U2 2. Übung

- Nachtrag zu Subversion
- SP-Abgabesystem: Team-Arbeit
- Dynamische Speicherverwaltung
- Fehler finden mit valgrind
- Aufgabe 1: lilo

U2-1 Nachtrag zu Subversion

1 Versionierungsschema

- Subversion nummeriert fortlaufend ab Revision 0 (1,2,3,...)
- spezielle Revisionsschlüsselwörter
  - HEAD: aktuelle Version des Repositories (neueste Version)
  - BASE: Revision eines Eintrags (Datei, Verzeichnis) der Arbeitskopie
  - COMMITTED: Letzte Änderungsrevision eines Eintrags älter als BASE
  - PREV: COMMITTED-1
- Revision zu einem bestimmten Zeitpunkt
  - {"2010-10-19 14:42"}

2 Kommentar bei commit

- Beim Aufruf von `svn commit` öffnet sich normalerweise ein Editor zum Eingeben eines Commit-Kommentars
  - Im CIP ist dieses der Editor `joe`
    - Zum Speichern und Verlassen des Editors `Strg-k x` drücken
    - Hilfemenü öffnet sich mit `Strg-k h`
  - Anderer Editor kann über die Umgebungsvariable `EDITOR` ausgewählt werden

```
alice@faui01[~]$ export EDITOR=vim
```

    - Umgebungsvariable ist nur in dieser Shell-Sitzung gültig
    - Durch Eintragen des Kommandos in die Konfigurationsdatei der eigenen Shell (z.B. `~/.bashrc`) wird der Standardeditor für jede neue Shell geändert
- Übergabe des Kommentars als Argument von `svn commit`:

```
alice@faui01[/proj/i4sp1/alice]$ svn commit -m "Ich schreibe
lieber gleich in die Befehlszeile und nicht in den Editor"
```


2 Exkurs: Quoting von Zeichen mit Sonderbedeutung

- Sonderzeichen (wie **<**, **>**, **&**, Leerzeichen) soll als Argument übergeben werden:

```
cd Eigene Dateien
```

- Problem: die Shell interpretiert diese Zeichen
  - im Beispiel das Leerzeichen als Trenner mehrerer Argumente
  - das Kommando wird mit zwei Argumenten ausgeführt: *Eigene* und *Dateien*
- Lösung: Quoting nimmt Zeichen die Sonderbedeutung:
  - Voranstellen von `\` nimmt genau einem Zeichen die Sonderbedeutung
    `\` selbst wird durch `\\` eingegeben
  - Klammern des gesamten Arguments durch `" "`,
    `"` selbst wird durch `\"` angegeben
  - Klammern des gesamten Arguments durch `' '`

```
cd Eigene\ Dateien
cd "Eigene Dateien"
cd 'Eigene Dateien'
```

3 Basisoperationen

- diff: Lokale Änderungen der Arbeitskopie anzeigen

```
alice@faui01[/proj/i4sp1/alice/trunk]$ svn status
M       hallo
alice@faui01[/proj/i4sp1/alice/trunk]$ svn diff
Index: hallo
===================================================================
--- hallo       (revision 23)
+++ hallo       (working copy)
@@ -0,0 +1 @@
+welt
```

- revert: Änderungen an Arbeitskopie zurücksetzen

```
alice@faui01[/proj/i4sp1/alice/trunk]$ svn revert hallo
Reverted 'hallo'
alice@faui01[/proj/i4sp1/alice/trunk]$ svn status
alice@faui01[/proj/i4sp1/alice/trunk]$
```


3 Basisoperationen

- list/ls: Dateien/Verzeichnisse im Repository anzeigen

```
alice@faui01[/proj/i4sp1/alice]$ svn ls
branches/
trunk/
```

- log: Historie anzeigen

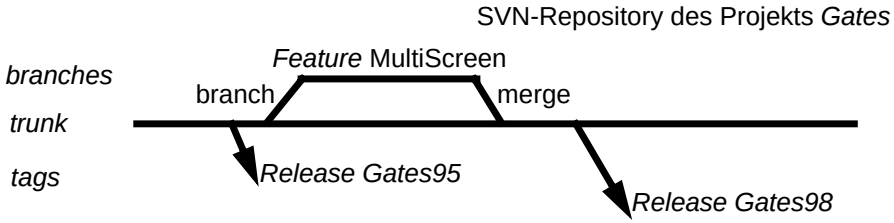
```
alice@faui01[/proj/i4sp1/alice]$ svn log
------------------------------------------------------------------------
r1 | www-data | 2010-04-20 15:03:14 +0200 (Tue, 20 Apr 2010) | 1 line

init repository
------------------------------------------------------------------------
```

- move/mv: Datei umbenennen oder verschieben
- copy/cp: Datei/Teilbaum kopieren

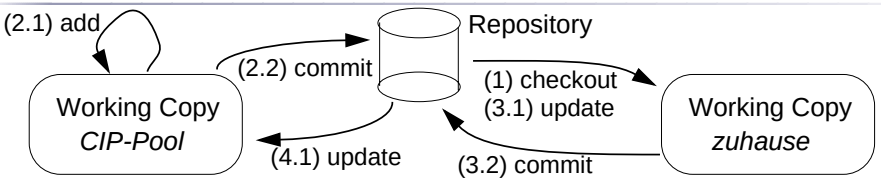
```
alice@faui01[trunk]$ svn cp aufgabe2 contest
# aufgabe2 wurde in contest kopiert
```


4 Konventionelles Repository-Layout



- Unterteilung des Wurzelverzeichnisses
  - Hauptentwicklungslinie: *trunk*
  - Verzeichnis mit Entwicklungszweigen: *branches*
  - Eingefrorene Versionen: *tags*
- Größere Features können von der Hauptlinie entkoppelt in einem eigenen Zweig (*branch*) entwickelt werden und nach Fertigstellung wieder in die Hauptlinie eingebracht (*merge*) werden
- Besondere Versionen können benamt (*getaggt*) werden (z.B. Release)

5 Arbeiten zuhause



(1) Zusätzliche Arbeitskopie(n) erstellen (*checkout,* einmalig)

(2) Start der Arbeit an einer Aufgabe im CIP-Pool
  - angelegte Dateien und Verzeichnisse unter Versionskontrolle stellen (*add*)
  - Zwischenstand ins Repository einchecken (*commit*)

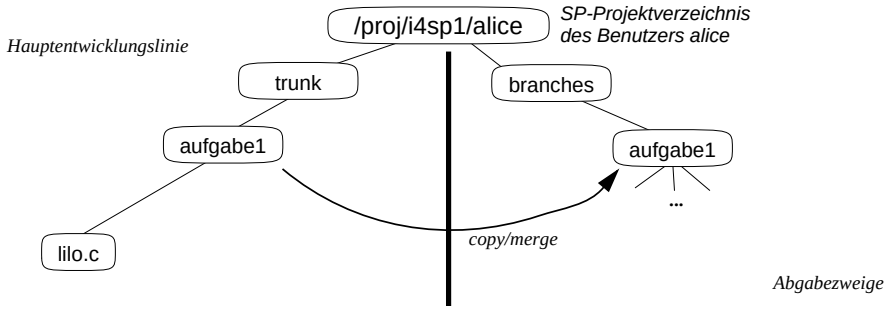
(3) Arbeit zuhause fortsetzen
  - Arbeitskopie zunächst auf den aktuellen Stand bringen (*update*)
  - Zwischenstand ins Repository einchecken (*commit*)

(4) Arbeit im CIP-Pool fortsetzen
  - Arbeitskopie zunächst auf den aktuellen Stand bringen (*update*)
  - ...

U2-2 Abgabesystem: Team-Arbeit

- Gemeinsame Bearbeitung im Repository eines Teammitglieds
  - Repository-Eigentümer: *alice*
  - Partner (nutzt Repository von *alice*): *bob*
- Abgabe erfolgt ebenfalls im Repository des Eigentümers
  - es ist nur eine Abgabe erforderlich
- Machen Sie sich frühzeitig mit dem Bearbeitungs-/Abgabeprozess vertraut
  - Arbeiten Sie von Beginn an in Ihrem Projektverzeichnis
  - Checken Sie auch Zwischenstände Ihrer Bearbeitung in das Repository ein
  - Sie können zu Beginn auch leere Dateien einchecken und abgeben
  - selbstverschuldet verspätete Abgaben werden nicht angenommen!
- **Hinweis**: bei Verständnis-Problemen zu Subversion empfiehlt sich die Lektüre zumindest der ersten beiden Kapitel des SVN-Buchs
  - **`http://svnbook.red-bean.com/`**

1 Funktionsweise der SP-Abgabe



- Zur Abgabe wird ein Abgabezweig für jede Aufgabe in *branches* erzeugt
  - unterhalb von *branches* sollte nicht von Hand editiert/eingecheckt werden
- Eine erfolgreiche Abgabe ist durch Aufruf der folgenden URL erkennbar:
  `https://www4.informatik.uni-erlangen.de/i4sp/ws11/sp1/<login>/branches/<aufgabe>/`
  - Dort werden die Dateien angezeigt, die zur Bewertung herangezogen werden

2 Ablauf für den Repository-Eigentümer

- Der Partner wird für jede Team-Aufgabe separat festgelegt

```
alice@faui01$ /proj/i4sp1/bin/set-partner aufgabe1 bob
```

- Hintergrund
  - Erzeugung und Commit einer Textdatei `partner` in `trunk/aufgabe1`
  - diese Datei enthält den Login-Namen (*bob*) des Partners für diese Aufgabe
  - Partner erhält Zugriff auf die relevanten Teile des Repositorys
    - `trunk/aufgabe1`
    - `branches/aufgabe1`
- Abgabe funktioniert wie gewohnt:

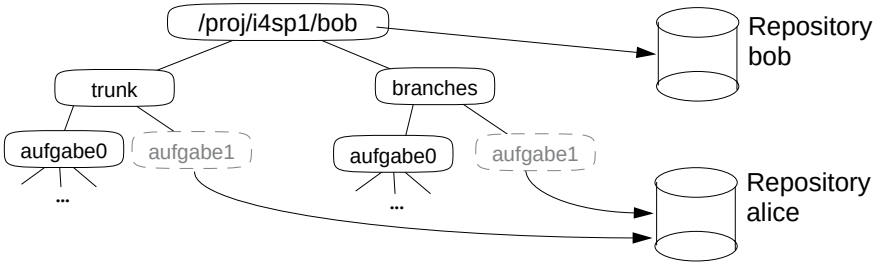
```
alice@faui01$ /proj/i4sp1/bin/submit aufgabe1
```


3 Ablauf für den Partner

- Partner setzt in seinem Repository einen Verweis auf Hauptrepository

```
bob@faui01$ /proj/i4sp1/bin/import-from-partner aufgabe1 alice
```

  - technisch: **`svn:externals`**-Property
    - irrelevant für Abgabe, unterstützt nur den Prozess der Teamarbeit
  - Achtung: Abgabe im eigenen Repository überlagert Partnerabgabe
    ➥ zum Umstieg auf Teamarbeit eigene Abgabe löschen (Übungsleiter hilft)



3 Ablauf für den Partner

- Arbeit in der eigenen Arbeitskopie fast normal möglich
  - Der Befehl `svn commit` übermittelt nur Änderungen an das Repository das für den aktuellen Pfad zuständig ist

```
bob@faui01[/proj/i4sp1/bob/trunk]$ svn status
M       aufgabe1/lilo.c
bob@faui01[/proj/i4sp1/bob/trunk]$ svn commit
# Es erfolgt kein Commit
bob@faui01[/proj/i4sp1/bob/trunk]$ svn status
M       aufgabe1/lilo.c
bob@faui01[/proj/i4sp1/bob/trunk]$ cd aufgabe1
bob@faui01[/proj/i4sp1/bob/trunk/aufgabe1]$ svn commit -m bla
Committed revision 5.
```

  - `svn commit` der Aufgabendateien muss im Verzeichnis `/proj/i4sp1/bob/trunk/aufgabe1` erfolgen

- Abgabe funktioniert wie gewohnt:

```
alice@faui01$ /proj/i4sp1/bin/submit aufgabe1
```


U2-3 Verkettete Liste

- Anforderungsanalyse für verkettete Liste
  - Wieviele Listenelemente gibt es maximal?
  - Welche Lebensdauer muss ein Listenelement besitzen?
  - In welchem Kontext muss ein Listenelement sichtbar sein?

U2-3 Verkettete Liste

- Anforderungsanalyse für verkettete Liste
  - Wieviele Listenelemente gibt es maximal?
  - Welche Lebensdauer muss ein Listenelement besitzen?
  - In welchem Kontext muss ein Listenelement sichtbar sein?
- Wir brauchen einen Mechanismus, mit dem Listenelemente
  - in a-priori nicht bekannter Anzahl
  - zur Laufzeit des Programms erzeugt und zerstört werden können

U2-4 Dynamische Speicherverwaltung

- Rückblick: Aufgabe 10.1 (Menschenkette) in AuD SS2011

```
WaitingHuman somebody = new WaitingHuman("Mrs. Somebody");
WaitingHuman nobody = new WaitingHuman("Mr. Nobody");
somebody.add(nobody);
```

  - In Java: Neues Listenelement wird mit Hilfe von `new` instanziiert
    - Reservieren eines Speicherbereichs für das Objekt
    - Initialisieren des Objektes durch Ausführen des Konstruktors

- In C: Anlegen eines Listenelements mittels `malloc()`

```
struct listelement *newElement;
newElement = malloc( sizeof( struct listelement ) );
if( newElement == NULL ) {
    // Fehlerbehandlung
}
```

  - Zurückgegebener Speicherbereich ist uninitialisiert
  - Initialisierung muss per Hand erfolgen

U2-4 Dynamische Speicherverwaltung

- Explizite Initialisierung mit `void *memset(void *s, int c, size_t n)`

```
memset(newElement, 0, sizeof( struct listelement ) );
```

- Vorinitialisierter Speicher kann mit `calloc()` angefordert werden:
  `void *calloc(size_t nelem, size_t elsize)`

```
struct listelement *newElement;
newElement = calloc( 1, sizeof( struct listelement ) );
if( newElement == NULL ) { ... }
```


U2-4 Dynamische Speicherverwaltung

- Explizite Initialisierung mit `void *memset(void *s, int c, size_t n)`

```
memset(newElement, 0, sizeof( struct listelement ) );
```

- Vorinitialisierter Speicher kann mit `calloc()` angefordert werden:
  `void *calloc(size_t nelem, size_t elsize)`

```
struct listelement *newElement;
newElement = calloc( 1, sizeof( struct listelement ) );
if( newElement == NULL ) { ... }
```

- Im Gegensatz zu Java gibt es in C keinen Garbage-Collection-Mechanismus
  - Speicherbereich muss von Hand freigegeben werden

```
free(newElement);
```

  - Nur Speicher, der mit einer der alloc-Funktionen zuvor angefordert wurde, darf mit `free()` freigegeben werden!

- Der Zugriff auf freigegebene Speicherbereiche hat undefiniertes Verhalten.

U2-5 Fehler finden mit valgrind

- Baukasten von Debugging- und Profiling-Werkzeugen (ausführbarer Code wird durch synthetische CPU auf Softwareebene interpretiert ➔ Ausführung erheblich langsamer!)
  - Memcheck: erkennt Speicherzugriff-Probleme
    - ➤ Nutzung von nicht-initialisiertem Speicher
    - ➤ Zugriff auf freigegebenen Speicher
    - ➤ Zugriff über das Ende von allokierten Speicherbereichen
    - ➤ Zugriff auf ungültige Stack-Bereiche
- Aufrufbeispiel: `valgrind ./lilo`
- Programm muss mit der Compileroption `-g` übersetzt werden.

```
gcc -g -o lilo lilo.c
```

  - Sogenannte Debug-Symbole werden vom Compiler in die ausführbare Datei eingebaut. Diese wertet valgrind aus um z.B. die Zeilennummer eines Funktionsaufrufs zu erhalten.

1 Fehler finden mit valgrind

- Zugriffe auf nicht allokierten Speicher finden

```
==11711== Invalid read of size 4
==11711==    at 0x804841B: main (gdb.c:19)
==11711==  Address 0x0 is not stack'd, malloc'd or (recently) free'd
==11711==
==11711==
==11711== Process terminating with default action of signal 11
(SIGSEGV)
==11711==  Access not within mapped region at address 0x0
```

  - In Zeile 19 wurden 4 Byte von Adresse 0x0 gelesen
    - Auch schreibende Zugriffe können entdeckt werden
  - Darauf hin wurde der Prozess beendet

1 Fehler finden mit valgrind

- Auffinden von nicht freigegebenem Speicher

```
==5344== HEAP SUMMARY:
==5344==     in use at exit: 16 bytes in 2 blocks
==5344==   total heap usage: 4 allocs, 2 frees, 32 bytes allocated
```

  - Bei Programmende sind noch 2 Speicherbereiche (Blöcke) belegt
  - Während der Programmausführung wurde viermal **`malloc()`** und nur zweimal **`free()`** aufgerufen
  - Mit Hilfe der Option **`--leak-check=full --show-reachable=yes`** wird angezeigt, wo der Speicher angelegt wurde, der nicht freigegeben wurde.

```
==5865== 8 bytes in 1 blocks are still reachable in loss record 1 of 2
==5865==    at 0x48DAF50: malloc (vg_replace_malloc.c:236)
==5865==    by 0x80484B9: insertElement (lilo.c:28)
==5865==    by 0x8048599: main (lilo.c:61)
```

    - In der Zeile 28 wurde der Speicher angelegt
    - Nun im Quellcode Stellen identifizieren, an denen **`free()`**-Aufrufe fehlen

1 Fehler finden mit valgrind

- Auffinden uninitialisierten Speichers

```
==18108== Conditional jump or move depends on uninitialised value(s)
==18108==    at 0x80484AC: insertElement (lilo.c:19)
==18108==    by 0x8048566: main (lilo.c:59)
```

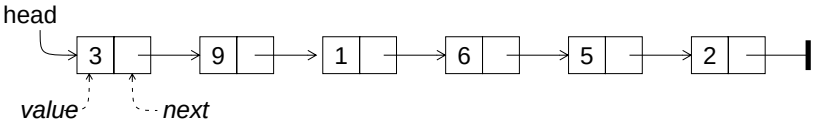
  - In der Funktion **`insertElement()`** in Zeile 19 wird auf uninitialisierten Speicher zugegriffen.
  - Mit Hilfe der Option **`--track-origins=yes`** wird angezeigt, wo der uninitialisierte Speicher angelegt wurde.

```
==18741== Conditional jump or move depends on uninitialised value(s)
==18741==    at 0x80484AC: insertElement (lilo.c:19)
==18741==    by 0x8048566: main (lilo.c:59)
==18741==  Uninitialised value was created by a heap allocation
==18741==    at 0x48DAF50: malloc (vg_replace_malloc.c:236)
==18741==    by 0x80484B9: insertElement (lilo.c:28)
==18741==    by 0x8048549: main (lilo.c:58)
```


U2-6 Aufgabe 1: lilo (last in - last out)

1 Einfach verkettete FIFO-Liste

- Zielsetzungen
  - Kennenlernen der Umgebung und Entwicklungswerkzeuge
  - Dynamische Speicherverwaltung und Umgang mit Zeigern
  - Verwendung des Abgabesystems



- Strukturdefinition:

```
struct listelement {
   int value;
   struct listelement *next;
};
typedef struct listelement listelement; // optional
```


2 Schnittstelle

- Nur die beiden folgenden Funktionen müssen implementiert werden
  - **int insertElement(int value)**: Fügt einen neuen, nicht-negativen Wert in die Liste ein, wenn dieser noch nicht vorhanden ist. Rückgabe *value* im Erfolgsfall, sonst -1.
  - **int removeElement()**: Entfernt den ältesten Wert in der Liste und gibt diesen zurück. Ist die Liste leer, wird -1 zurückgeliefert.
- Keine Listen-Funktionalität in die **`main()`**-Funktion schreiben.
  - Allerdings: Erweitern der **`main()`** zum Testen erlaubt und erwünscht
- Sollte bei der Ausführung einer verwendeten Funktion (z.B. **`malloc()`**) ein Fehler auftreten, sind keine Fehlermeldungen auszugeben.