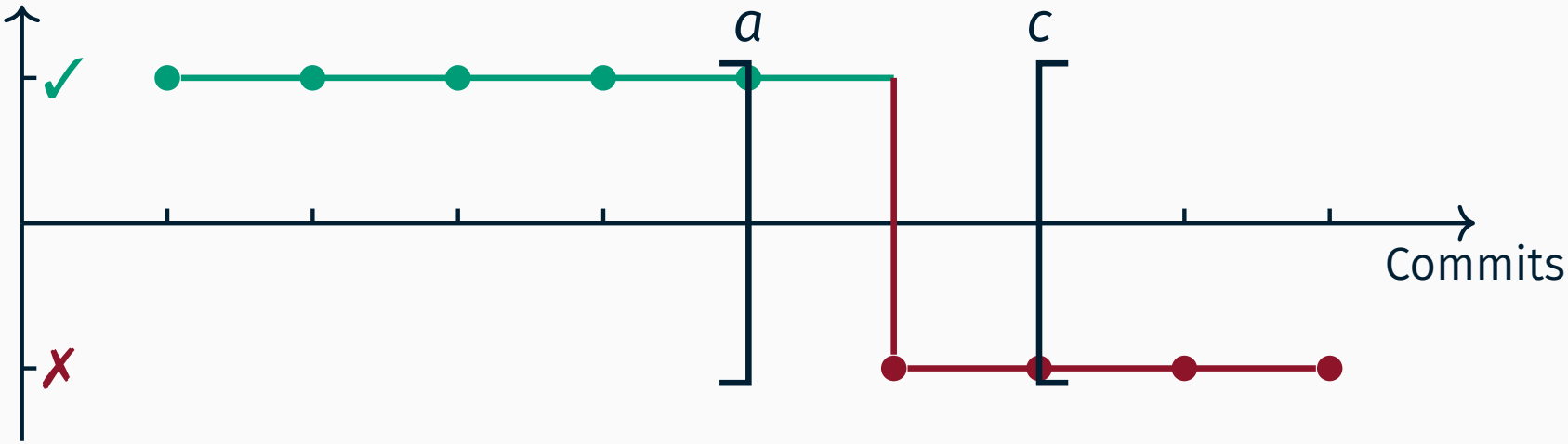
Bisektionsalgorithmus am Beispiel



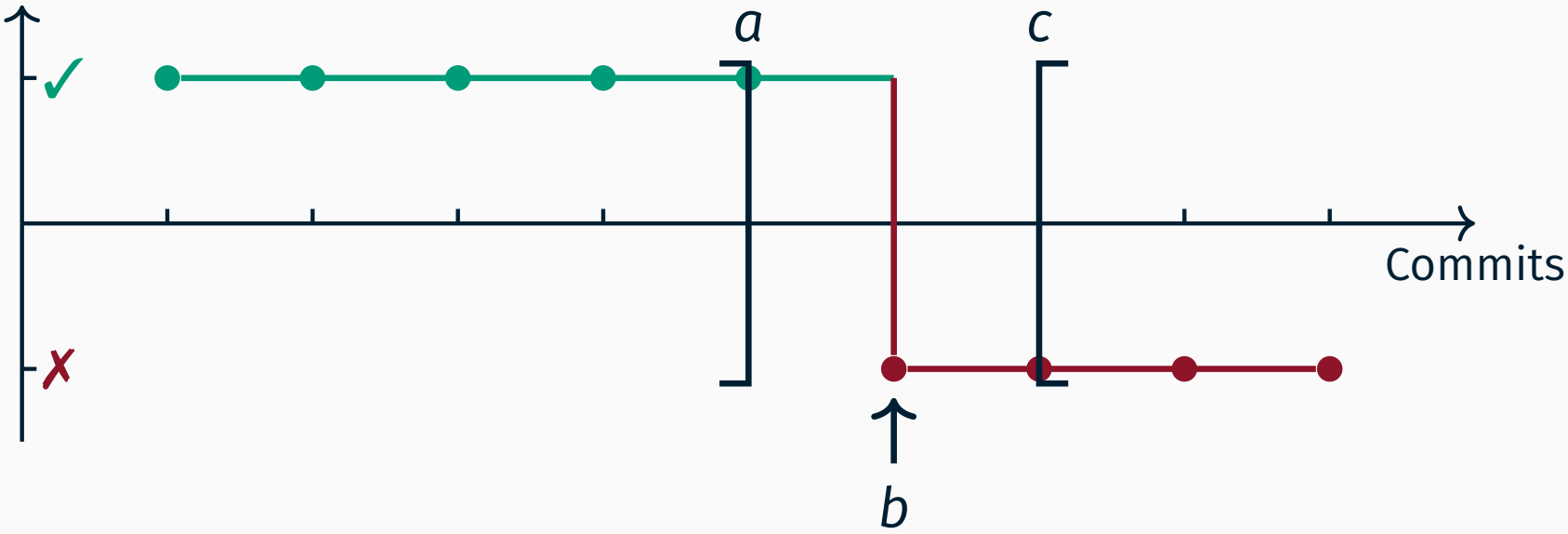
Bisektionsalgorithmus am Beispiel



Bisektionsalgorithmus am Beispiel



Bisektionsalgorithmus am Beispiel



Bisektionsalgorithmus in der Praxis

Verfahren mit Git

1. Bisektion starten & fehlerhaften (bad) / guten (good) Commit markieren

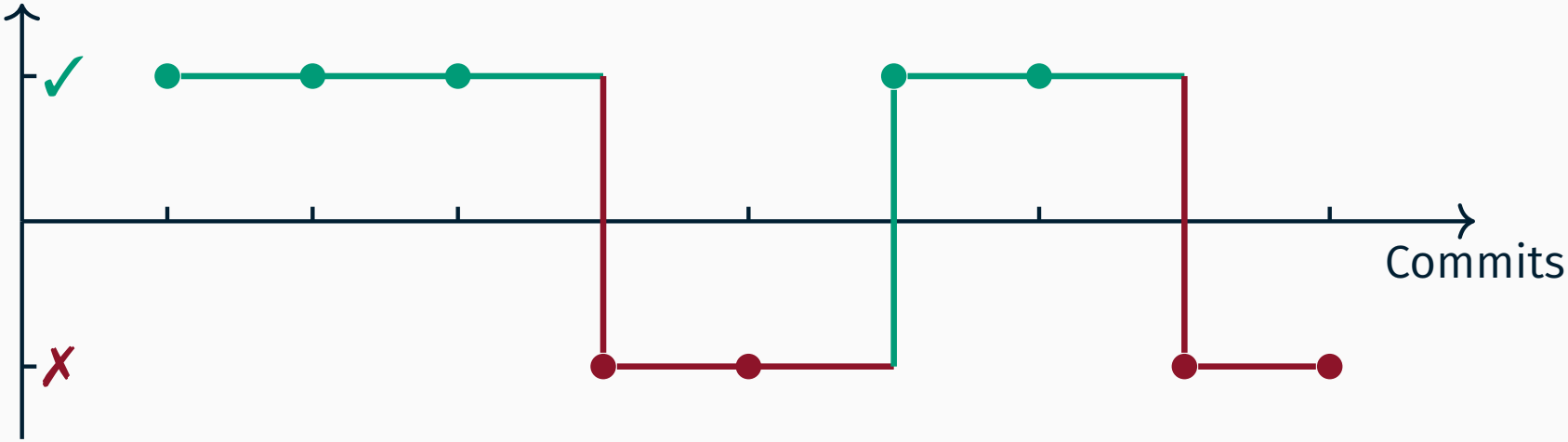
   ```
   git bisect start
   git bisect bad
   git bisect good HEAD~5
   ```

2. Testen auf Fehler
   - Bisektionsschritte werden automatisch ausgecheckt
   - Markieren mit `git bisect [good|bad|skip]`
3. Git zeigt an wenn Fehlerhafter Commit identifiziert

- Visualisierung des aktuellen Stands

  ```
  git bisect log
  git bisect visualize
  ```

Bisektionsalgorithmus in der Praxis



Die Commmithistory kann mehrere Übergänge enthalten!

spart Zeit…

- Binäre Suche nach Fehler auf Intervall von n Revsionen
- Jeder Schritt halbiert Intervall (ausser bei „skip“)
- Bester Fall: $\lceil \log_2 n \rceil$
  - ~10 Schritte für 1000 Revisionen
  - ~20 Schritte für 1 Mio. Revisionen
- Schlechtester Fall: $n - 1$ Schritte
  - Nur wenn keine Revision baut/testbar

Moral

Nur übersetzbaren Code in den Hauptentwicklungszweig!

spart noch mehr Zeit...

- Wenn bekannt ist, dass Fehler nur Teilmodul betrifft
  - können Commits zu anderen Modulen ignoriert werden
  - spart zusätzlich Bisektionsschritte

$\Rightarrow$ Nur Teilbäume betrachten

```
git bisect start -- src/subsystem/subsubsystem
```

Moral

Nur logisch zusammenhängende Änderungen in Commits!

und noch mehr Zeit...

- Nonplusultra:
  - \+ automatischen Test
  - \+ funktionierendes Build-Skript
  - \+ kleines Skript das beides aufruft
  - = automagisches `git bisect`

```
git bisect run ./test.sh
```

Moral

automatische & schnelle Testsuites und schnelle Build-Skripte sind toll

siehe gitlab.cs.fau.de/i4/git-bisect-demo.git

Linux Entwicklung

Freie Software

The four essentials freedoms[1]

„The freedom to...

0. ... run the program as you wish, for any purpose.
1. ... study how the program works, and change it so it does your computing as you wish.
2. ... redistribute copies so you can help your neighbor.
3. ... distribute copies of your modified versions to others.“

[1] Richard Stallman - „The Free Software Definition“

Freie/Libre Software und Quelloffene Software

- Streitpunkt über Bedeutung der Begriffe
- Diskussionsfrage:
  - Wie umgehen mit Änderungen und Redistribution?
  - Wie umgehen mit Änderungen die anderen Lizenzen unterliegen?
  - Aber auch: Stellt die GPL Freiheit sicher oder schränkt sie diese ein?
- Free/Libre Open Source Software als „Kompromiss“
  - Libre als bewusste Abgrenzung zum Gratisbegriff
  - „Think free as in free speech, not free beer.“

Linux-Upstream-Entwicklung

- „The Cathedral and the Bazaar” (Eric S. Raymond)
  - Essay über Methoden der (Open-Source) Software-Entwicklung
  - Basiert auf Beobachtungen des Entwicklungsprozesses des Linuxkern
- Hauptaussagen
  - „Every good work of software starts by scratching a developer’s personal itch.”

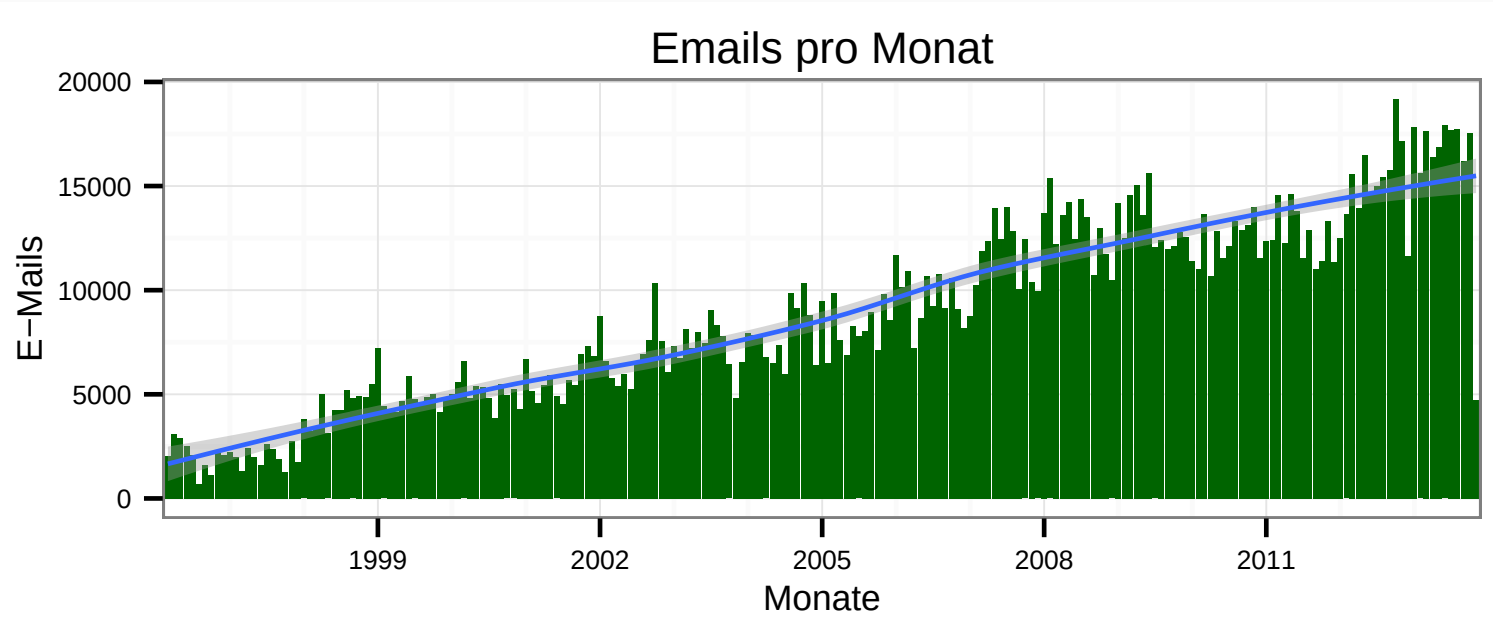
    Kapitel 2: „The Mail Must Get Through”
  - „Release early. Release often”

    Kapitel 4: „Release Early, Release Often”
  - „If you treat your beta-testers as if they’re your most valuable resource, they will respond by becoming your most valuable resource.”

    Kapitel 5: „Is A Rose Not A Rose?”

Linux-Upstream-Entwicklung (Fortsetzung)

- Gesamte **relevante** Kommunikation nur via Email
- Zentraler Kommunikationskanal:
  Linux-Kernel-Mailing-List (LKML)



- Ca. 5700 Abonnenten
- Weitere Verbreitung über RSS-Feeds, News u.Ä.

Kollaboration am Linuxkernel

- Änderungen durchlaufen Kreuzgutachten („Peer-Review”)
  - Gewachsene Hierarchie mit „Benevolent Dictator For Life”
  - Meistens siegt der überlegene Ansatz
- Festgelegter Prozess - Unabhängig ob Fix oder Feature

Einflussreichste Aktoren

Linus Torvalds

Andrew Morton

Alan Cox

Greg Kroah-Hartman

Entwicklungen absegnen (1/2)

- Patches enthalten generell nur Änderungen & deren Beschreibung
- Problem: Lizenzfragestellungen
  - Ist Sender des Patches auch sein Autor?
  - Wie dürfen die Änderungen genutzt werden?

$\Rightarrow$ Konvention

  - Patches müssen von Autoren abgesegnet werden
  - Spezielle Notation für Art des Beitrags

Entwicklungen absegnen (2/2)

```
Signed-off-by: Random J Dev <random@developer.example.org>
```

Bedeutung
(Auszug, Details in `Documentation/SubmittingPatches`)

- Der Autor bestätigt das Werk ganz oder in Teilen selbst geschrieben zu haben.
- Der Autor bestätigt das Recht zur Veröffentlichung zu haben
- Der Autor erlaubt die Verwendung und den Vertrieb der Änderung unter den Bedinungen der ursprünglichen Version

Weitere Absegnungen

- `Reported-by:` Wer hat das Problem (richtig) gemeldet
- `Reviewed-by:` Wer hat die Änderung begutachtet
- `Tested-by:` Wer hat den Patch getestet
- `Acked-by:` Patch ist zur Kenntnis genommen worden, nicht notwendigerweise aber getestet

Beispiel für Absegnungen

```
Subject: [PATCH] ACPICA: Fix possible fault in return package
object repair code

Fixes a problem that can occur when a lone package object is
wrapped with an outer package object in order to conform to
the ACPI specification. Can affect these predefined names:
_ALR,_MLS,_PSS,_TRT,_TSS,_PRT,_HPX,_DLM,_CSD,_PSD,_TSD

https://bugzilla.kernel.org/show_bug.cgi?id=44171

The bug got introduce by commit 6a99b1c94d053b3420eaa4a4bc
in v3.4-rc6, thus it needs to get pushed into 3.4 stable
kernels as well.

Reported-by: Vlastimil Babka <caster@gentoo.org>
Tested-by: Vlastimil Babka <caster@gentoo.org>
Tested-by: marc.collin@laboiteaprog.com
Signed-off-by: Bob Moore <robert.moore@intel.com>
Signed-off-by: Lin Ming <ming.m.lin@intel.com>
CC: stable@vger.kernel.org
---
 drivers/acpi/acpica/nspredef.c |    2 +-
 1 files changed, 1 insertions(+), 1 deletions(-)
```

Should I just stop attemting to make these trivial fixes?

LKML Post lkml.org/lkml/2004/12/20/255

Should I just stop attemting to make these trivial cleanups/fixes/whatever patches? are they more noice than gain? am I being a pain to more skilled people on lkml or can you all live with my, sometimes quite ignorant,patches?

Should I just stop attemting to make these trivial fixes?

LKML Post lkml.org/lkml/2004/12/20/255

Should I just stop attemting to make these trivial cleanups/fixes/whatever patches? are they more noice than gain? am I being a pain to more skilled people on lkml or can you all live with my, sometimes quite ignorant,patches?

- Oftmals wird der Nutzen von kleineren Patches unterschätzt
- Contributer fehlt der Mut den Patch einzureichen
- Und insbesondere **die Ausdauer** den Prozess auch bis zum Ende zu gehen

Should I just stop attemting to make these trivial fixes?

Nicht schüchtern sein

- FLOSS Projekte in der Regel froh über Beiträge
- Allerdings werden die ersten Einreichungen von Neulingen, auch bei scheinbar trivialen Änderungen, in ihrer ursprünglichen Form oftmals **nicht akzeptiert**
- Maintainer sind auch nur Menschen mit unterschiedlichen Persönlichkeiten.

Should I just stop attemting to make these trivial fixes?

Nicht schüchtern sein

- FLOSS Projekte in der Regel froh über Beiträge
- Allerdings werden die ersten Einreichungen von Neulingen, auch bei scheinbar trivialen Änderungen, in ihrer ursprünglichen Form oftmals **nicht akzeptiert**
- Maintainer sind auch nur Menschen mit unterschiedlichen Persönlichkeiten.
- Aber jeder Maitainer auch irgendwo seine persönliche Schmerzgrenze!

- Open Source Development and Sustainability: A Look at the Bouncy Castle Project $\Rightarrow$ BCCollabTalkSlides.pdf
- news.ycombinator.com/item?id=16851123

Checkliste: Einsenden von Patches in den Linuxkern (1/2)

- ✓ Patches im unified diff Format
- ✓ Sinnvolle und nachvollziehbare Beschreibungen der Änderungen
- ✓ Nur **eine** logische Änderung pro Patch (/Commit)
- ✓ Richtige(n) Adressat(en) finden
- ✓ Keine Dateianhänge, kein MIME, kein HTML

Checkliste: Einsenden von Patches in den Linuxkern (2/2)

- ✓ Basisversion genau angeben
- ✓ Knappe (70-75 Zeichen!) Kurzbeschreibung
  - 70-75 Zeichen
  - Zusätzlich ausführliche Patchbeschreibung und Intention
- ✓ Unnötige Diskussionen über Geschmack (etc.) vermeiden
  - ⇝ Siehe auch bikeshed.org

Weitere Resourcen

- Andi Kleen: „On submitting kernel patches”
  halobates.de/on-submitting-patches.pdf
- `linux/Documentation/SubmittingPatches`

Auffinden von zuständigen Betreuern

- Änderungen immer an den jeweiligen Betreuer (Maintainer) senden
- Zuständigkeiten für Teilbereiche ist aufgeteilt
- Hilfsmittel: `scripts/get_maintainer.pl`:

```
$ scripts/get_maintainer.pl -f fs/btrfs/volumes.c
Chris Mason <chris.mason@oracle.com>
linux-btrfs@vger.kernel.org
linux-kernel@vger.kernel.org
```

Tipp

Kann auch direkt auf einen Patch angewendet werden

Linux Coding Style

- Quelltext in Linux werden normalisiert bearbeitet
- `linux/Documentation/CodingStyle`
- automatisierter Test: `scripts/checkpatch.pl`

$\Rightarrow$ Erleichtert das Lesen und Verständnis

Linux Coding Style

- Quelltext in Linux werden normalisiert bearbeitet
- `linux/Documentation/CodingStyle`
- automatisierter Test: `scripts/checkpatch.pl`

$\Rightarrow$ Erleichtert das Lesen und Verständnis

... und vermeidet Auseinandersetzungen

Verwendet `checkpatch.pl`!

Verstöße gegen die Richtlinien führen häufig zur Ablehnung des Patches.

Häufige Probleme und Lösungen

- Auswahl des richtigen Branches
  - staging-next
  - linux-next
- beim Wiedereinreichen von Patchen:
  - Versionszähler im Betreff
    `[PATCH v3 2/4] fix something`
    `git format-patch --reroll-count`
  - Beschreibung des Patchdeltas zu Vorgängerversion
- Signed-off-by:
  - immer alle PASST Gruppenteilnehmer
  - Reihenfolge: Absender zuerst
  - nur funktionierende Email-Adressen!

Kernel Janitors - Die Hausmeisterei des Linuxkerns

- Gruppe von Kernelentwicklern die Code „aufräumt”
  - Vereinheitlichung von APIs
  - Fehlerhafter Code/Formatierung
  - Stellen ToDo Listen bereit
- Unterprojekt von `http://kernelnewbies.org`
- Hervorragende Anlaufstelle für erste Arbeiten am Kernel
  - ... leider leicht angestaubt

`http://kernelnewbies.org/KernelJanitors/`

Zusammenfassung

Zusammenfassung

- `git bisect` erlaubt Suche nach Revisionen die Fehler einführen
  - Erfordert funktionierende & fehlerhafte Revision
  - Suche mit Bisektionsalgorithmus
- Überblick über Entwicklungsprozess des Linuxkernels
  - Diskussion & Kommunikation über Email
  - Alle Änderungen durchlaufen Peer-Review-Prozess
- Für erfolgreiche Beiträge sind Konventionen zu beachten!

Aufgabe 4

Linux fixen. (1/2)

1. Programmcode analysieren und Defekte finden
   - Guter Anlaufpunkt: Linux-Staging in linux-next
   - „The Linux Staging Tree, what it is and is not.“
2. Patches für Fehler erstellen und ausreichend testen:
   - Patch anwenden
   - Kompilieren
   - Sicherstellen, dass der betroffene Code übersetzt wurde
     `#error “Attempt to compile this“`
   - Überprüfung des Patches mit `scripts/checkpatch.pl`

Linux fixen. (2/2)

3. Patches an linux-kernel@i4.cs.fau.de senden, OK abholen
4. Patches an `devel@linuxdriverproject.org` mit Kopie (`CC:`) an die zuständigen Maintainer (`get_maintainer.pl`) und linux-kernel@i4.cs.fau.de senden
5. Vorstellung der Ergebnisse in der Tafelübung (Diskussionsrunde)
   - Tipp: 2-3 Übersichtsfolien sind Hilfreich

Bearbeitung bis 21. Juni

Vorstellung der Ergebnisse in der Tafelübung am 27. Juni

Fragen?