

Verlässliche Echtzeitsysteme

Übungen zur Vorlesung

Florian Franzmann, Martin Hoffmann, Tobias Klaus

Friedrich-Alexander-Universität Erlangen-Nürnberg
Lehrstuhl Informatik 4 (Verteilte Systeme und Betriebssysteme)
<http://www4.cs.fau.de>

4. Juni 2014



Überblick

1 C-Quiz Teil I

2 Versionsverwaltung mit git



Die Programmiersprache C

- C ist eine sehr alte Programmiersprache
 - Der C-Sprachumfang ist überschaubar
 - Deswegen denken viele Leute C sei einfach
 - **Das stimmt so leider nicht**
- ↪ C folgt nicht dem *Prinzip der geringsten Verwunderung!*

Auch heute noch viel sicherheitskritische Software in C

↪ In jeder Übung ein kleines Quiz zum Thema C-Gemeinheiten



Annahmen

- C99
- x86 bzw. x86-64, d. h.
 - vorzeichenbehaftete Integer als Zweierkomplement implementiert
 - char hat 8 Bit
 - short hat 16 Bit
 - int hat 32 Bit
 - long hat 32 Bit auf x86 und 64 Bit auf x86-64



Frage 1

Zu was wird `1 > 0` ausgewertet?

1. 0
2. 1
3. nicht definiert

Erklärung

- Jeder Wert ausser 0 ist in C wahr.
- Vergleichende Operatoren geben laut C-Standard entweder 1(wahr) oder 0(falsch) zurück.



Frage 2

Zu was wird `1U > -1` ausgewertet?

1. 0
2. 1
3. nicht definiert

Erklärung

- `unsigned` gewinnt bei impliziter Typumwandlung.

~> `1U > -1U ⇒ 1U > UINT_MAX`



Frage 3

Angenommen: `int x = 1`; Zu was wird `(unsigned short)x > -1` ausgewertet?

1. 0
2. 1
3. nicht definiert

Erklärung

- vor dem Vergleich beide Operanden nach `int` umgewandelt
- weil dies ohne Wertverlust geschehen kann
- ~> hier werden zwei `signed`-Werte verglichen
- ~> ein `unsigned int` würde nicht umgewandelt werden!



Überblick

1 C-Quiz Teil I

2 Versionsverwaltung mit git



Anforderungen

Typische Aufgaben eines Versionsverwaltungssystems sind:

- *Sichern* alter Zustände (⇒ commits)
- *Zusammenführung* paralleler Entwicklung
- *Transportmedium*

Idealerweise zusätzlich:

- *Unabhängige Entwicklung* ohne zentrale Infrastruktur



git

- wir werden in VEZS git verwenden
- 2005 von Linus Torvalds für den Linux-Kernel geschrieben
- Konsequenz der Erfahrungen mit *bitkeeper*
- Eigenschaften:
 - dezentrale, parallele Entwicklung
 - Koordinierung hunderter Entwickler
 - Visualisierung von Entwicklungszweigen



git-Arbeitsschritte

- initial Repository herunterladen:

```
% git clone <URL>
```

- oder anlegen:

```
% git init
```

- Commit im Index zusammenbauen (⇒ „*Verladerampe*“):

```
% git add <Datei1>
```

```
% git add <Datei2>
```

```
% ...
```

- anschauen was bei git commit passieren würde:

```
% git status
```

oder

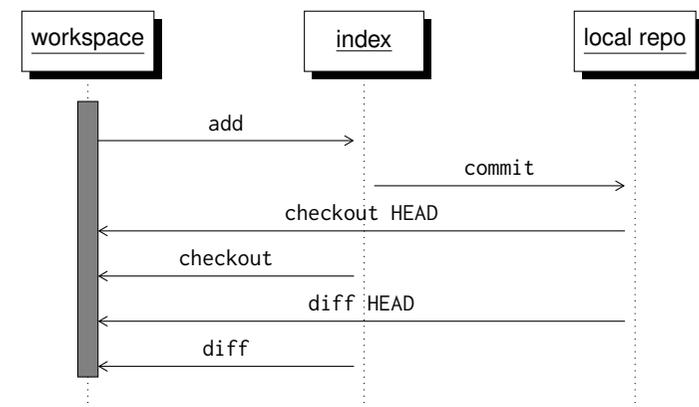
```
% git diff --cached
```

- anschließend Index an das Repository übergeben:

```
% git commit (⇒ „Einladen in den LKW“)
```



git-Arbeitsschritte – lokal



git-Kommandos: Lokale Quellcodeverwaltung I

- Repository erstellen:
`% git init`
- Änderung hinzufügen:
`% git add <Datei>`
- oder interaktiv:
`% git add -i`
- feingranulares hinzufügen:
`% git add -p`
- Änderungen einchecken:
`% git commit -i <Datei1> <Datei2> ...`



git-Kommandos: Lokale Quellcodeverwaltung II

- alles was nicht im git ist löschen:
`% git clean -d <Pfad>`
nur anzeigen, was gelöscht werden würde:
`% git clean -n -d <Pfad>`
- herausfinden was beim nächsten Commit verändert wird:
`% git diff --cached`
- oder als Kurzzusammenfassung:
`% git status`
- geänderte aber noch nicht eingetragene Datei zurücksetzen:
`% git checkout -- <Datei>`



git-Kommandos: Lokale Quellcodeverwaltung III

- das Log anschauen:
`% git log`
mit Graph:
`% git log --graph`
- herausfinden, was im letzten Commit verändert wurde:
`% git whatchanged`
- einen Commit rückgängig machen:
`% git revert <commit-id>`
- Änderungen sichern, aber noch nicht einchecken:
`% git add ...`
`% git stash`

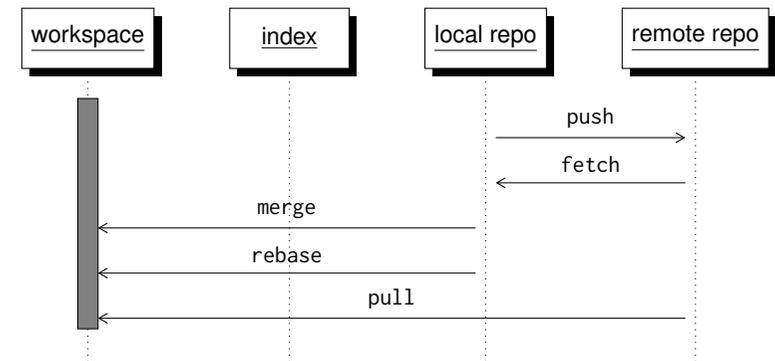


git-Kommandos: Lokale Quellcodeverwaltung IV

- gesicherte Änderungen wieder hervorholen:
`% git stash apply`
- Stashinhalt anzeigen:
`% git stash list`
- Stash-Element löschen:
`% git drop <id>`
- einen Branch anlegen:
`% git branch <Name>`
- alle registrierten Branches anzeigen:
`% git branch -a`
- zu einem Branch wechseln:
`% git checkout <Name>`



- menügeführt das Repository befragen:
% tig



git push [<remote> [<branch>]]

- schiebt Commits nach <remote> in den ausgewählten <branch>
- dies geht nur, wenn lokales Repo auf dem aktuellen Stand ist!
- sonst beschwert sich git:

```
% git push origin master
```

```
To /tmp/test.git
! [rejected]        master -> master (non-fast-forward)
error: failed to push some refs to '/tmp/test.git'
To prevent you from losing history, non-fast-forward updates were rejected
Merge the remote changes (e.g. 'git pull') before pushing again. See the
'Note about fast-forwards' section of 'git push --help' for details.
```

~ wir müssen das Repository erst auf den aktuellen Stand bringen



git pull [<remote> [<branch>]]

- holt Änderungen aus remote in den aktuellen Branch
- verschmilzt aktuellen Branch mit geholten Änderungen
- gleicher Effekt wie % git fetch && git merge FETCH_HEAD

```
% git pull origin
```

```
remote: Counting objects: 5, done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From /tmp/test
 38b95cb..8ec6e93 master -> origin/master
Auto-merging test.txt
CONFLICT (content): Merge conflict in test.txt
Automatic merge failed; fix conflicts and then commit the result.
```

- jemand hat in der Zwischenzeit die gleiche Stelle der Datei verändert

~ Konflikte müssen von Hand behoben werden



Konflikt beheben

■ % cat test.txt

```
hallo
<<<<<< HEAD
we!T!   meine Version
=====
Welt!   Version in origin/master
>>>>>> 8ec6e9309fa37677e2e7ffc9553a6bebf8827d6
```

~ sich für eine von beiden Versionen entscheiden

■ Konflikt auflösen:

% git add test.txt && git commit

```
[master 4d21871] Merge branch 'master' of /tmp/test
```

% git push origin master

```
Counting objects: 5, done.
Writing objects: 100% (3/3), 265 bytes, done.
Total 3 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
To /tmp/test.git
 8ec6e93..278c740  master -> master
```

■ juhu!



git-Kommandos: Austausch von Quellcode I

■ initiales *Klonen*:

```
% git clone http://www4.cs.fau.de/...
```

■ Einspielen entfernter Änderungen:

```
% git pull
```

⇒ äquivalent zu

```
% git fetch && git merge
```

■ Mehrere Repositories registrieren:

```
% git remote add 32-stable git://git.kernel.org/.../...
```

■ registrierte Remotes untersuchen:

```
% git remote -v
```



git-Kommandos: Austausch von Quellcode II

■ alle Remotes nachladen (aktueller Branch wird nicht verändert)

```
% git remote update
```

■ lokalen Branch aus dem neuen 'Remote' anlegen:

```
% git checkout -b work 32-stable/master
```

■ Unterschiede zwischen lokalem und entferntem Branch untersuchen:

```
% git log ..origin/master
```

■ aktuelle Änderungen auf dem entfernten Branch neu aufspielen:

```
% git pull --rebase
```

■ die neuste Änderung untersuchen:

```
% git show
```



git-Kommandos: Austausch von Quellcode III

■ herausfinden wer für welche Zeilen einer Datei verantwortlich ist:

```
% git blame
```

■ die letzten drei Änderungen als Patch:

```
% git format-patch HEAD~~
```

■ Sendeziel für Patchversand per E-Mail vorgeben:

```
% git config sendemail.to=...@...
```

■ Patchset letzten drei Änderungen per E-Mail senden:

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

■ einen Patch aus einer Mailbox anwenden:

```
% git am < <Datei>
```



Arbeitsablauf mit Branches

In den meisten Versionsverwaltungssystemen

1. Featurebranch anlegen
2. Feature im Branch implementieren, testen
3. Featurebranch mit master verschmelzen
4. ggf. Featurebranch löschen

Naiver Ansatz

~> skaliert nicht!



Warum branch/edit/merge nicht skaliert

Aufgaben von Versionsverwaltung

1. Codeschreiben unterstützen
2. Konfigurationsmanagement/Branches
~> z. B. Release-Version, HEAD-Version ...

~> Konflikt

1. braucht Checkpoint-Commits
 - möglichst oft einchecken
 - ~> skaliert nicht
2. braucht Stable-Commits
 - nur einchecken, wenn Commit perfekt
 - ~> nicht praktikabel



Lösung mit git: öffentlicher vs. privater Branch

Öffentlicher Branch ~> verbindliche Geschichte

Commits sollen

}	atomar	}	sein
	gut dokumentiert		
	linear		
	unveränderlich		

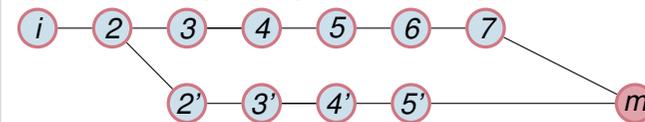
Privater Branch ~> Schmierpapier

- für einzelnen Entwickler
- möglichst lokal
- wenn im zentralen Repo ~> auf Privatheit einigen



Aufräumen

- verschmelze nie direkt privaten mit öffentlichem Branch
 - Historie wird sonst unübersichtlich
- ~> nicht einfach git merge im master machen



- vorher immer erst git
 - rebase ~> Commits auf Branch anwenden
 - merge --squash ~> einzelnen Commit aus Branch-Commits
 - commit --amend ~> letzten Commit überarbeiten
- Ziel: öffentlicher Commit ≡ Kapitel eines Buches

Michael Crichton

Great books aren't written – they're rewritten.



Arbeitsablauf für kleinere Änderungen

- `git merge --squash`
↳ zieht Änderungen aus einem Branch in den aktuellen Index

Branch

```
% git checkout -b private_feature_branch (Branch anlegen)
% touch file1.txt file2.txt
% git add file1.txt; git commit -am "WIP1" (file1.txt einchecken)
% git add file2.txt; git commit -am "WIP2" (file2.txt einchecken)
```

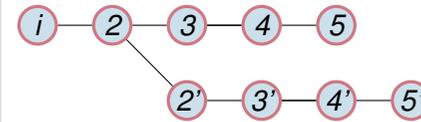
Merge

```
% git checkout master (nach master wechseln)
% git merge --squash private_feature_branch
(Änderungen auf Index von master anwenden)
% git commit -v (Änderungen einchecken)
```



git rebase <branch>

- Aufsetzen auf bestehenden <branch>

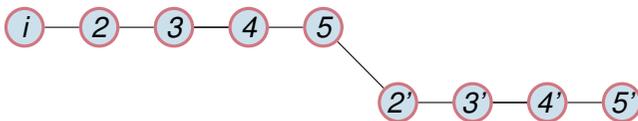


- Patches aus dem „unteren“ Zweig werden auf den „oberen“ aufgespielt
- Die Historie ist nun linear
- Linearisierte Änderungen lassen sich häufig einfacher bewerten
- **Vorsicht!**
 - Verzweigungen vom alten Zweig können nun nicht mehr zusammengeführt werden
 - Keine gemeinsamen Vorgänger mehr
 - Visualisierung der Historie ist nun bestenfalls verwirrend



git rebase <branch>

- Aufsetzen auf bestehenden <branch>



- Patches aus dem „unteren“ Zweig werden auf den „oberen“ aufgespielt
- Die Historie ist nun linear
- Linearisierte Änderungen lassen sich häufig einfacher bewerten
- **Vorsicht!**
 - Verzweigungen vom alten Zweig können nun nicht mehr zusammengeführt werden
 - Keine gemeinsamen Vorgänger mehr
 - Visualisierung der Historie ist nun bestenfalls verwirrend



git rebase --interactive <commit>

- schreibt Geschichte um

```
% git rebase --interactive ccd6e62^
```

pick ↳ übernimmt Commit

```
pick ccd6e62 Work on back button
pick 1c83feb Bug fixes
pick f9d0c33 Start work on toolbar
```

fixup ↳ verschmilzt Commit mit Vorgänger

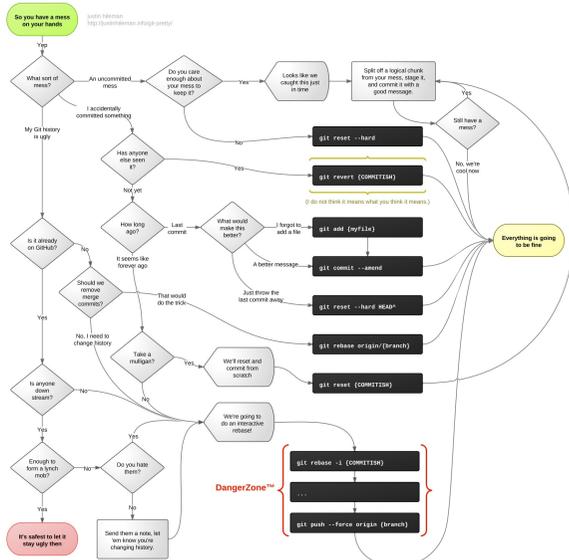
```
pick ccd6e62 Work on back button
fixup 1c83feb Bug fixes # mit Vorgaenger verschmelzen
pick f9d0c33 Start work on toolbar
```

reword ↳ Beschreibung editieren

edit ↳ kompletten Commit editieren



Geschichte neuschreiben



git push --force



Wenn der Feature-Branch im Chaos versinkt?

~ aufgeräumten Branch anlegen

1. auf Branch master wechseln

```
% git checkout master
```

2. Branch aus master erzeugen

```
% git checkout -b cleaned_up_branch
```

3. Branch-Änderungen in den Index und die Working Copy ziehen

```
% git merge --squash private_feature_branch
```

4. Index zurücksetzen

```
% git reset
```

■ danach Commits neu zusammenbauen

~ git cola

git reflog

■ Zeigt die Befehls Geschichte

```
git reflog
```

```
8afd010 HEAD@{0}: rebase -i (finish): returning to refs/heads/master
8afd010 HEAD@{1}: checkout: moving from master to 8afd010ae2ab48246d5
7f97fab HEAD@{2}: commit: Pentax K20D fw version 1.04.0.11 wb presets
8c37332 HEAD@{3}: rebase -i (finish): returning to refs/heads/master
8c37332 HEAD@{4}: checkout: moving from master to 8c373324ca196c337dd
9d66ec9 HEAD@{5}: clone: from git://github.com/darktable-org/darkt...
```

■ git reset --hard HEAD@{2} stellt alten Zustand wieder her

Nützliche Aliase

.bashrc

```
function git_current_branch() {
  git symbolic-ref HEAD 2> /dev/null | sed -e 's/refs\/heads\/\\/'
}

# git push ohne tracking
alias gpthis='git push origin HEAD:${git_current_branch}'
# alle branches holen und dann rebase
alias gup='git fetch origin && git rebase -p origin/${git_current_branch}'
```

~ <https://gist.github.com/geelen/590895>



Euer Gruppenrepository

- bitte eine Mail an hoffmann@cs.fau.de mit
 - den Namen der Gruppenmitglieder
 - euren öffentlichen SSH-Schlüsseln
 - ~ % `ssh-keygen -t rsa -f ~/.ssh/i4git`
 - ~ die Datei `~/.ssh/i4git.pub` an die E-Mail anhängen
- SSH konfigurieren:
 - die Datei `~/.ssh/config` anpassen:

```
Host i4git
  HostName i4git.informatik.uni-erlangen.de
  User gitosis
  IdentityFile ~/.ssh/i4git
  ForwardAgent no
  ForwardX11 no
```
- Repository klonen:

```
% git clone ssh://i4git/vezs_ss14_<nummer>.git vezs-uebung
```



git-Konfiguration für die erste Übungsaufgabe

.git/config

```
[remote "origin"]
  fetch = +refs/heads/*:refs/remotes/origin/*
  url = ssh://i4git/vezs_ss14_<nummer>.git

[remote "vorgabe-astime"]
  fetch = +refs/heads/*:refs/remotes/origin/*
  url = https://www4.cs.fau.de/Lehre/SS14/V_VEZS/Vorgaben/astime.git
```



Lesenswertes zu git

- <http://gitready.com>
- <http://book.git-scm.com/>
- <http://gitcasts.com>
- <http://eagain.net/articles/git-for-computer-scientists/>
- <http://sandofsky.com/blog/git-workflow.html>
- <http://365git.tumblr.com/>
- <http://blog.sensible.io/post/33223472163/git-to-force-push-or-not-to-force-push>

