B-BKSPP-PR (WS 2013/2014)
Einführung in die Systemprogrammierung mit C
Teil 1b: Werkzeuge

26. November 2013

Gute Softwareentwickler sind durchaus in der Lage, kleinere Probleme komplett „im Kopf“ zu lösen und sehr direkt niederzuschreiben; manchmal soger mit wenigern oder keinen Fehlern. In realistischen Programmen ist dies aber nicht so einfach:

- Realistische Probleme sind selten klein
- Neue Lösungen für realistische Probleme müssen normalerweise mit existierenden Programmkomponenten interagieren
- Existierende und neue Programmkomponenten haben normalerweise bekannte und unbekannte Programmierfehler
- Lösungen für realistische Problem sind oft nicht neu, sondern vielmehr Veränderungen existierender Lösungen
- Einzelne Softwareentwickler lösen Probleme meist nicht alleine, sondern in Kollaboration mit anderen Softwareentwicklern.

Aus diesen und anderen Gründen ist es nicht ausreichend, Softwareentwicklung als eine einfache Pipeline der Form ‘Problem → Lösungsidee → Implementierung → Test → Auslieferung’ zu begreifen[1]; die Entwicklungsstrategie pendelt vielmehr zwischen diesen Phasen hin und her.

All dies führt zu vielerlei Komplikationen, die von *Software-Engineering-Werkzeugen* zumindest teilweise aufgefangen werden. Software-Engineering-Werkzeuge sind Werkzeuge, die die Softwareentwicklung vereinfachen; sie fallen grob in zwei teilweise überlappende Kategorien:

- *Software-Werkzeuge* arbeiten direkt mit Software, um diese zu analysieren, zu transformieren, oder zu verwenden.
- *Verwaltungswerkzeuge* verwalten *Metainformationen* über Software, also abstrakte Eigenschaften der Programme wie z.B. deren Korrektheit, Performanz, oder Dokumentation.

Wir werden uns hier auf Software-Werkzeuge konzentrieren, allerdings auch einige Werkzeuge verwenden, die in beide Bereiche fallen.

[1] Zwar existieren Softwareentwicklungsmethodologien, die dieses Konzept verfolgen, aber sie sind deutlich aufwendiger als alternative Entwicklungsmethodologien und müssen in der Praxis mit formaler Verifikation verknüpft werden— nur die teuersten Großprojekte (Satelliten, Flugzeuge, . . .) leisten sich diesen Aufwand.

0.1 Punkte

In diesem Teil des Praktikums sind 17 Punkte und 7 Bonuspunkte erreichbar; Notation (x + yB), wobei x Punkte und y Bonuspunkte sind . Dabei gilt:

- Der Praktikumsteil gilt als bestanden, wenn die Summe der Punkte und Bonuspunkte insgesamt mindestens 13 erreicht.
- Die Anzahl der Bonuspunkte für eine Aufgabe ist begrenzt durch die Anzahl der Punkte für die gleiche Aufgabe. Eine Aufgabe mit (**2** + 2B) Punkten, deren Bonusaufgaben komplett korrekt bearbeitet sind, deren Hauptaufgaben aber fehlen, wird somit mit 0 Punkten angerechnet.

0.2 Dokumentation

In diesem Teil werden Sie mit einigen neuen Funktionen und Werkzeugen konfrontiert. Diese sind größtenteils separat dokumentiert. Teil Ihrer Aufgabe ist es, die relevanten Stellen der Dokumentation zu finden und zu interpretieren. In komplexeren Fällen wird Ihnen dazu in diesem Dokument direkte Hilfestellung gegeben, aber andernfalls werden Sie oft Doukumentation wie folgt referenziert finden:

- `http:// ...`: Webseite mit relevanten Informationen
- **foo**: `man`-Seite mit relevanten Informationen. Beachten Sie, daß unter Umständen mehrere Seiten mit gleichem Namen in verschiedenen Abschnitten existieren. Manchmal wird Doukumentation auch mit dem entsprechenden Abschnitt referenziert, z.B. **open(2)**, aber nicht immer.

0.3 Einsendung

Senden Sie Ihre Lösung bis zur Abgabedeadline (23.11.2013, 18:00) in folgendem Format ein:

- Verpackt in einer einzigen `.tar.gz` oder `zip`-Datei
- Der Name der Datei muß entweder mit den Namen der Einsendenden beginnen, also z.B. `laura_heinzelmann__hans-peter_kloebner`, oder mit dem Gruppennamen.
- Die Beiträge zu den verschiedenen Teilaufgaben müssen in Unterverzeichnissen abgelegt werden, die der Nummer der Aufgabe entsprechen (also z.B. `1`, `2`, ...)
- Die Datei *muß* alle für die Aufgabe erstellten Quellcode-Artefakte beinhalten (also alles, was von Hand geschrieben wurde).
- Die Datei *muß* alle zusätzlichen geforderten Informationen enthalten. Wenn explizite Fragen gestellt wurden, müssen diese in einer Datei `ANTWORTEN` angegeben werden.
- Wenn Sie Bonusaufgaben bearbeitet haben, müssen Sie dies explizit in der Datei ANTWORTEN erwähnen und erklären.
- Die Datei *darf* Binärartefakte (insbesondere auch das `.git`-Verzeichnis) enthalten.
- Wenn die Datei größer als 1 MB ist oder aus anderen Gründen nicht leicht per Mail versendet werden kann, darf sie in mehrere Dateien geteilt werden (**split(1)**).
- Die Datei muß per Mail an den Veranstalter gesendet werden.

Die Verzeichnisstruktur in der Archivdatei sollte z.B. also wie folgt aussehen (bei zwei bearbeiteten Aufgaben):

```
.
|-- 1
|   |-- programm.c
|   `-- ANTWORTEN
`-- 2
    |-- Makefile
    |-- foo.h
    |-- foo.c
    `-- ANTWORTEN
```

Alternativ können Sie zur Einsendung auch `git` verwenden. Dazu ist für Ihre Gruppe folgendes Repository eingerichtet: textttgit@web.sepl.cs.uni-frankfurt.de:${GRUPPENNAME}/abgabe-1b

0.4 Sprache

Im Folgenden werden für verschiedene Fachkonzepte die englischen Begriffe verwendet, sofern diese Begriffe von den vorgestellten Werkzeugen eingesetzt werden. Die Absicht dahinter ist, die Interpretierung der Werkzeugausgaben und -Dokumentation zu vereinfachen.

0.5 Vorbereitung zur Gruppenarbeit

Ab Aufgabe 3 erhalten Sie die Möglichkeit, Quellcode auf einem unserer Server abzulegen. Um sich auf diesem Server authentifizieren zu können, müssen Sie dem Veranstalter Ihren *öffentlichen* DSA-Schlüssel für ssh-Verbindungen senden[2]. Dieser liegt in der Datei

```
~/.ssh/id_dsa.pub
```

Wichtig: Die Datei `/.ssh/id_dsa` (ohne `.pub`) beinhaltet ihren *privaten* Schlüssel, den Sie auf keinen Fall weiterverbreiten sollten.

Wenn sie diese Dateien noch nicht haben, können Sie sie per

```
ssh-keygen -t dsa
```

erzeugen.

[2] `http://wiki.ubuntuusers.de/SSH#Authentifizierung-ueber-Public-Keys`

1 Einfache Zeitmessung ($1 + \frac{1}{2}$B)

Der Befehl **time(1)** kann auf der Kommandozeile verwendet werden, um die Gesamtlaufzeit eines Programmes auszugeben. Stellen Sie den Befehl einfach dem Programmaufruf voran, z.B.

```
time ./mein-programm
```

(falls sie ein Programm namens `mein-programm` im aktuellen Verzeichnis haben), und Sie erhalten nach Ausführung des Programmes eine Ausgabe der folgenden Form:

```
real    0m4.000s
user    0m2.000s
sys     0m1.000s
```

Die Bedeutung ist wie folgt:

- `real` ist die Zeit vom Start des Programmes bis zu dessen Ende. Diese 'wallclock time' ist die übliche Zeit für Geschwindigkeitstests.
- `user` ist die Zeit, die das Programm tatsächlich auf dem Prozessor gelaufen ist. Bei parallelen Programmen wird dabei die Zeit aller verwendeten Prozessorkerne addiert.
- `sys` ist die Zeit, die der Betriebssystemkern mit der Beantwortung von direkten Anfragen des Programmes (z.B. Dateizugriff) verbringt.

Aufgaben.

1. Vergleichen Sie die Laufzeit Ihre Bubble-Sort-Implementierung aus Teil 1a mit Quicksort (**qsort(3)**) aus der C-Standardbibliothek. Erzeugen Sie dazu ein `int`-Array der Größe 100.000 und initialisieren Sie dessen Felder zufällig per **rand(3)**. Schreiben Sie zwei separate Programme, von denen eines **qsort(3)** und das Andere Ihren Bubble-Sort verwendet.
2. Messen Sie die Gesamtausführungszeit (`real`) fünf Mal pro Programm.
   (a) Welche Implementierung ist im Median schneller, und um wie viel?
   (b) Wie stark variieren die Messungen?

(Bonus) Führen Sie zwei weitere Vergleichsmessungen durch:

- **qsort** im Vergleich zu der korrigierten quicksort-Implementierung aus Aufgabe 4
- **qsort** mit optimierendem Übersetzer (`-O3`) versus **qsort** mit nicht optimierendem Übersetzer (`-O0`). Beachten Sie dabei, daß **qsort** eine Bibliotheksfunktion ist, die nicht neu übersetzt wird.

2 Textdifferenzen (1)

Angenommen, wir haben ein existierendes Programm verbessert und möchten dem Autor des Programmes unsere Verbesserung zusenden. Eine einfache Möglichkeit, dies zu tun, ist, die modifizierten Quellcodedateien von Hand herauszusuchen und zu senden. Das Problem dabei ist, daß der ursrpüngliche Autor mit mehreren Problemen zu kämpfen hat:

- Die Dateien zu vergleichen, um die Änderungen zu finden
- Die Änderungen mit anderen Änderungen der gleichen Datei zu integrieren, selbst wenn diese Änderungen völlig andere Teile der Datei betreffen.

Um dieses Problem zu umgehen, existiert das Konzept einer *Programmdifferenz*. Die grundlegendste Form dieser Differenz ist das Text-*Diff* (meist nur *Diff* genannt): eine Beschreibung der hinzugefügten, geänderten, und entfernten Zeilen des Programmes. Außer Zeilendifferenzen existieren auch intelligentere, sogenannte *semantische* Diffs, aber dieses Gebit ist noch Teil der Forschung.

Um die Verwendung solcher Diffs zu unterstützen, existieren unter UNIX zwei Programme:

- **diff**, das zwei Dateien (oder zwei Verzeichnisse) miteinander vergleicht und Differenzen ausgibt
- **patch**, das eine Datei oder ein Verzeichnis mit den in einer von **diff** erzeugen Programmdifferenz modifiziert. Aufgrund dieser dualen Funktion werden Diffs auch manchmal auch als Patches bezeichnet.

Aufgaben.

1. Entpacken Sie das Programm `binscan` (als Quellcode gegeben): `http://web.sepl.cs.uni-frankfurt.de/2013-ws/b-bkspp-pr/resources/binscan-0.1.0.tar.gz`
2. Übersetzen Sie das Programm, indem Sie im entpackten Verzeichnis folgende Befehle ausführen:
   (a) `./configure` (Dieser Befehl führt ein Shell-Skript aus, das die Konfiguration Ihres Systems überprüft)
   (b) `make` (Dieser Befehl übersetzt das Programm.)
3. Überprüfen Sie, daß Sie das erzeugte Programm `binscan` aufrufen können. Es ist ein Werkzeug, das in kleinen und mittelgroßen Dateien nach Byte-Folgen sucht. Folgender Befehl sucht z.B. nach allen Zeilenumbrüchen (Bytewert `0x0a`) in `binscan.c`:

   ```
   ./binscan -x0a binscan.c
   ```
4. Zwei andere Entwickler haben Ihnen zwei diffs gesendet. Das erste Diff korrigiert ein paar Probleme der Dokumentation, das zweite Diff erweitert das Programm so, daß es mehrere Muster gleichzeitig sucht:
   (a) `http://web.sepl.cs.uni-frankfurt.de/2013-ws/b-bkspp-pr/resources/documentation-update.diff`
   (b) `http://web.sepl.cs.uni-frankfurt.de/2013-ws/b-bkspp-pr/resources/multi-match.diff`

   Laden Sie beide Diffs auf Ihr System.
5. Verwenden Sie **patch(1)**, um den `binscan`-Quellcode mit einem der Diffs zu aktualisieren. Verwenden Sie dazu den (üblicheren) `patch`-Aufruf `patch <patchdatei`. Dazu müssen Sie u.U. zwei in der `man`-Seite angegebene Parameter verwenden:

- `-R` muß gesetzt werden, wenn das Diff 'falsch herum' berechnet wurde (also als Veränderung vom neuen zum alten Programm hin). Normalerweise kann **diff** dies aber selbst feststellen.
- `-p[n]`, wobei `[n]` eine nichtnegative Zahl ist. Dieser Parameter entfernt Elemente des Dateipfades aus dem Diff, bevor es angewendet wird. Wenn also eine Datei im Diff z.B. den Namen `foo/bar/x.c` hat, wird `-p 2` die Datei in `x.c` umbenennen.

Wenn **diff** eine Warnung ausgibt und sich weigert, das Diff ohne weitere Rücksprache anzuwenden, dann ist einer dieser Parameter nicht gesetzt. Wenn `-R` nicht gesetzt ist, fragt **diff** explizit danach. Wenn `-p` falsch gesetzt ist, erkennt man dies normalerweise daran, daß **diff** nach einer Datei fragt, die zwar existiert, aber weiter oben im Verzeichnisbaum als von **diff** postuliert.

6. Verwenden Sie **patch(1)**, um das zweite Diff anzuwenden. Untersuchen Sie sorgfältig die Ausgabe: es kann zu einem Konflikt kommen. In diesem Fall müssen Sie von Hand mit einem Editor eingreifen.

7. Stellen Sie sicher (durch erneuten Aufruf von `make` und Ausführen des Programmes), daß die gewünschten Änderungen funktionieren.

8. Implementieren Sie eine weitere Änderung: bauen Sie einen zusätzlichen Parameter (`-B` bzw. `-brief`) in das Programm ein, der dafür sorgt, daß nur die Adressen der gefundenen Bytefolgen ausgegeben werden, ohne eckige Klammern und Bytefolge. Statt

   ```
   [000000a2] ca fe
   ```

   soll also z.B. nur

   ```
   000000a2
   ```

   ausgegeben werden, aber nur, wenn der Parameter gesetzt ist. Das Programm verwendet bereits die Funktion **getopt_long(3)** um Parameter einzulesen; diese Funktion macht es Ihnen einfach, neue Parameter einzubauen, indem Sie die Konzepte aus `binscan` kopieren und anpassen.

9. Verwenden Sie **diff(1)**, um eine Differenz Ihres nun veränderten Programmes zum urpsrünglichen `binscan` (Version 0.1.0) zu berechnen. Verwenden Sie dazu die Parameter `-ru`.

Parameter `-u` ist der übliche Parameter, um ein sogenanntes *unifiziertes Diff* zu berechnen. Diese sind weniger fehleranfällig als Diffs, die ohne `-u` generiert werden, da sie sich zusätzlichen Kontext (vorhergehende bzw. folgende Programmzeilen) merken. In der Praxis bewährt sich die Parameterfolge `-ruN`.

3 Revisionskontrolle mit `git` (1 + 1B)

Um diese Aufgabe bearbeiten zu können, beachten Sie Abschnitt 0.5.

Sobald mehr als eine Person an einem Projekt arbeitet, stellt sich die Frage, wie Dateien und Veränderungen ausgetauscht und synchronisiert werden können. Die verbreitetste Lösung dazu sind *Revisionskontrollsysteme* (auch *Versionskontrollsysteme* genannt). Das Grundkonzept aller aktuellen Revisionssysteme ist gleich:

- Es existiert (mindestens) ein Verwahrungsort (*repository*).
- Alle Programmänderungen werden mit diesem *repository* synchronisiert.
- Das Revisionskontrollsystem verwendet ein Verfahren, um Konflikte bei der Synchronisierung zu erkennen und dem Benutzer zur Bearbeitung vorzulegen.

Revisionskontrollsysteme werden meist in zwei Kategorien unterteilt:

- Zentralisierte Revisionskontrollsysteme (RCS, CVS, Subversion, Perforce, ...), in denen genau ein zentraler Verwahrungsort existiert. Jegliche Kommunikation mit dem Revisionskontrollsystem findet (üblicherweise) über Netzwerkzugriff statt.
- Dezentrale Revisionskontrollsysteme (Mercurial, git, DARCS, ...) in denen beliebig viele Verwahrungsorte existieren können. Entwickler verwenden oft lokale Kopien des (bzw. eines der) Hauptverwahrungsorte, um Netzwerkzugriff zu vermeiden.

Dezentrale Revisionskontrollsysteme sind mächtiger und komplexer als zentralisierte. Wir verwenden in diesem Praktikum das dezentrale Revisionskontrollsystem **git**, beschränken uns aber auf dessen grundlegendste Operationen.

Die fundamentalen Komponenten von **git** sind folgende:

- *repository*: Ein Verwahrhungsort für Daten, die vom **git**-Werkzeugt verwaltet werden. Es dient als eine Datenbank für all die anderen Objekte, die unten angegeben sind. Repositories können untereinander abgeglichen werden; auf diese Weise werden Änderungen zwischen Enwicklern ausgetauscht.

  Ein Repository dient in **git** zwei Zwecken:

  1. Als Datenbank, wie beschrieben.
  2. Üblicherweise auch als 'Projektionsfläche' für gespeicherte Dateien (Abschnitt 3.1).

  Ein solches Repository ist an einem Unterverzeichnis namens `.git` erkennbar[3].

- *commit*: Ein bestimmter (historischer oder aktueller) Zustand des im Repository gespeicherten Projektes. Jeder Commit besteht aus:
  - Einem 40-Zeichen-Hexadezimal-Hashcode, der den Commit global eindeutig identifiziert (also auch zwischen Repositories)[4].
  - Einem Baum (*tree*, s.u.) mit den Inhalten des Commits; dieser muß nur selten direkt betrachtet werden.
  - Eltern: Jeder Commit kann einen oder mehrere „Elternteile" haben, aus denen Projektinhalte (Dateien, Verzeichnisse, ...) geerbt werden. Wir konzentrieren uns hier auf den Fall, in dem ein Commit höchstens einen Elternteil hat.

[3] Beachten Sie, daß dieses Verzeichnis normalerweise "unsichtbar" ist, also von den meisten Werkzeugen bei der Verzeichnisdarstellung unterdrückt wird. Um unsichtbare Dateien mit **ls** anzuzeigen, müssen Sie den Parameter `-a` verwenden, also `ls -a`.

[4] Beachten Sie, daß auch andere **git**-Konzepte wie *tree* und *blob* solche Hashcodes als Bezeichner verwenden.

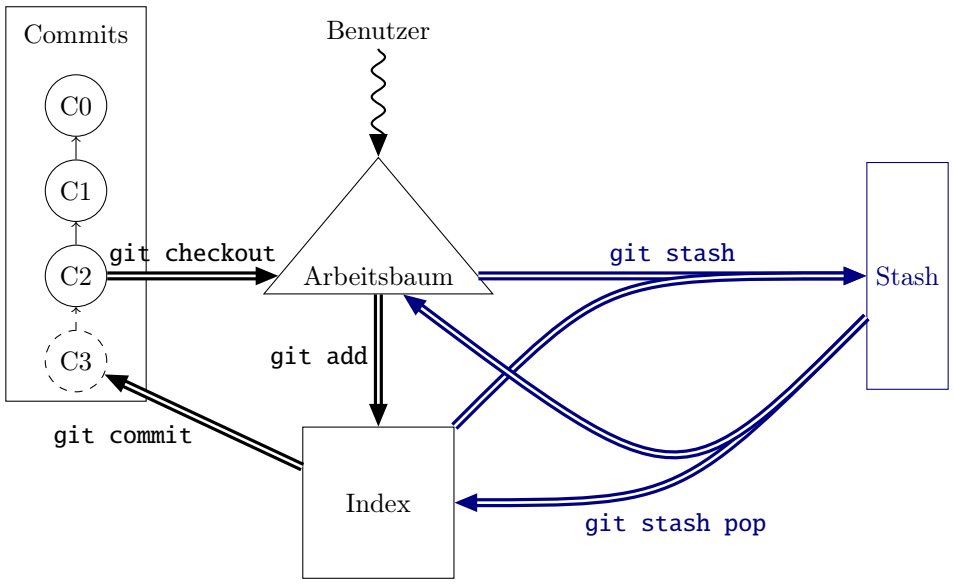

Abbildung 1: Schwarzer Teil: Zusammenspiel zwischen Index, Arbeitsbaum, und Commit-Geschichte beim Erstellen eines neuen Commits C3. Commit C0 ist hierbei der älteste (ursprüngliche) Commit, C1 und C2 sind nachfolgende Commits. C0 ist Eltern-Commit von C1, C1 Eltern-Commit von C2, und C2 wird Eltern-Commit von C3 werden. Um diesen neuen Commit C3 zu bauen, modifiziert der Benutzer den Arbeitsbaum, teilt die Änderungen dem Index per `git add` mit, und verwandelt schließlich alle aktiven Änderungen im Index in den neuen Commit C3, per `git commit`. Blauer Teil: Der (optionale) Stash, der zum Zwischenspeichern aller Veränderungen verwendet werden kann. `git stash` (bzw. `git stash save`) isoliert alle Änderungen, verschiebt sie in den Stash, und entfernt sie aus Arbeitsbaum und Index. `git stash pop` kehrt diesen Schritt um und läd also alle Änderungen zurück in Index und Arbeitsbaum. Der Stash funktioniert wie ein Stapel/Kellerautomat, kann sich also mehrere solche Änderungsblöcke merken.

- *tree*: Eine Baumstruktur, die aus Unterbäumen und *blobs* besteht. Das Konzept entspricht grob dem eines Verzeichnisses in der UNIX-Welt. Man muß diese Objekte meist nicht von Hand betrachten.
- *blob*: Eine Binär- oder Textdatei.

Ein Repository besteht aus folgendem:

- *commit*, *tree*, und *blob*-Objekten
- symbolischen Namen (*tags* und *branches*), die auf besonders 'interessante' Commits zeigen. Besonders wichtig dabei ist dabei der Name `HEAD` (Abschnitt 3.1).
- Einen Index oder *staging area*, der sich unfertige commits merkt
- Einen *stash*, auf den der Index verschoben werden kann (**git-stash(1)**).

Lesen Sie sich zunächst das Git-Tutorial (**gittutorial(7)**[5]) durch, um eine Übersicht darüber zu erhalten, wie diese Komponenten zusammenspielen. Dabei können Sie das Konzept der verschiedenen Zweige (*branches*) überspringen.

Beachten Sie insbesondere das Zusammenspiel zwischen Commits, Index und Arbeitsbaum (Abbildung 1). Der Arbeitsbaum sind all die Dateien, die in einem von Git verwalteten Verzeichnis liegen und von denen Git Kenntnis hat, mit Ausnahme des Verzeichnisses `.git` und dessen Inhalten, da dieses das Repository selbst enthält.

3.1 Der Arbeitsbaum als Projektionsfläche

Beachten Sie, daß Git den Arbeitsbaum zum 'Projizieren' der aktuellen Revision verwendet. Wenn Sie also ein Repository wie im Tutorial beschrieben klonen (**git-clone(1)**), werden dessen Dateien und Verzeichnisse im Arbeitsbaum dargestellt. Wenn Sie eine ältere Version oder einen anderen Zweig per **git-checkout(1)** laden, werden diese Dateien und Verzeichnisse überschrieben, sofern sie keine unerwarteten Veränderungen beinhalten oder Einträge im Index liegen. Ansonsten wird Git sie auf dieses Problem hinweisen.

Sollten Sie mit diesem Problem zu kämpfen haben, ist die einfachste Lösung meist, diese Änderungen auf den Stash zu verschieben (**git-stash(1)**). Dort können sie entweder verworfen oder zu einem späteren Zeitpunkt zurückgeholt werden.

3.2 Git-push

Zusätzlich zu den im Tutorial behandelten Befehlen interessiert uns hier der Befehl `push` (**git-push(1)**). Wenn Sie ein Repository mit **git-clone(1)** kopiert haben und eine Änderung per `git commit` festgeschrieben haben, können Sie diese per

```
git push origin master
```

zurück zum Ausgangsort (`origin`) in den Zweig `master` (der „Standardzweig“) schieben— sofern sich dort inzwischen keine anderen Änderungen angesammelt haben, sonst müssen Sie zunächst `git pull` ausführen und eventuelle Konflikte wie im Tutorial beschrieben behandeln (*merge*).

[5] Eine deutsche Übersetzung finden Sie hier: `http://web.sepl.cs.uni-frankfurt.de/2013-ws/b-bkspp-pr/resources/gittutorial.7` Verwenden Sie `man -l gittutorial.7`, um die Datei darzustellen.

Aufgaben

1. Stellen Sie sicher, daß der Veranstalter ein Git-Konto für Ihre Gruppe eingerichtet hat (Abschnitt 0.5).
2. Klonen Sie `git@web.sepl.cs.uni-frankfurt.de:${GRUPPENNAME}/1-3` in ein lokales Verzeichnis, wobei `${GRUPPENNAME}` der Ihnen zugewiesene Gruppenname ist. Dort finden Sie ein Programm mit mehreren Commits.
3. Finden Sie heraus, in welchem Commit die Datei `src/parser.y` entfernt wurde.
4. Klonen Sie `git@web.sepl.cs.uni-frankfurt.de:${GRUPPENNAME}/1-2` in ein anderes lokales Verzeichnis. Dieses Repository sollte leer sein.
5. Installieren Sie den Quellcode des Programmes `binscan` aus Aufgabe 2 in dieses Repository.
6. Koordinieren Sie sich mit Ihrem Partner: Jeder von Ihnen sollte einen separaten Klon des Repositories haben, der die Ursprungsfassung von `binscan-0.1.0` enthält.
7. Jeder von Ihnen installiert je einen der Patches aus Aufgabe 2.
8. Senden Sie Ihre Anderungen mit `git push` zurück an den Server.

   Beachten Sie, daß ein `git-push` abgelehnt werden kann, wenn Ihr lokales Repository veraltet ist. In diesem Fall müssen Sie Ihr Repository mit `git pull` aktualisieren.

   Früher oder später müssen Sie dazu Konfliktbehandlung durchführen, wie im Tutorial beschrieben. Notieren Sie alle Schritte, die Sie in dieser Teilaufgabe durchgeführt haben und erklären Sie zu jedem Schritt, was dieser Schritt bewirkte.

(Bonus) Gehen Sie zurück zu Projekt `1-3`. Erzeugen Sie einen neuen Zweig `deutsch`:

- Übersetzen Sie alle in C-Dateien erzeugten Ausgaben nach Deutsch (sie werden nicht nach der Qualität Ihrer Grammatik bewertet) und fügen Sie diesen dem Zweig `deutsch` hinzu, ohne den ursprünglichen Zweig zu verändern.
- Kopieren Sie all ihre Änderungen zurück auf den Server.

Literatur

- `http://web.sepl.cs.uni-frankfurt.de/2013-ws/b-bkspp-pr/resources/gittutorial.7`
- https://studi.f4.htw-berlin.de/www/help/tutorial/git/
- John Wiegley, "Git from the Bottom Up", `http://ftp.newartisans.com/pub/git.from.bottom.up.pdf`
- kernel.org, "Everyday GIT with 20 commands or so" `https://www.kernel.org/pub/software/scm/git/docs/everyday.html`
- Offizielle homepage: `http://www.git-scm.com`

3.3 Git für die verbleibenden Aufgaben

Für Ihr Team sind weitere git-Repositories eingerichtet, die Sie zum Austausch verwenden können. Insbesondere ist ein Projekt

```
git@web.sepl.cs.uni-frankfurt.de:${GRUPPENNAME}/test
```

zum Testen eingerichtet (wobei ${GRUPPENNAME} wieder der Ihnen zugewiesene Gruppenname ist), sowie mehrere Repositories

```
git@web.sepl.cs.uni-frankfurt.de:${GRUPPENNAME}/1-${n}
```

wobei `${n}` die Nummer der betreffenden Aufgabe ist. Beachten Sie, daß für einige Aufgaben bereits Daten in Ihrem Projektverzeichnis abgelegt sind; dies ist jeweils in der Aufgabe angegeben.

4 Der Debugger `gdb` ($2\frac{1}{2}$)

Lesen Sie zunächst die folgende Einführung in den GNU Debugger `gdb`:
`http://openbook.galileocomputing.de/linux_unix_programmierung/Kap17-005.htm`

In Ihrem Repository

```
git@web.sepl.cs.uni-frankfurt.de:${GRUPPENNAME}/1-4
```

ist eine Implementierung des Quicksort-Algorithmus angegeben, der mehrere Fehler hat.

Aufgaben

1. Verschaffen Sie sich Zugriff zu dem gegebenen Repository.
2. Untersuchen Sie, warum Quicksort nicht funktioniert, indem Sie das gegebene Programm mit gdb ausführen.
   - Das Programm hat mehrere Tests. Einige, aber nicht alle davon, schlagen fehl– Sie können diese als Startpunkt für Ihre Fehlersuche verwenden.
   - Beachten Sie, daß das Programm Speicherfehler enthalten und abstürzen kann.
3. Dokumentieren Sie genau, welche Fehler Sie wo gefunden haben.
4. Korrigieren Sie alle Fehler, die von den Tests gefunden wurden.

5 Unit-Testen mit **Check** ($2 + \frac{1}{2}$B)

Automatische Tests wie in der letzten Aufgabe sind eines der nützlichsten Werkzeuge zur Fehlersuche. Tests, die ein einziges, minimal großes Modul auf Korrektheit überprüfen, werden auch als *unit tests* bezeichnet. Solche Unit-Tests sind eines der wichtigsten Werkzeuge in der Qualitätssicherung der Softwareentwicklung.

Es existieren mehrere *unit test frameworks*, die das Schreiben und Verwenden von Unit-Tests vereinfachen. Wir werden hier mit der `check`[6]-Bibliothek arbeiten, die auch von dem in Aufgabe 3 verwendeten Projekt eingesetzt wird (dort können Sie sich inspirieren lassen).

Um die Check-Bibliothek zu verwenden, müssen Sie die betreffende Header-Datei einbinden:

```
#include <check.h>
```

Einen Test können Sie dann schreiben, indem Sie ihn in entsprechende Markierungen einschließen:

```
// Jeder Test beginnt wie folgt:
START_TEST(testname)
{
    ...
}
END_TEST
// Testende muss IMMER angegeben werden
```

Damit der Test weiß, ob er erfolgreich war oder nicht, können Sie Überprüfungen hinzufügen, wie z.B. die folgenden:

```
START_TEST(testname)
{
        ck_assert_int_eq(0, 0);          // int-Gleichheit
        ck_assert_int_lt(-1, 0);         // Kleiner-als
        ck_assert_int_le(-1, 0);         // Kleiner-oder-gleich
        ck_assert_int_gt(1, 0);          // Groesser-als
        ck_assert_int_ge(1, 0);          // Groesser-oder-gleich
        ck_assert_str_eq("foo", "foo");  // Stringgleichheit
        if (1 > 2) {
                fail("Mathematik beschaedigt!");
        }
}
END_TEST
```

Der Test `testname` überprüft also einige arithmetische Eigenschaften, die hoffentlich alle wahr sein sollten. Ein typisches Programm beinhaltet mehrere solche Tests. Diese werden dann in einer Test-Suite zusammengefaßt, die aus mehreren Testfällen (*test cases*) besteht. Check erlaubt mehrere Tests pro Testfall:

```
// Erzeuge eine Testsuite.  Ein Programm kann mehrere haben.
Suite *
test_suite()
{
        Suite *s = suite_create("Name der Testsuite");

        // Neue Unit-Tests werden an TCase-Objekte angehaengt
        TCase *tc = tcase_create("Testfaelle");

        // Man kann automatisch vor und nach den Tests bestimmte
        // Funktionen ausfuehren.  Mit folgendem Aufruf wird immer
```

[6] `http://check.sourceforge.net`

```
        // vorher die Funktion 'setup' und nachher 'teardown'
        // ausgefuehrt.  Dies ist nuetzlich, um das System in
        // einen bestimmten Zustand zu versetzen oder Ressourcen
        // nachher freizugeben.
        //
        //   tcase_add_checked_fixture(tc_core, setup, teardown);

        // Der Test 'testname' wird tc hinzugefuegt:
        tcase_add_test(tc, testname);
        // tcase_add_test(tc, testname_2); // falls wir noch einen Test haben
        // ... weitere Tests

        // Fuege das TCase-Objekt der Suite hinzu:
        suite_add_tcase(s, tc);

        return s;
}
```

Zu guter Letzt benötigen Sie nur noch einen Testtreiber, also ein Hauptprogramm, das die Test-Suite ausführt:

```
int
main (void)
{
        // Treiberprogramm
        int number_failed;
        Suite *s = test_suite();          // Erzeuge Test-Suite
        SRunner *sr = srunner_create(s);  // Bereite Lauf der Tests vor
        srunner_run_all(sr, CK_NORMAL);   // Fuehre Tests aus
        number_failed = srunner_ntests_failed(sr);
        srunner_free(sr);                 // Speicher freigeben
        // Programm gibt 0 zurueck (Erfolg), wenn alle Tests
        // erfolgreich waren
        return (number_failed == 0) ? 0 : 1;
}
```

Beachten Sie zum Übersetzen: Sie müssen dem Übersetzer den Parameter `-lcheck` angeben, um die Check-Bibliothek mit einzubinden; damit werden wir uns im nächsten Abschnitt beschäftigen.

Aufgaben.

1. Implementieren Sie Hashtabellen, die unten angegebenen Headerdatei entsprechen: `http://web.sepl.cs.uni-frankfurt.de/2013-ws/b-bkspp-pr/resources/chash.h`

   - Schreiben für diese Aufgabe zunächst alles in eine einzige `c`-Datei (mit Ausnahme der Header-Dateien).
   - Verwenden Sie früh und aggressiv Unit-Tests, um Fehler zu finden.
   - Die Größe der Hashtabelle ist immer eine Zweierpotenz. Sie können daher den Bit-Und-Operator(`&`) statt dem langsameren Modulus-Operator verwenden, um aus einem Hash-Wert die Position des korrespondierenden Eintrages zu berechnen.
   - Die Hashtabelle muß automatisch wachsen, wenn sie zu klein ist.
   - Die Hashtabellen sollen *open addressing* verwenden. Das heißt, daß im Falle einer Hash-Kollision mit einem bereits existierenden Schlüssel stattdessen nachfolgende Einträge der Hashtabelle gefüllt werden (und zwar insgsamt bis zu `OPEN_ADDRESS_LINEAR_LENGTH` viele). Entsprechend viele Einträge müssen ggf. auch bei der Suche nach einem Element betrachtet werden. Falls kein freier Platz existiert, muß die Tabelle wachsen.

- Beachten Sie, daß es nicht möglich ist, Elemente dieser Tabellen zu entfernen. (Dies ist eine Vereinfachung.)
- **Diese Hashtabellen werden später in anderen Aufgaben verwendet. Falls Sie diese Aufgabe nicht fertigstellen können, kontaktieren Sie den Veranstalter innerhalb der ersten zwei Wochen.**

(Bonus) Übersetzen Sie die Tests aus Aufgabe 4 in Check-Unit-Tests.

6 Objektdateien (1 + 1B)

Wir betrachten nun die inkrementelle Übersetzung von Programmen. C-Programme bestehen in der Praxis meist aus mehreren `.c` und `.h`-Dateien, wobei die `.c`-Dateien *separat* übersetzt werden (in Objektdateien, Endung `.o`). Diese `.o`-Dateien, die wir als separate Module verstehen können, werden danach mit dem sogenannten *Binder* zusammengefügt.

Aufgabe des Binders ist es, Referenzen auf Variablen und Funktionen (sogenannte *Symbole*) zwischen den Modulen zu klären– wenn `a.o` eine Funktion `f` definiert und `b.o` diese Funktion verwendet, kümmert sich der Binder darum, daß `b.o` weiß, wo diese Funktion definiert ist. Der Binder wiederum wird normalerweise vom Übersetzer `gcc` direkt aufgerufen.

Die Objektdateien können nach ihrer Übersetzung wie folgt gebunden werden:

- In eine ausführbare Datei (meist ohne Dateiendung)
- In eine *statische* Bibliothek (meist mit Endung `.a`)
- In eine *dynamische* Bibliothek (meist mit Endung `.so`)

Zum Hintergrund beachten Sie Foliensatz 4, Folien 8–35 (`http://sepl.cs.uni-frankfurt.de/2013-ss/b-sysp/folien-04.pdf`) (ohne die Assembler-spezifischen Teile), und Lesen Sie dazu Foliensatz 7, Folien 14–22 (`http://sepl.cs.uni-frankfurt.de/2013-ss/b-sysp/folien-07.pdf`).

Beachten Sie insbesondere Foliensatz 4, Folie 8, die verschiedene Sektionen der Objektdateien erklärt.

Wir werden uns nun mit einigen Befehlen beschäftigen, über die man diese Objektdateien und Bibliotheken untersuchen kann, und genauer betrachten, wie der Bindevorgang funktioniert. Dazu werden wir direkt den Binder **ld(1)** zum Binden verwenden, statt ihn (wie üblich) von **gcc(1)** aufrufen zu lassen.

Aufgaben

1. Installieren Sie zunächst das Programm `binscan`, `http://web.sepl.cs.uni-frankfurt.de/2013-ws/b-bkspp-pr/resources/binscan-0.1.0.tar.gz`. Rufen Sie `./configure` auf, aber nicht `make`. Das Programm `./configure` erzeugt ein `Makefile` (mehr dazu später) und eine Header-Datei namens `config.h`, die sich systemspezifische Konfigurationseigenschaften merkt.

2. Übersetzen Sie die vier im Projekt enthaltenen `.c`-Dateien per `gcc -DHAVE_CONFIG_H -I. -c` von Hand in Objektdateien. Das Präprozessorsymbol `HAVE_CONFIG_H` müssen Sie mit `-D` definieren, da der Quellcode es an einigen Stellen überprüft, um festzustellen, ob `config.h` verwendet werden sollte; ohne das Symbol wird das Programm falsche Annahmen über Ihr System machen. `-I.` teilt dem Präprozessor mit, wo es `config.h` (und andere per `#include <...>` angegebene Header-Dateien) finden kann.

3. Verwenden Sie **nm(1)**, um festzustellen, welche Symbole das Hilfsmodul `xmalloc.o` aus anderen Objektdateien und Modulen importiert. Geben Sie die Ausgabe von **nm** mit Kommentaren an.

4. Welche `xmalloc.o`-Definitionen werden in `binscan.o` verwendet? Erklären Sie anhand der Ausgabe von **nm**.

5. Versuchen Sie nun, die vier `.o`-Dateien mit **ld(1)** zu binden:

   ```
   ld *.o -o binscan
   ```

   Dies wird fehlschlagen, und Ihnen werden einige fehlende Symbole angezeigt werden. Schreiben Sie mindestens drei davon auf.

6. Die fehlenden Symbole stammen aus der Systembibliothek `libc`, die die meisten Standard-C-Funktionen zur Verfügung stellt. Wir können sie direkt per Namen zur Liste der Objektdateien hinzufügen, aber sie befindet sich auf verschiedenen Systemen in verschidenen Orten. Der Binder verwendet ein Konfigurationssystem, um sich zu merken, wo Bibliotheken liegen; wir werden diesem System **ldconfig(1)** hier vertrauen und den Parameter `-lc` verwenden. Dieser weist den Binder an, `libc.so` aus einem geeigneten Pfad einzubinden (ähnlich kann man z.B. auch `-lm` für `libm.so` einbinden, die Bibliothek für mathematische Hilfsfunktionen.) Diese Parameter können Sie so auch an `gcc` übergeben. Falls diese Pfade nicht ausreichen sollten, kann man **ld** und **gcc** per `-Lpfad/zur/Bibliothek` zusätzliche Pfade geben.

   Verwenden Sie `ld` wie zuvor, nun aber auch mit `-lc`. Welches Symbol fehlt noch?

7. Dieses fehlende Symbol wird von einer speziellen Objektdatei des C-Übersetzers hinzugefügt, die dieser stillschweigend mit bindet. Das Symbol wird vom Binder aufgerufen, und es stellt den notwendigen Systemzustand her, um `main` aufzurufen.

   Finden Sie die Dateien `crt1.o`, `crti.o`, und `crtn.o` in Ihrem System. Diese drei sollten alle im gleichen Verzeichnis liegen, meist `/usr/lib/`, `/usr/lib/x86_64-linux-gnu/`, `/usr/lib64`, oder `/usr/lib/lib64`. Sie können ggf. **locate(1)** zur Suche verwenden. Welche Symbole werden in den Dateien definiert?

8. Versuchen Sie, `crt1.o` (als erste Datei in der Parameterliste) an **ld** mit zu übergeben. Können Sie nun binden? Falls nicht, welches Symbol fehlt nun?

9. Korrigieren Sie das Problem (falls nötig) und geben Sie den vollständigen Aufruf an `ld` an. Sie haben nun wahrscheinlich alle die Komponenten gesehen, die der Binder benötigt, um Ihr Programm zu binden.

10. Alle statisch gebundenen Module (`.o`) und Bibliotheken (`.a`, hier nicht verwendet) sind nun direkt Teil Ihres Binärprogrammes. Dynamische Bibliotheken allerdings werden erst beim Laden des Programmes hinzugefügt. Geben Sie die Liste der dynamischen Abhangigkeiten Ihres Programmes aus, per **ldd(1)**[7].

(Bonus) Verwenden Sie **objdump(1)**, um die Inhalte der Objektdatei `xmalloc.o` auszugeben:

```
objdump -x xmalloc.o
```

Erklären Sie die Ausgabe in Ihren eigenen Worten. Sie können sich dabei auf die Folien, man-Seiten, und eigene Recherche beziehen.

[7] `linux-vdso.so.1` (oder auch `linux-gate.so.1` auf älteren Systemen) ist eine 'virtuelle Bibliothek', die angibt, auf welche Weise Systemaufrufe (Kerneleinsprünge) am Effizientesten durchgeführt werden können– dies unterscheidet sich auf der Intel-Architektur, da neuere Prozessoren einen effizienteren Mechanismus dafür verwenden.

7 Bibliotheken (1)

Wir werden nun Bibiliotheken bauen und verwenden. Beachten Sie dazu zunächst die vorhergehende Aufgabe, und lesen Sie dann
`http://openbook.galileocomputing.de/linux_unix_programmierung/Kap17-001.htm`.

Aufgaben.

1. Nehmen Sie Ihre Hashtabelle aus Aufgabe 5 und trennen Sie Unit-Tests und Hashtable-Implementierung in separate Module.

2. Bauen Sie eine *statische* Bibliothek der Hashtabelle. Übersetzen Sie die Unit-Tests separat, und binden Sie beides zusammen. Geben Sie alle verwendeten Befehle an.

3. Bauen Sie eine *dynamische* Bibliothek der Hashtabelle. Übersetzen Sie die Unit-Tests separat, und binden Sie beides zusammen. Geben Sie alle verwendeten Befehle an.

8 Build-Systeme am Beispiel von **cmake** ($3\frac{1}{2} + 1\frac{1}{2}$B)

Es kann sehr umständlich sein, alle zum Übersetzen eines Programmes nötigen Befehle von Hand einzugeben. Im Laufe der Zeit haben sich drei Strategien herausgebildet, um dieses Problem anzugehen:

- *Shell-Skripte*, die sich den Übersetzunsprozeß merken: Mit etwas Vorsicht können diese Skripte modular aufgebaut werden und somit effektiv funktionieren. Ein großes Problem dieser Skripte ist aber, daß es ihnen oft nicht leicht fällt, *inkrementelle Übersetzung* zu realisieren, also Übersetzung auf die Teile eines Programmes zu beschränken, die sich auch verändert haben.
- *IDEs* bieten oft automatische Mechanismen an, um Projekte zu übersetzen. Die Projektformate sind aber oft zwischen den IDEs sehr unterschiedlich. Da Entwickler sich zwar oft auf ein Betriebssystem, aber selten auf eine IDE einigen können, hat sich dieses Verfahren nur in wenigen Bereichen durchgesetzt.
- *Build-Systeme* sind spezialisierte Systeme, die *Abhängigkeiten* und *Übersetzungsregeln* verstehen und daraus die nötigen Übersetzungsschritte synthetisieren können. Die verbreitetsten Build-Systeme sind *Makefiles* und *ant* (für Java).

Für C sind Makefiles, unterstützt durch das Programm **make(1)**, die verbreitetste Lösung. In Makefiles konfiguriert man die notwendigen Übersetzungsschritte als Abhängigkeiten ('`binscan` benötigt `xmalloc.o`') und Übersetzungsregeln ('um x`.c` nach x`.o` zu übersetzen, rufe `gcc -c` x`.c` auf'); das System kann dan auf Befehle wie

`make binscan`

reagieren, und alle nötigen (und auch nur die nötigen!) Übersetzungsschritte auslösen, um die Datei `binscan` zu erzeugen oder nach Änderungen der Quelldateien auf den neuesten Stand zu bringen.

Eine Einführung zu **make(1)** finden Sie hier (optional): `http://openbook.galileocomputing.de/linux_unix_programmierung/Kap17-000.htm`

Makefiles sind in der Praxis aber keine vollständige Lösung aller zur Übersetzung relevanten Probleme; so gehen sie z.B. nicht auf Konfigurationsmanagement ein und integrieren keine Informationen über das zugrundeliegende System. Diese beiden Konzepte sind aber wichtig für portable Programmierung, da unterschiedliche Plattformen oft im Detail unterschiedliche Übersetzungsschritte benötigen. Stattdessen werden Build-Systeme wie `automake`/`autoconf` oder `cmake` vorangestellt, die selbst Makefiles erzeugen. Wir betrachten hier spezifisch CMake, ein inzwischen sehr verbreitetes Build-System, das zwar sicher nicht die letzte Antwort auf dieses Problem ist, aber den aktuellen Stand der Technologie gut darstellt.

CMake[8] nimmt Dateien mit dem Namen `CMakeLists.txt` und übersetzt sie in Makefiles. Diese Makefiles überprüfen die Systemkonfiguration (nur beim ersten Aufruf), schreiben diese Konfiguration in eine Datei `CMakeCache.txt`, und führen dann einen Übersetzungsprozeß aus, der den eingestellten Anforderungen entspricht. Wenn sich die Systemkonfiguration ändert, kann man CMake durch Löschen von `CMakeCache.txt` dazu zwingen, diese neu festzustellen.

Zum Vergleich: die *autotools* (`automake`/`autoconf`), die wir hier nicht verwenden, basieren auf mehreren Konfigurationsdateien. Sie überprüfen die Konfiguration des Systems in einem `configure`-Skript, das wiederholt ausgeführt werden kann und die Makefiles erzeugt. Viele Projekte verwenden die Autotools, so daß Ihnen dieser Prozeß öfters begegnen könnte.

Betrachten wir nun eine Datei `CMakeLists.txt`:

```
# Projektname
project(projektname)

# Aktiviere C99 (Vorsicht: funktioniert nur für gcc)
# Aktiviere POSIX-Unterstützung (Version bis September 2008)
```

[8] `http://www.cmake.org`

```
# Aktiviert alle Warnungen des Übersetzers
set(CMAKE_C_FLAGS "-std=c99 -D_POSIX_C_SOURCE=200809 -Wall")

# Unterverzeichnisse, die cmake verwalten soll (Mehrfachnennungen möglich)
add_subdirectory(src)
```

Kommentare in CMake werden durch einen vorgestellten Lattenzaun (#) eingeleitet. Der Rest besteht aus Deklarationen der Form

Prädikat `(` *Argumente* `)`

wobei *Prädikat* ein Wort ist und *Argumente* eine Folge von Argumenten, üblicherweise Worte oder Zeichenketten (mit Anführungszeichen). Einzelne Argumente werden durch Leerzeichen getrennt.

Viele Projekte legen ihren Quellcode in einem Unterverzeichnis wie z.B. `src/` ab. Wir drücken dies in der obigen Datei durch

`add_subdirectory(src)`

aus; diese Deklaration kündigt an, daß in `src/` noch eine weitere `CMakeLists.txt` existiert, die rekursiv eingebettet wird.

Der Artikel `http://www.linux-magazin.de/Ausgaben/2007/02/Mal-ausspannen` beschreibt schon auf den ersten drei Seiten die wichtigsten Aspekte von CMake.

Das Projekt aus Abschnitt 3 verwendet ebenfalls bereits `cmake`. Sie können die dortigen Angaben als Vorlage nehmen.

8.1 Autorenidentifizierung als CMake-Projekt

Im folgenden schreiben Sie ein CMake-basiertes Projekt, das Ihre in Aufgabe 7 in zwei Teile geteilten Hashtabellen verwendet. Ziel des Projektes ist *Autorenidentifizierung*: Wir haben vier Bücher unbekannter Autoren, und wollen über statistische Methoden herausfinden, wer von drei Autoren (Goethe, Schiller, Grillparzer) der wahrscheinlichste Autor jedes dieser Werke ist. Dazu führen wir zwei Schritte durch:

- Wir erstellen ein *Modell* der Schreibweise jedes Autors. Vereinfachend betrachten wir dabei nur die relative Häufigkeit der vom Autor verwendeten Worte, oder genauer der Wort*stämme*: dies stellt sicher, daß z.B. *Tisch* und *Tische* gleichwertig gezählt werden. Wir verwenden eine Bibliothek, um diese Wortstämme zu berechnen.

  Dies könnte z.B. ergeben:

goethe	tisch	0.0002
	seh	0.0003
	ich	0.0001
schiller	tisch	0.0001
	seh	0.0002
	ich	0.0001

  Das heißt, z.B., daß die Wahrscheinlichkeit, daß ein beliebiges von Goethe verwendetes Wort den Wortstamm *Tisch* hat, bei 0,002 liegt:

  $$P(\text{Goethe}|\text{Tisch}) = \frac{\text{Anzahl Vorkommnisse von „Tisch“ in Goethes Büchern}}{\text{Anzahl aller Worte in Goethes Büchern}} = 0{,}002$$

- Wir berechnen dann die Wahrscheinlichkeit, daß ein bestimmter Autor eines der unbekannten Werke geschrieben hat, indem wir die Wahrscheinlichkeiten ausrechnen, daß der Autor die betreffenden Wörter verwendet hat. Den Satz

  „Ich sehe Tische“

  würden wir z.B. mit einer Wahrscheinlichkeit von

$$P(\text{Goethe}|\text{Ich sehe Tische}) = 0.0002 \times 0.0003 \times 0.0001 = 6 \times 10^{-12}$$

Goethe zuschreiben, und Schiller mit

$$P(\text{Schiller}|\text{Ich sehe Tische}) = 0.0001 \times 0.0002 \times 0.0001 = 2 \times 10^{-12}$$

Folglich ist Goethe nach diesem Modell der wahrscheinlichere Autor[9].

Das Modell ist nachher eine Liste von Paaren von Wortstammen mit deren Wahrscheinlichkeit. Da diese Wahrscheinlichkeiten sehr klein werden können, besteht das Risiko eines Fließkommazahlenunterlaufs. Wir verwenden daher einen in der künstlichen Intelligenz weit verbreiteten Trick: Anstatt Wahrscheinlichkeiten zu speichern und miteinander zu multiplizieren, speichern wir *logarithmische* Wahrscheinlichkeiten und *addieren* diese, da

$$\exp^{\ln a + \ln b} = a \times b$$

Aufgaben.

1. Laden Sie eine Büchersammlung herunter:
   `http://web.sepl.cs.uni-frankfurt.de/2013-ws/b-bkspp-pr/resources/buecher.tar.gz`
   In diesem Archiv finden sich je 10 Bücher von Goethe, Schiller, und Grillparzer, sowie vier unbekannte Bücher `A`, `B`, `C`, `D`.

2. Schreiben Sie ein Programm, das alle auf der Kommandozeile angegebenen Dateien nacheinander einliest, indem Sie **open(2)** und **close(2)** verwenden. Zum Lesen können Sie **read(2)** verwenden; die Größe der Datei können Sie über **fstat(2)** feststellen.

3. Schreiben Sie passende `CMakeLists.txt`-Dateien, verwenden Sie '`cmake .`' zum Erstellen der Makefiles, und `make` zum Übersetzen. Verwenden Sie diese auch im Rest dieser Aufgabe.

4. Iterieren Sie über das eingelesene Programm, finden Sie alle Wörter, und wandeln Sie diese in Kleinbuchstaben um. Schreiben Sie nach Möglichkeit einen Unit-Test hierzu.

   Beachten Sie dabei, daß C keine gute Unterstützung für Umlaute hat. Zur Vereinfachung arbeiten wir nicht in Unicode, sondern in einem verbreiteten Kodierungsformat, das die Zahlen zwischen 128 und 255 verwendet, um Umlaute zu kodieren. **iso-8859-1(7)** beschreibt diese Kodierung; die unteren 128 Zeichen sind wie sonst bei ASCII üblich belegt. Alle der angegebenen Bücher sind in diesem Format kodiert.

5. Schreiben Sie passende Unit-Tests und aktualisieren Sie diese, um Fehler so früh wie möglich zu finden. Sie können beliebige weitere Werkzeuge verwenden, um Fehler zu suchen.

6. Integrieren Sie die folgende Bibliothek:
   `http://snowball.tartarus.org/dist/libstemmer_c.tgz` Diese Bibliothek kann Wortstämme zu berechnen. Die wichtigste API wird in `include/libstemmer.h` beschrieben. Beim Übersetzen der Bibliothek wird eine Datei libstemmer.o erzeugt, die tatsächlich eine `.a`-Datei ist.

   Den für uns relevanten 'Stemmer' (Wortstammberechner) erhalten Sie mit dem folgenden Aufruf: `sb_stemmer_new("de", ÏSO_8859_1")`.

7. Zählen Sie die Anzahl der Vorkomnisse aller Wortstämme mit Ihren Hashtabellen.

[9] Dieses Verfahren ist für kurze Sätze natürlich ungeeignet, aber bei hinreichend großen Texten selbst in seiner naivsten Fassung oft erfolgreich.

8. Berechnen Sie die Wortwahrscheinlichkeiten als **log(3)**-Wahrscheinlichkeiten. Dazu müssen Sie die mathematische Hilfsbibliothek `libm` einbinden. Schreiben Sie die Wahrscheinlichkeiten per **printf(3)** aus.

9. Leiten Sie die Ausgabe des Programmes in eine Textdatei um, die Sie als Modell verwenden.

10. Schreiben Sie ein separates Programm, das mehrere Modelle (**fscanf(3)** ist wahrscheinlich die einfachste Lösung dazu) und ein Buch als Textdatei einliest und den wahrscheinlichsten Autor bestimmt.

11. Verifizieren Sie, daß Ihr Programm die Autoren der 30 bereits identifizierten Bücher korrekt erkennt. Identifizieren Sie dann die vier unbekannten Bücher.

(Bonus) Importieren Sie die Unit-Tests in CMake, indem Sie CTest verwenden (Ähnlich dem Projekt aus Abschnitt 3). Sie können dann mit `cmake test` oder `ctest` Tests ausführen. Dies liefert vereinfachte Fehlerausgabe; mit `ctest -V` können Sie detailliertere Fehlerausgabe erhalten.

(Bonus) Verwenden Sie (im Vorgriff auf Teil 2) statt **read(2)** die Operationen **mmap(2)** und **munmap(2)**.

9 Statische Fehlersuche ($\frac{1}{2}$ + $\frac{1}{2}$B)

Statische Fehlersuche[10] ist eine Strategie, in der der Quellcode eines Programmes analysiert und gezielt nach Programmierfehlern durchsucht wird. Teilweise können Sie solche Überprüfungen direkt mit `gcc` durchführen, wenn Sie den Parameter `-Wall` angeben, aber es existieren statische Fehlersucheprogramme, die noch weitere Fehlermuster entdecken können.

Installieren Sie das Programm `cppcheck`[11]. Laden Sie dazu den Quellcode (verlinkt auf der Seite) `http://sourceforge.net/projects/cppcheck/` herunter und führen Sie `make` aus. Sie erhalten ein ausführbares Programm `cppcheck`: ein statisches Fehlersuchprogramm.

Aufgaben.

1. Untersuchen Sie die folgenden Dateien mit `cppcheck`. Beschreiben Sie die gefundenen Fehler und geben Sie an, ob bzw. warum diese Fehler echte Fehler sind und ggf. wie sie behoben werden können.

   - `http://web.sepl.cs.uni-frankfurt.de/2013-ws/b-bkspp-pr/resources/vms_directory.c` (Aus dem Programm `xpdf`)
   - `http://web.sepl.cs.uni-frankfurt.de/2013-ws/b-bkspp-pr/resources/reader.c` (Aus dem Programm `network-manager`; Vorsicht: `goto`!)
   - `http://web.sepl.cs.uni-frankfurt.de/2013-ws/b-bkspp-pr/resources/ux_audio_oss.c` (Aus dem Programm `frotz`)

(Bonus) Wenden Sie `cppcheck` auf `binscan.c` (aus `binscan-0.1.0.tar.gz`) an. Verwenden Sie nun das vorinstallierte Programm `splint` auf `binscan`. Überprüfen Sie je 20 Meldungen jedes der beiden Programme (sofern genug Meldungen erschienen, sonst alle erschienenen Meldungen). Korrigieren Sie das Programm entweder entsprechend, oder geben Sie an, warum Sie die Meldung nicht hilfreich fanden.

[10] traditionell durch das Programm `lint`, so z.B. auch **splint**, daher auch manchmal als *linting* bezeichnet
[11] `http://cppcheck.sourceforge.net/`

10 Speicherfehlersuche mit `valgrind` ($\frac{1}{2} + \frac{1}{2}$B)

Das Komplement zur statischern Fehlersuche ist dynamische Fehlersuche, also Fehlersuche durch Beobachtung des Laufzeitverhaltens eines Programmes. Das verbreitetste Beispiel dazu sind Unit-Tests, aber es existieren spezielle Werkzeuge, die zusätzliche Fehler entdecken können.

Statische und dynamische Fehlersuche ergänzen sich gegenseitig. Ihre relativen Vorteile sind wie folgt:

- *Statische Fehlersuche*:
  - Kann das komplette Programm betrachten, nicht nur die Teile, die in einem bestimmten Durchlauf ausgeführt wurden
  - Benötigt keine Testläufe oder Benutzerinteraktion zur Fehlersuche
- *Dynamische Fehlersuche*:
  - Beobachtet immer 'echte' Fehler (statische Fehlersuche findet gelegentlich 'falsche' Fehler, *false positives*)
  - Kann genaue Beispiele (stack trace, Speicheradressen) zu gefundenen Fehlern angeben

Effektive Softwareprojekte verwenden beide Strategien.

Wir betrachten hier **valgrind**, ein besonders nützliches Programm zur dynamischen Fehlersuche für nicht typsichere Sprachen wie C und C++.

Lesen Sie zunächst folgendes:
`http://openbook.galileocomputing.de/linux_unix_programmierung/Kap17-007.htm`

Beachten Sie, daß die Abhandlungen zu `libefence` für dieses Praktikum nicht relevant sind.

Falls Sie Universitätsrechner verwenden, sollte **valgrind(1)** bereits installiert sein.

Aufgaben.

1. Finden und korrigieren Sie einen Speicherfehler in `binscan-0.1.0` (ohne angewendete Diffs) aus Aufgabe 2.
2. Finden und korrigieren Sie einen Speicherfehler in dem zusammengefügten `binscan` (mit allen Diffs) aus Aufgabe 2.

(Bonus) Wenden Sie Valgrind auf Ihren Modell-Bauer und Klassifizierer aus Aufgabe 8 an. Geben Sie alle Ausgaben an und korrigieren Sie alle Speicherfehler. Beschreiben Sie, welche Speicherfehler sie entdeckt haben und wie Sie diese korrigiert haben.

11 Grundlagen der Performanzanalyse (1 + 1B)

Ein wichtiger Grund für den Einsatz von C und C++ in der Systementwicklung ist das Erzielen hoher Verarbeitungsgeschwindigkeiten. Um diese zu erreichen, ist es oft notwendig, die Performanz verschiedene Designalternativen miteinander zu vergleichen. Betrachten Sie dazu zunächst Foliensatz 2, Folie 3 (`http://sepl.cs.uni-frankfurt.de/2013-ss/b-sysp/folien-02.pdf`) und Foliensatz 8, Folien 33–39 (`http://sepl.cs.uni-frankfurt.de/2013-ss/b-sysp/folien-08.pdf`).

Aufgaben.

1. Berechnen Sie eine Annäherung an die Zeit, die nötig ist, um einen Eintrag in einer Ihrer Hashtabellen zu finden. Verwenden Sie als Schlüssel zufällig generierte Zeichenketten der Länge 7, und tragen Sie 32 unterschiedliche(!) Schlüssel in die Tabelle ein. Verwenden Sie **gettimeofday(2)** oder **clock_gettime(2)**, um die Zeit zu messen.

   Beachten Sie, daß das Messen oder auch das Generieren von Zufallszahlen ebenfalls Zeit benötigt, und daß die Zeiten für den Hashtabellenzugriff im Verhältnis sehr klein sein können. Daher ist es sinnvoll, wenn Sie mehrere Zugriffe in einer Schleife durchführen, und nur zu Beginn und Ende der Schleife die Zeit messen. Um allerdings zufällige Faktoren zu entdecken, müssen Sie diese Schleifenmessung wiederholt durchführen.

2. Geben Sie zu Ihren Messungen zumindest die drei Hauptkennzahlen der Kastendarstellung an: Median und Beginn/Ende der beiden inneren Quartile. Die gemessenen Zahlen sollen sich auf die (geschätzte) Einzelzugriffszeit auf die Tabelle beziehen.

3. Messen Sie, wie lange der Zugriff im Schnitt dauert, wenn Sie 100% erfolgreiche und 100% nicht erfolgreiche Nachfragen haben. Geben Sie die Zeiten dafür an.

4. Was sind Ihre Messungen, wenn Ihre Nachfragen 50% der Zeit erfolgreich sind?

5. Führen Sie die gleichen drei Tests (0%, 50%, 100% erfolgreich) durch, aber mit 65536 unterschiedlichen Schlüsseln, die in der Tabelle gespeichert sind.

(Bonus) Geben Sie zusätzlich zu den drei Kennzahlen die komplette Kastendarstellung (inklusive Ausreissern (*outliers*)) an, als Text oder graphisch.

(Bonus) Variieren Sie `OPEN_ADDRESS_LINEAR_LENGTH`. Welcher Wert gibt Ihnen die beste Performanz in den beiden oben genannten Größen bei 50% Erfolgsquote?

12 Profiling mit `gprof` ((1))

Profiling ist die gezielte Suche nach einem Modell, das die Wichtigkeit der verschiedenen Komponenten eines Systems für dessen Gesamtlaufzeit charakterisiert. Es wird insbesondere verwendet, um gezielt nach den lohnendsten Optimierungsmöglichkeiten zu suchen.

Lesen Sie zunächst folgendes:
`http://openbook.galileocomputing.de/linux_unix_programmierung/Kap17-004.htm`
Beachten Sie, daß die dort auch behandelte Coverage-Analyse in diesem Praktikum nicht verwendet wird.

Aufgabe.

1. Verwenden Sie `gprof`, um zu identifizieren, welche Funktion die meiste Zeit in Ihrem Klassifizierungsprogramm (Aufgabe 8) benötigt. Geben Sie alle relevanten Ausgaben an.

13 Zusammenfassung (1 + $\frac{1}{2}$B)

Aufgabe.

1. Führen Sie alle in diesem Praktikumsteil aufgeführten Werkzeuge auf und beschreiben Sie in ihren eigenen Worten deren Zweck.

(Bonus) Beschreiben Sie unterschiedliche Schwächen und Nachteile der verwendeten Werkzeuge anhand von Beispielen aus der Bearbeitung. Geben Sie insgesamt mindestens drei solche Fälle an.

14 Literatur

- "The C Programming Language" (Brian W. Kernighan, Dennis M. Ritchie) (auch auf Deutsch in der Bibliothek)
- C99–Spezifikation (draft) `http://sepl.cs.uni-frankfurt.de/2013-ss/b-sysp/C99.pdf`
- "The Linux Programming Interface: A Linux and UNIX System Programming Handbook" (Michael Kerrisk)
- „Linux– Unix- Systemprogrammierung“ (Helmut Herold)
- Foliensätze 02-08 der Veranstaltung B-SYSP vom Sommersemester 2013 `http://sepl.cs.uni-frankfurt.de/2013-ss/b-sysp/veranstaltungen.de.html`