SNP Laboratories

welo, bazz, fisa, huno, grop, donn, scia

Last updated : February, 2022

More documents are available at github.zhaw.ch

Table of contents

Chapter 1

01 - Erste Schritte mit C

1.1 1. Übersicht

In diesem Praktikum erstellen Sie mehrere kleine C-Programme, in denen Sie Input- und Output-Funktionen der C Standard Library verwenden.

Arbeiten Sie in Zweiergruppen und diskutieren Sie ihre Lösungsansätze miteinander, bevor Sie diese umsetzen.

Bevor Sie mit den Programmieraufgaben beginnen, setzen Sie eine virtuelle Maschine mit der vorbereiteten Praktikumsumgebung auf.

1.2 2. Lernziele

In diesem Praktikum schreiben Sie selbst von Grund auf einige einfache C-Programme und wenden verschiedene Kontrollstrukturen an.

- Sie können mit *#include* Funktionen der C Standard Library einbinden
- Sie können mit *#define* Macros definieren und diese anwenden
- Sie wenden die *Input-* und *Output-Funktionen* von C an, um Tastatur-Input einzulesen und formatierte Ausgaben zu machen.
- Sie verwenden die Kommandozeile, um ihren Sourcecode in ein ausführbares Programm umzuwandeln.
- Sie wenden for-und while-Loops sowie if-then-else-Verzweigungen an.
- Sie setzen eine Programmieraufgabe selbständig in ein funktionierendes Programm um.

1.3 3. Aufgabe 1: virtuelle Maschine

Im Moodle-Kurs "Systemnahe Programmierung" finden Sie unter "Praktika" eine Installationsanleitung für die virtuelle Maschine, die wir Ihnen zur Verfügung stellen. Die virtuelle Maschine enthält ein Ubuntu Linux-Betriebssystem und die für das Praktikum benötigten Frameworks.

Folgen sie der Anleitung, um die virtuelle Maschine auf ihrem Rechner zu installieren.

1.4 4. Aufgabe 2: Hello World

Schreiben Sie ein C-Programm, das "Hello World" auf die Standardausgabe schreibt. Verwenden Sie die printf-Funktion aus der Standard Library. In den Vorlesungsfolien finden Sie bei Bedarf eine Vorlage.

Erstellen sie das Source-File mit einem beliebigen Editor, sie benötigen nicht unbedingt eine IDE. Speichern Sie das Source-File mit der Endung `.c`.

Um ihr Source-File zu kompilieren, verwenden Sie den GNU Compiler auf der Kommandozeile:

```
$> gcc hello.c
```

Der Compiler übersetzt ihr Programm in eine ausführbare Datei `a.out`, die Sie mit

```
$> ./a.out
```

ausführen können. Sie können den Namen der ausführbaren Datei wählen, indem Sie die Option `-o` verwenden:

```
$> gcc hello.c -o hello
```

erzeugt die ausführbare Datei `hello`.

Verwenden Sie die Option `-Wall`, um alle Warnungen des Compilers auszugeben. Dies weist Sie auf allfällige Programmierfehler hin.

1.5 5. Aufgabe 3: Tabellenausgabe

Schreiben Sie ein Programm in C, das von `stdin` einen Umrechnungsfaktor zwischen CHF und Bitcoin einliest und danach eine Tabelle von Franken- und Bitcoin-Beträgen ausgibt. Die Tabelle soll sauber formatiert sein, z.B. so:

```
Enter conversion rate (1.00 BTC -> CHF): 43158.47
  200 CHF        <-->         0.00463 BTC
  400 CHF        <-->         0.00927 BTC
  600 CHF        <-->         0.01390 BTC
  800 CHF        <-->         0.01854 BTC
 1000 CHF        <-->         0.02317 BTC
 1200 CHF        <-->         0.02780 BTC
 1400 CHF        <-->         0.03244 BTC
 1600 CHF        <-->         0.03707 BTC
```

- Verwenden Sie eine Schleife und die `printf`-Funktion für die Tabellenausgabe
- Definieren Sie ein Makro `NUM_ROWS`, um an zentraler Stelle im Source-Code zu definieren, wie viele Einträge die Tabelle in der Ausgabe haben soll.
- Lesen Sie den Umrechnungsfaktor mit der `scanf`-Funktion als `double` von der Kommandozeile ein.

1.6 6. Aufgabe 4: Zeichen und Wörter zählen

Schreiben Sie ein C-Programm, welches die Zeichen und Wörter einer mit der Tastatur eingegebenen Zeile zählt. Wortzwischenräume sind entweder Leerzeichen (' ') oder Tabulatoren ('\t'). Die Eingabe der Zeile mit einem newline-character ('\n') abgeschlossen. Danach soll ihr Programm die Anzahl Zeichen und die Anzahl Wörter ausgeben und terminieren.

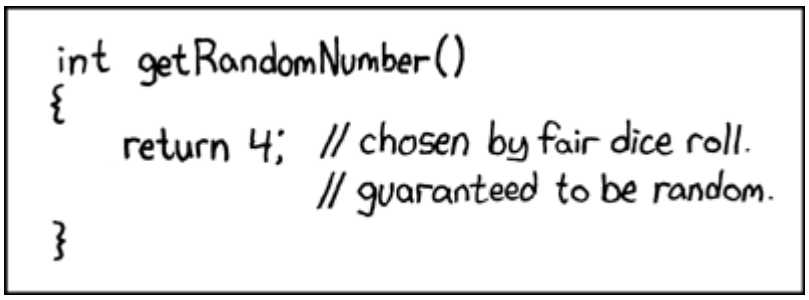
- Verwenden Sie die `char getchar(void)` Funktion aus der `stdio.h` Library, um die Zeichen einzeln einzulesen. Die Funktion `getchar` kehrt nicht gleich bei Eingabe des ersten Zeichens zurück, sondern puffert die Daten, bis die Eingabe einer kompletten Zeile mit Return abgeschlossen wird. Dann wird das erste Zeichen aus dem Puffer zurückgegeben und mit weiteren Aufrufen von getchar können die nachfolgenden Zeichen aus dem Puffer gelesen werden. Gibt `getchar` das Zeichen `\n` zurück, ist die Zeile komplett zurückgegeben und der Puffer ist wieder leer.
- Setzen Sie eine Schleife ein, die beim Zeichen '\n' terminiert.
- Benutzen Sie if-then-else-Strukturen um die Wörter zu zählen.

1.7 7. Bewertung

Die gegebenenfalls gestellten Theorieaufgaben und der funktionierende Programmcode müssen der Praktikumsbetreuung gezeigt werden. Die Lösungen müssen mündlich erklärt werden.

Chapter 2

02: Funktionen, Datentyp “enum”



(Copyright Bild: xkcd.com)

2.1 1. Übersicht

In diesem Praktikum sind zwei Themen im Fokus: Funktionen und der Datentyp enum.

Funktionen sind der wesentlichste Bestandteil der C Programmierung welcher eine strukturierte Programmierung ermöglicht:

- Eine Funktion ein Teil eines C Codes, der eine spezielle Aufgabe ausführt. Sie kann aus dem Hauptprogramm, oder aus anderen Funktionen, aufgerufen werden.
- Jede Funktion besitzt einen eindeutigen Namen, eine eindeutige Signatur (Typen und Reihenfolge der Parameter) und einen Rückgabewert (int falls nichts angegeben wird).
- Eine Funktion kann Werte aus dem aufrufendem Kontext übernehmen und bei Bedarf einen Wert an den aufrufenden Kontext zurückliefern. Beispiel einer Additions-Funktion:

```c
#include <stdio.h>

/* Funktionsdeklaration */
int add(int a, int b);

int main(void) {
   int aa = 1, bb = 2, cc;
   printf("%a + %bb = %cc", aa, bb, add(aa, bb););
 return 0;
}
```

(continues on next page)

(continued from previous page)

```
/* Funktionsdefinition */
int add(int a, int b) {
   return a + b;
}
```

Der Daten typt enum wird verwendet um die Lesbarkeit von Programmen zu erhöhen:

Beispiel eines enum:

```
enum Ampeln = {rot =1, gelb, gruen};

int main(void) {
   Ampeln ampel1;
   if (ampel1 == rot) {...}
 return 0;
}
```

2.2 2. Lernziele

In diesem Praktikum lernen Sie Funktionen zu definieren und aufzurufen, sowie enum anzuwenden.

- Sie können ein Programm schreiben welches aus mehreren Funktionen besteht.
- Sie können Funktionen deklarieren, definieren und aufrufen.
- Sie können enum Typen definieren und deren Werte bestimmen und abfragen.

2.3 3. Aufgaben

(Copyright Bild: www.planet-wissen.de)

2.3.1 3.1 Aufgabe 1 Tage pro Monat

In der ersten Aufgabe berechnen Sie die Tag pro Monat einer beliebigen Kombination Monat / Jahr. Erweitern Sie dazu das Programm um folgende Aspekte:

- Bereichsprüfung von Jahr und Monat
- Funktion istSchaltjahr, welche berechnet, ob das Jahr eine Schaljahr ist
- Funktion tageProMonat, welche die Anzahl Tage des gegebenen Monats und Jahres berechnet.

Vorgaben:

- Die Funktion istSchaltjahr nimmt ein Integer (jahr) entgegen und gibt 1 im Falle eiens Schltjahres und 0 im andreren Fall zurück
- Die Funktion tageProMonat nimmt zwei integer (monat und jahr) entgegeben und gibt die Anzahl Tage als Integer zurück
- Die Jahreszahl, welche den Funktionen übergeben wird, muss überprüft werden und grösser gleich 1599 und kleiner als 10000 sein
- Der übergebene Monat muss grösser als 0 und kleine als 13 sein.

Die Regeln für die Schaltjahrberechnung:

- Schaltjahre sind alle Jahre, die durch 4 teilbar sind.
- Eine Ausnahme bilden die Jahrhunderte (1600, 1700...). Diese sind keine Schltjahre.
- zu den 100er gibt es ebenfalls Ausnahmen: Diese sind immer Schaltjahre, wenn sie durch 400 teilbar sind ... also zum Beispiel 1600 ist eines, nicht jedoch 1700. Weiterführende Details finden Sie unter https://de.wikipedia.org/wiki/Gregorianischer_Kalender

Gegeben ist die main Funktion des Programms. Ergänzen Sie die enum Definition und die fehlenden Funktionen:

- gibIntWert: Die Funktion soll einen Int Wert zurückgeben. Der Bereich, wie auch Fehleingaben sollen sollen berücksichtigt werden. (atoi unfd fgets sind hier hilfreich)
- istSchaltjahr: Die Funktion gibt 1 im Falle eines Schltjahr und o im anderen Falle zurück.
- tageProMonat: Die Funktion gibt den die Tage des Monats für das definierte Jahr zurück. Verwenden Sie die Switchanweisung , sowie den enum Datentypen

```
int main (int argc, char *argv[]) {

    int monat, jahr;

    //  Monat einlesen und Bereich ueberpruefen
    monat = gibIntWert("Monat", 1, 12);
    jahr  = gibIntWert("Jahr", 1600, 9999);

    //  Ausgabe zum Test
    printf("Monat: %d, Jahr: %d \n", monat, jahr);

    //  Ausgabe zum Test (hier mit dem ternaeren Operator "?:")
    printf("%d ist %s Schaltjahr\n", jahr, istSchaltjahr(jahr) ? "ein" : "kein");

        // Ausgabe
        printf("Der Monat %02d-%d hat %d Tage.\n", monat, jahr, tageProMonat(jahr, monat));

    return 0;
}
```

Tipp: Angenommen Sie verwenden den enum month_t { JAN=1, FEB, MAR, APR, MAI, JUN, JUL, AUG, SEP, OKT, NOV, DEZ }; Dann können Sie im Programm direkt die Konstanten verwenden:

```
if (m == 2) ...                 // schlecht lesbar
if (monat == 2) ...         // besserer Variablenname
if (monat == FEB) ...         // am besten lesbar
```

Als Abnahme müssen die Tests unverändert ohne Fehler ausgeführt werden (`make test`)

2.3.2 3.2 Aufgabe 2 Bestimmen des Wochentags

Erweitern Sie das vorgegebene zweite Programm Gerüst an den bezeichneten Stellen so, dass das Programm von der Kommando Zeile ein Argument entgegennimmt, es auf Gültigkeit überprüft und schliesslich den Wochentag für das gegebene Datum berechnet und ausgibt. Prüfen Sie die Umsetzung beider Teilaufgaben mittels make test.

2.3.2.1 3.2.1 Teilaufgabe Argumente Parsen und auf Korrektheit prüfen

Das Argument stellt ein gültiges Datum unseres Gregorianischen Kalenders dar (d.h. ein Datum ab Donnerstag, den 15. Oktober 1582, mit der Gregorianischen Schaltjahr Regel). Wenn kein Argument gegeben ist oder wenn das eingegebene Datum nicht gültig ist, soll das Programm einem Hilfetext auf stderr ausgeben und mit EXIT_FAILURE Exit Code terminieren. Wenn ein gültiges Datum erkannt wurde terminiert das Programm mit Exit Code EXIT_SUCCESS.

3.2.1.1 Argument Format

Das Format des Kommando Zeilen Arguments soll yyyy-mm-dd sein, wobei yyyy für das vier-stellige Jahr, mm für einen 1-2-stelligen Monat (1...12) und dd für einen Tag des Monats, begin-nend mit 01. Z.B. 2020-02-29.

3.2.1.2 Korrektes Datum

Das Datum muss alle folgenden Bedingungen erfüllen damit es als korrekt erkannt wird:

- Obergrenze für ein «sinnvolles» Datum ist das Jahr 9999
- es muss Gregorianisch sein, d.h. ab 15. Oktober 1582 (inklusive)
- es darf nur Monate von 1 für Januar bis 12 für Dezember beinhalten
- der Tag muss grösser oder gleich 1 sein
- der Tag darf nicht grösser als 31 sein für Monate mit einer Länge von 31 Tagen
- der Tag darf nicht grösser als 30 sein für Monate mit einer Länge von 30 Tagen
- der Tag darf für den Februar nicht grösser sein als 29 für ein Schaltjahr
- der Tag darf für den Februar nicht grösser sein als 28 für ein Nicht-Schaltjahr

3.2.1.3 Vorgaben an die Umsetzung

1. Definieren Sie einen enum Typen mit (typedef) Namen month_t dessen Werte die Englischen 3-Zeichen Abkürzungen der Monate sind, nämlich Jan, Feb, ... Dec und stellen Sie sicher dass die Abkürzungen für die uns geläufigen Monatsnummer stehen.
2. Definierend Sie einen struct Typen mit (typedef) Namen date_t und den int Elementen year, month, day. Lesen Sie das Argument (falls vorhanden) via sscanf und dem Formatstring "%d-%d-%d" in die drei Elemente einer Date Variable. Siehe dazu die Hinweise im Anhang.
3. Für die Berechnung der Monatslänge implementieren Sie die Hilfsfunktion is_leap_year(date_t date) (nach obigen Vorgaben). Der Return Wert 0 bedeutet «Kein Schaltjahr», 1 bedeutet «Schaltjahr».
4. Implementieren Sie die Funktion `int get_month_length(date_t date)`. Diese soll für den Monat des Datums die Monatslänge (was dem letzten Tag des Monats ent-spricht) ausgeben – geben Sie 0 für ungültige Monatswerte zurück.
5. Schliesslich implementieren Sie die Funktion int is_gregorian_date(date_t date) welche prüft, ob ein gegebenes Datum im Bereich 15. Oktober 1582 und dem Jahr 9999 ist (0 = nein, 1 = ja).
6. Implementieren Sie eine Funktion int is_valid_date(date_t date), welche obige Bedingungen für ein gültiges Datum umsetzt. Der Return Wert 0 bedeutet «Kein gültiges Datum», 1 bedeutet «Gültiges Datum». Benutzen Sie für die Prüfung des Datums die `month_t` Werte wo immer möglich und sinnvoll. Verwenden Sie die oben implemen-tierten Hilfsfunktionen.

3.2.1.4 Hinweise

Beachten Sie die Kommentare im Code für die geforderten Implementierungs-Details.

2.3.2.2 3.2.2 Teilaufgabe Wochentag Berechnung

Schreiben Sie eine Funktion welche zu einem Datum den Wochentag berechnet. Die Formel wird Georg Glaeser zugeschrieben, möglicherweise angelehnt an eine Formel von Carl Friedrich Gauss.

$$w = (d + \lfloor 2{,}6 \cdot m - 0{,}2 \rfloor + y + \left\lfloor \frac{y}{4} \right\rfloor + \left\lfloor \frac{c}{4} \right\rfloor - 2c) \bmod 7$$

(Quelle: https://de.wikipedia.org/wiki/Wochentagsberechnung)

Hier ist eine für C abgewandelte Variante davon.

```
weekday = ((day + (13 * m - 1) / 5 + y + y / 4 + c / 4 - 2 * c) % 7 + 7) % 7
alle Zahlen sind int Werte und alles basiert auf int-Arithmetik
m = 1 + (month + 9) % 12
a = year - 1 (für month < Mar), ansonsten year
y = a % 100
c = a / 100
```

Erweitern sie das Programm so, dass vor dem erfolgreichen Terminieren des Programms fol-gende Zeile (inklusive Zeilenumbruch) ausgegeben wird: yyyy-mm-dd is a Ddd, wobei yyyy für das Jahr, mm für die Nummer des Monats (01...12) und dd für den Tag im Monat (01...). Z.B. 2020-02-29 is a Sat. Vorgaben an die Umsetzung

1. Definieren Sie einen enum Typen mit (typedef) Namen weekday_t dessen Werte die Englischen 3-Zeichen Abkürzungen der Tage sind, nämlich Sun, Mon, ... Sat und stel-len Sie sicher dass die Abkürzungen für die Werte 0...6 stehen.
2. Schreiben Sie eine Funktion weekday_t calculate_weekday(date_t date) nach der Beschreibung der obigen Formel. Das date Argument ist als gültig angenom-men, d.h. es ist ein Programmier-Fehler, wenn das Programm diese Funktion mit einem ungültigen Datum aufruft. Machen Sie dafür als erste Codezeile in der Funktion eine Zu-sicherung (assert(is_valid_date(date));)
3. Schreiben Sie eine Funktion void print_weekday(weekday_t day), welche für jeden gülteigen Tag eine Zeile auf stdout schreibt mit den Englischen 3-Zeichen Ab-kürzungen für den Wochentag, z.B. Sonntag: Sun, Montag: Mon, etc. Wenn ein ungülti-ger Wert für day erkannt wird, soll assert(!"day is out-of-range"); aufgeru-fen werden. Hinweise • Für interessierte, siehe: https://de.wikipedia.org/wiki/Wochentagsberechnung

---

2.4 4. Bewertung

Die gegebenenfalls gestellten Theorieaufgaben und der funktionierende Programmcode müssen der Praktikumsbetreuung gezeigt werden. Die Lösungen müssen mündlich erklärt werden können.

Aufgabe	Kriterium	Gewicht
alle	Sie können das funktionierende Programm inklusive funktionierende Tests demonstrieren und erklären.	
gibIntWert	Eingabe, Bereichsüberprüfung korrekt	1
istSchaltjahr	Funktion korrekt	1
TageProMonat	Funktion korrekt	1
Aufgabe 2	Fehlenden Teile ergänzt und lauffähig	1

2.5 5. Anhang

2.5.1 5.1 Sprach Element

```
  ...
}        argc: Anzahl Einträge in argv.
argv: Array von Command Line Argumenten.
argv[0]: wie das Programm gestartet wurde
argv[1]: erstes Argument
...
argv[argc-1]: letztes Argument
int a = 0;
int b = 0;
int c = 0;
int res = sscanf(argv[1]
                 , "%d-%d-%d"
                 , &a, &b, &c
                 );
if (res != 3) {
    // Fehler Behandlung...
    // ...
}
```

2.5.2 5.2 Beschreibung

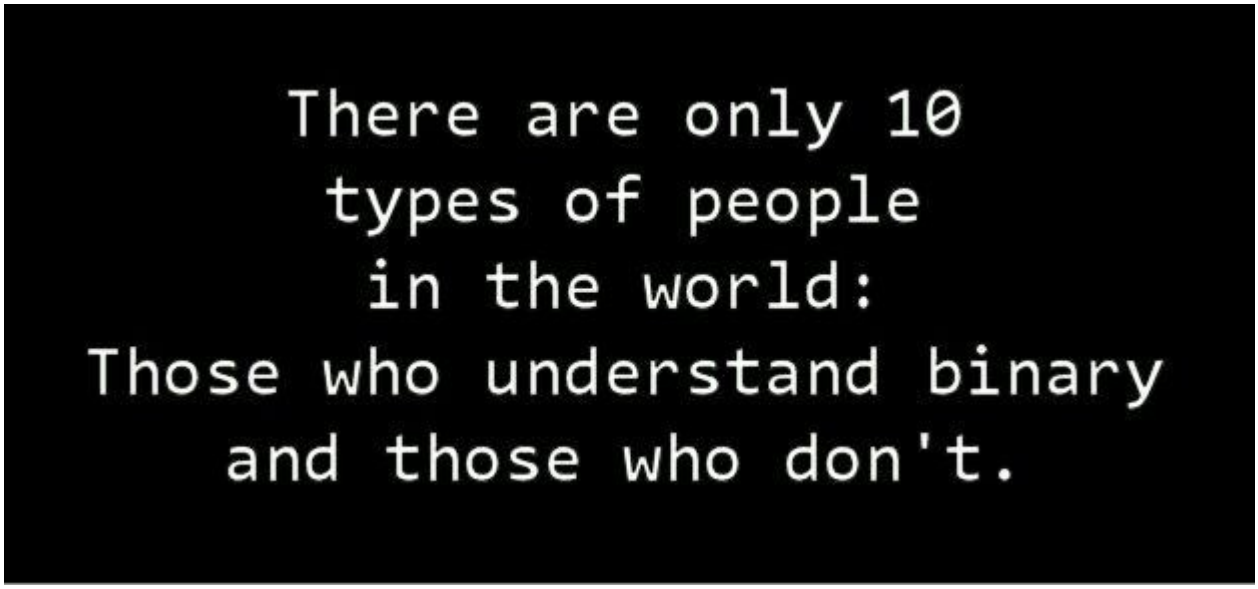
Siehe man 3 sscanf. Die Funktion sscanf gibt die Anzahl erfolgreich erkannte Argumente zurück. Unbedingt prüfen und angemessen darauf reagieren. Die gelesenen Werte werden in a, b und c, gespeichert, dazu müssen Sie die Adresse der Variablen übergeben. Mehr Details dazu werden später erklärt. fprintf(stderr, "Usage: %s... \n", argv[0]); Siehe man 3 fprintf. Schreibt formatierten Text auf den stderr Stream.

Version: 15.02.2022

Chapter 3

03 - Bit Operationen, Struct, Typedef

3.1 1. Bit Operationen



Bit Operationen sind allgegenwärtig in den Computer-Wissenschaften und finden in vielen Disziplinen Anwendung. Folgend ein kleiner Auszug aus den wichtigsten Themen:

- **Bit Felder**: Sind die effizienteste Art, etwas darzustellen, dessen Zustand durch mehrere "wahr" oder "falsch" definiert werden kann. Besonders auf Systemen mit begrenzten Ressourcen sollte jede überflüssige Speicher-Allozierung vermieden werden.
  Beispiel:

```
// primary colors
#define BLUE  0b100
#define GREEN 0b010
#define RED   0b001

// mixed colors
#define BLACK   0 /* 000 */
#define YELLOW  (RED | GREEN) /* 011 */
#define MAGENTA (RED | BLUE) /* 101 */
#define CYAN    (GREEN | BLUE) /* 110 */
#define WHITE   (RED | GREEN | BLUE) /* 111 */
```

https://de.wikipedia.org/wiki/Bitfeld

- **Kommunikation**:
  - **Prüfsummen/Paritätsbit**: Übertragungsfehler und Integrität können bis zu einem definiertem Grad erkannt werden. Je nach Komplexität der Berechnung können mehrere Fehler erkannt oder auch korrigiert werden. https://de.wikipedia.org/wiki/Parit%C3%A4tsbit, https://de.wikipedia.org/wiki/Pr%C3%BCfsumme
  - **Stoppbit**: Markieren bei asynchronen seriellen Datenübertragungen das Ende bzw. Start eines definierten Blocks. https://de.wikipedia.org/wiki/Stoppbit
  - **Datenflusssteuerung**: Unterschiedliche Verfahren, mit denen die Datenübertragung von Endgeräten an einem Datennetz, die nicht synchron arbeiten, so gesteuert wird, dass eine möglichst kontinuierliche Datenübermittlung ohne Verluste erfolgen kann. https://de.wikipedia.org/wiki/Datenflusssteuerung
  - ...
- **Datenkompression**: Bei der Datenkompression wird versucht, redundante Informationen zu entfernen. Dazu werden die Daten in eine Darstellung überführt, mit der sich alle – oder zumindest die meisten – Information in kürzerer Form darstellen lassen. https://de.wikipedia.org/wiki/Datenkompression
- **Kryptographie**: Konzeption, Definition und Konstruktion von Informationssystemen, die widerstandsfähig gegen Manipulation und unbefugtes Lesen sind. https://de.wikipedia.org/wiki/Verschl%C3%BCsselung
- **Grafik-Programmierung**: XOR (oder ^) ist hier besonders interessant, weil eine zweite Eingabe derselben Eingabe die erste rückgängig macht (ein Beispiel dazu weiter unten: "Variablen tauschen, ohne Dritt-Variable "). Ältere GUIs verwendeten dies für die Hervorhebung von Auswahlen und andere Überlagerungen, um kostspielige Neuzeichnungen zu vermeiden. Sie sind immer noch nützlich in langsamen Grafikprotokollen (z. B. Remote-Desktop).

3.1.1 1.1 Übungen

3.1.1.1 1. Basis Operationen

Manipulationen von einzelnen Bits gehören zu den Basis Operationen und dienen als Grundlagen um weitere komplexere Konstrukte zu schaffen. Verfollständigen sie folgendes Beispiel mit den drei Basis Operationen:

```c
#include <stdlib.h>

int main() {
  unsigned int number;
  unsigned int bit = 3; // bit at position 3

  // Setting a bit
  number = ...; // solution: number |= 1 << bit;

  // Clearing a bit
  number = ...; // solution: number &= ~(1 << bit);

  // Toggling a bit
  number = ...; // solution; number ^= 1 << bit;

  return EXIT_SUCCESS;
}
```

3.1.1.2 2. Variablen tauschen (ohne Dritt-Variable)

Zwei Variablen zu vertauschen scheint ein einfach lösbares Problem zu sein. Eine offensichtliche Variante wäre mittels einer temporären Variablen:

```c
#include <stdlib.h>
#include <stdio.h>

int main(){
  int a = 3;
  int b = 4;
  printf("a: %d; b: %d\n", a, b);

  int temp = a;
  a = b;
  b = temp;

  printf("a: %d; b: %d\n", a, b);
  return EXIT_SUCCESS;
}
```

Es gibt aber auch eine Variante, die ohne zusätzliche Variable auskommt. Dabei wird die Tatsache, dass eine zweite XOR Operation eine erste XOR Operation rückgängig macht:

0011 XOR 0100 = 0111

0111 XOR 0100 = 0011

Somit kommt man von einem XOR Resultat (*0111*) wieder auf beide Anfangs Operanden zurück indem man einfach ein zweites Mal mit einem Operanden eine XOR Verknüpfung macht. Damit kann ein Operand als Zwischenspeicher dienen und man muss nicht extra eine Zusatzvariable verwenden.

Überlegen sie sich wie sie damit zwei Variablen vertauschen können ohne Zusatzvariable:

```c
#include <stdlib.h>
#include <stdio.h>

int main(){
  int a = 3;
  int b = 4;
  printf("a: %d; b: %d\n", a, b);


  ...

  /* Solutions:
          // a == 0011; b == 0100
  a ^= b; // a == 0111; b == 0100
  b ^= a; // a == 0111; b == 0011
  a ^= b; // a == 0100; b == 0011
  */

  printf("a: %d; b: %d\n", a, b);
  return EXIT_SUCCESS;
}
```

3.1.1.3 3. Lower- / Uppercase

```c
#include <stdlib.h>
#include <stdio.h>

int main(){
  char word[8] = "sREedEv";
  char *wordptr = &word[0];

  while(wordptr < &word[7]) {
    printf("UPPERCASE: %c\n", *wordptr & '_'); // converts the char into uppercase regardless␣
↪of the current casing
    printf("LOWERCASE: %c\n", *wordptr | ' '); // converts the char into lowercase regardless␣
↪of the current casing
    wordptr++;
  }

  return EXIT_SUCCESS;
}
```

3.1.1.4 4. Prüfen auf 2-er Potenz

```c
#include <stdio.h>
#include <stdlib.h>

int main(){
  int a=32;
  if(a > 0 && (a & (a - 1)) == 0){
    printf("%d is a power of 2", a);
  }
  return EXIT_SUCCESS;
}
```

3.2 2. Struct & typedef

3.2.1 2.1 Übungen

3.2.1.1 1. Bit Operationen Rechner

- Bitweise Operationen mit 2 Operanden
- Rechnung wird als ein String über scanf dem Programm übergeben
  - String wird in Token zerstückelt und in struct gespeichert:

```c
typedef struct {
  unsigned int operand_1;
  unsigned int operand_2;
  char operation;
} Expression;
```

  - Ausgabe in 3 verschiedenen Formaten:

```
Bin:
  0000'0000'0000'0001
& 0000'0000'0000'0011
  -------------------
  0000'0000'0000'0001

  Hex
  0x01 & 0x03 = 0x01

  Dec
  1 & 3 = 1
```

3.3 4. Bewertung

Die gegebenenfalls gestellten Theorieaufgaben und der funktionierende Programmcode müssen der Praktikumsbetreuung gezeigt werden. Die Lösungen müssen mündlich erklärt werden können.

Aufgabe	Kriterium	Gewicht
alle	Sie können das funktionierende Programm inklusive funktionierende Tests demonstrieren und erklären.	
gibIntWert	Eingabe, Bereichsüberprüfung korrekt	1
istSchaltjahr	Funktion korrekt	1
TageProMonat	Funktion korrekt	1
Aufgabe 2	Fehlenden Teile ergänzt und lauffähig	1

Chapter 4

04 - Modularisieren von C Code

4.1 1. Übersicht

In diesem Praktikum wird eine kleine Sammlung von Funktionen als Modul erstellt.

In der ersten Aufgabe schreiben Sie zu einem bestehenden C Programm die notwendigen Header Files plus passen das Makefile so an, dass die entsprechenden Module mit kompiliert werden.

In der zweiten Aufgabe erstellen Sie Makefile Regeln um aus Konfigurationsdateien graphischen Darstellungen zu erzeugen.

4.2 2. Lernziele

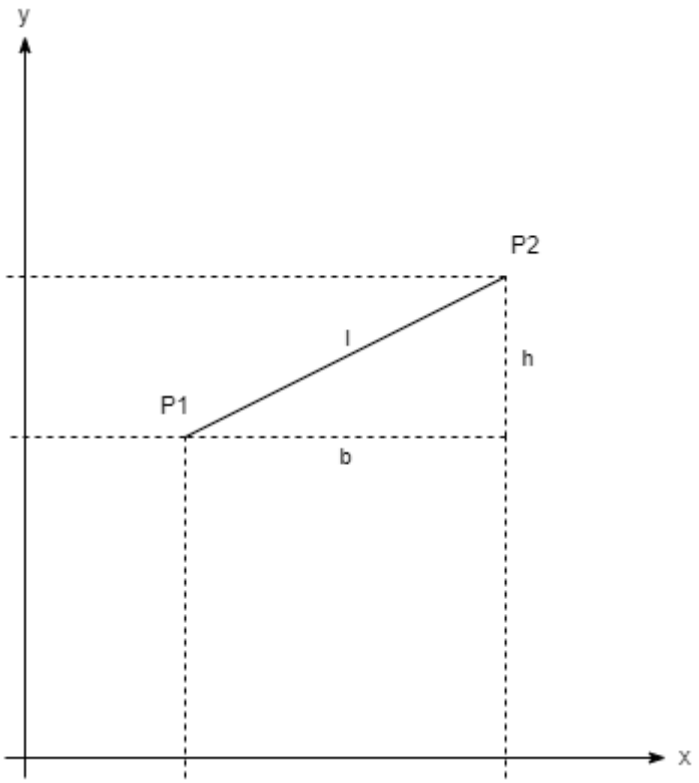
In diesem Praktikum lernen Sie die Handgriffe um ein Programm zu modularisieren, d.h. in mehrere Module aufzuteilen.

- Sie wissen, dass ein Modul aus einem C-File und einem passenden H-File bestehen.
- Sie können Header Files korrekt strukturieren.
- Sie wissen wie **Include Guards** anzuwenden sind.
- Sie können Module im **Makefile** zur Kompilation hinzufügen.
- Sie können anhand einer Beschreibung Typen und Funktionen in den passenden Header Files deklarieren.
- Sie können **Makefile** Regeln schreiben.

Die Bewertung dieses Praktikums ist am Ende angegeben.

Erweitern Sie die vorgegebenen Code Gerüste, welche im **git** Repository **snp-lab-code** verfügbar sind.

4.3 3. Aufgabe 1: Modularisieren



4.3.1 3.1 Teilaufgabe Modules einbinden, Header Files schreiben

- src/objects.h
  - 2 Datenstukturen definieren
  - `struct point` mit 2 double für x und y Koordinate
  - `struct line` mit 2 point
- src/functions.h und .c
  - 2 Funktionen deklarieren und definieren
  - Berechnung der Länge `get_length`einer Linie (Annahme: Koordinaten sind alle positiv)
    - l = sqrt(h^ 2 + b^ 2)
    - ev. muss hier in den Anhang `#include <math.h>`
  - Berechnung der Steigung `get_slope` der Linie gegenüber dem Koordinatensystem
    - m = h / b
- tests vorgeben
- src/objects.h
  - Include Guard
  - Includes
  - Struct für Punkt und Linie
  - Include Guard
- src/functions.h
  - Include Guard
  - Includes
  - Deklarationen der Funktionen für Berechnung der Länge und Steigung
  - Include Guard
- src/functions.c
  - Includes
  - Definitionen der Funktionen für Berechnung der Länge und Steigung
  - Include Guard

4.4 4. Aufgabe 2: Makefile Regeln

Makefile ergänzen, damit Modul `functions` korrekt eingebunden und kompiliert wird.

1. Kompilieren Sie das ganze mittels **make clean default**. Es sollten keine Compiler Fehler auftreten.

4.4.1 4.1 Neue Regeln hinzufügen

- Vorraussetzung: tab2svg.sh aus Praktikum 3 wird um die Möglichkeit erweitert eine Linie zu zeichnen (`line:x1:y1:x2:y2:color`)
- Studierende erstellen
  - mind. 2 Files `long.line` und `short.line` mit 2 unterschiedlichen Linien
  - Makefile Regeln um aus einem File `.line` ein File `.svg` mit Hilfe des Scripts zu erstellen
  - PHONY Regel `display` um beide `.svg` mit Firefox darzustellen
    - Vorgabe: sie sollen eine Variable für die Input-Dateien nutzen

Nachdem das Programm in Aufgabe 1 umgesetzt ist, geht es nun darum, im **Makefile** Regeln zu definieren welche die einzelnen Schritte von den Source Files zu den **png** Files ausführen.

Prüfen Sie schliesslich die Umsetzung mittels `make display`.

4.5 5. Aufgabe 3

- Studierende sollen Ausgabe von `make doc` analysieren und die Include Diagramme erklären können

```
make doc
firefox doc/index.html &
```

4.6 6. Bewertung

Die gegebenenfalls gestellten Theorieaufgaben und der funktionierende Programmcode müssen der Praktikumsbetreuung gezeigt werden. Die Lösungen müssen mündlich erklärt werden.

4.7 7. Erweiterung Doxyfile für Abhängigkeitsanalyse

```
--- /home/vagrant/huno/snp-new/snp/praktika/Shared/work/Doxyfile        2022-02-07 21:16:42.
↪343302707 +0100
+++ /home/vagrant/snp/Doxyfile        2022-02-07 22:22:36.266839126 +0100
@@ -297,14 +297,14 @@
 UML_LOOK               = NO
 UML_LIMIT_NUM_FIELDS   = 10
 TEMPLATE_RELATIONS     = NO
-INCLUDE_GRAPH          = NO
```

(continues on next page)

(continued from previous page)

```
-INCLUDED_BY_GRAPH      = NO
+INCLUDE_GRAPH          = YES
+INCLUDED_BY_GRAPH      = YES
 CALL_GRAPH             = NO
 CALLER_GRAPH           = NO
-GRAPHICAL_HIERARCHY    = NO
-DIRECTORY_GRAPH        = NO
+GRAPHICAL_HIERARCHY    = YES
+DIRECTORY_GRAPH        = YES
 DOT_IMAGE_FORMAT       = png
-INTERACTIVE_SVG        = NO
+INTERACTIVE_SVG        = YES
 DOT_PATH               =
 DOTFILE_DIRS           =
 MSCFILE_DIRS           =
```

Chapter 5

05 - Arrays/Strings/TicTacToe

5.1 1. Übersicht

In diesem Praktikum werden Sie in der ersten Aufgabe ein Programm zum Einlesen, Sortieren und Ausgeben von Strings von Grund auf entwickeln.

In der zweiten Aufgabe werden Sie einen Programmrahmen zu einem funktionierenden TicTacToe-Spiel erweitern. Sie implementieren hierbei die fehlenden Funktionen bis alle Tests erfolgreich durchlaufen. Die gewählte Vorgehensweise entspricht somit Test-Driven-Development (TDD). D.h. es existieren zuerst Tests, welche alle fehlschlagen. Schrittweise werden die Funktionen implementiert bis alle Tests erfolgreich durchlaufen. Wenn die Tests erfolgreich durchlaufen, wird auch das Programm funktionieren.

5.2 2. Lernziele

In diesem Praktikum schreiben Sie selbst von Grund auf ein C-Programme, das mit Strings operiert. Ferner ergänzen Sie ein bestehendes und lernen dabei den Zugriff auf Arrays.

- Sie können mit Arrays von Strings umgehen.
- Sie können String-Funktionen aus der Standard Library verwenden.
- Sie können anhand einer Beschreibung im Code die fehlenden Funktionen die auf Arrays zugreifen implementieren.

5.3 3. Aufgabe 1: Sortieren von Strings

Schreiben Sie ein C-Programm, das bis zu 10 Wörter mit einer maximalen Länge von jeweils 20 char von der Tastatur einliest, diese in Grossbuchstaben umwandelt, in einem Array der Reihe nach ablegt und zum Schluss im Array alphabetisch sortiert und ausgibt. Wiederholt eingegebene Wörter sollen dabei ignoriert werden. Das Ende der Eingabe soll durch das Erreichen der zehn unterschiedlichen Wörter oder durch die Eingabe von „ZZZ“ erfolgen. Die Ausgabe der sortierten Wörter soll direkt nach Beendigung der Eingabe erfolgen.

Hinweise:

- Zur Speicherung der Wörter sollten Sie ein zweidimensionales Array verwenden.
- Verwenden Sie die String-Funktionen der C Standard Library (include <string.h>), z.B. um Strings alphabetisch zu vergleichen.

- Wenn Sie aus anderen Vorlesungen bereits einen effizienten Sortieralgorithmus kennen, können Sie diesen natürlich verwenden. Sonst erfinden Sie einfach einen eigenen.
- Strukturieren Sie das Programm durch geeignete Funktionen.

5.4 4. Aufgabe 2: TicTacToe

Das zu ergänzende Programm tic-tac-toe hat folgende Funktionalität:

1. es stellt ein 3x3 TicTacToe Spielbrett auf dem Terminal dar
2. es liest von stdin eine Ziffer 0...9 ein, wobei 0 für Programm-Terminieren, die übrigen Ziffern für die Wahl eines Feldes stehen
3. der erste Spielzug wird von Spieler A geführt, danach wechselt das Programm zwischen den Spielern A und B
4. bei Gewinn oder bei vollem Brett ist das Spiel vorbei

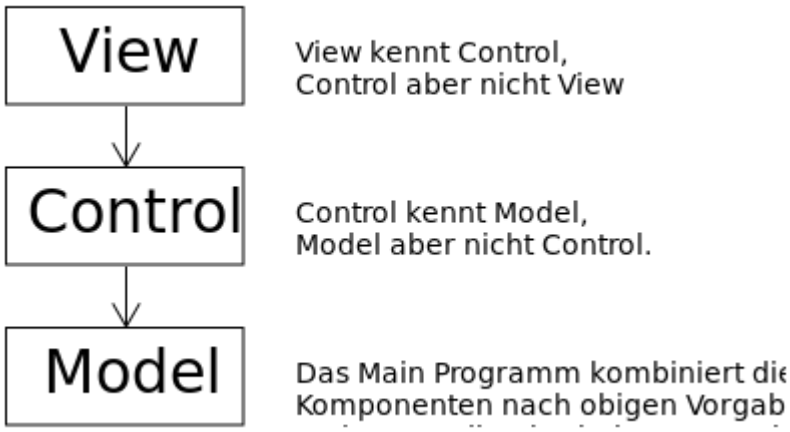
Erweitern Sie die vorgegebenen Code Gerüste, welche im git Repository snp-lab-code verfügbar sind.

Wenn die Aufgabe erfolgreich umgesetzt ist, können Sie das Spiel ausführen:

```
bin/tic-tac-toe
```

Als Abnahme müssen die Tests unverändert ohne Fehler ausgeführt werden (`make test`).

Die Architektur des Programms folgt dem MVC – Model-View-Control Paradigma. Dieses Paradigma besagt, dass die View (Eingabe und Darstellung) über Control (Vermittler) das Modell (die eigentliche Programm-Logik) steuert und darstellt. Dabei sind folgende Abhängigkeiten gegeben:



5.4.1 4.1 Teilaufgabe test_model_init

Das Programm besteht aus folgenden Files:

Datei	ToDo
Makefile	-> gegeben, d.h. nichts anzupassen
tests/tests.c	-> gegeben, d.h. nichts anzupassen
src/main.c	-> gegeben, d.h. nichts anzupassen
src/view.h	-> gegeben, d.h. nichts anzupassen
src/view.c	-> gegeben, d.h. nichts anzupassen
src/control.h	-> gegeben, d.h. nichts anzupassen
src/control.c	-> gegeben, d.h. nichts anzupassen
src/model.h	-> gegeben, d.h. nichts anzupassen
src/model.c	-> **anzupassen:** umsetzen gemäss den Angaben unten

1. Führen Sie `make test` aus

```
Suite: lab test
  Test: test_model_init ...
                init_model:... 0/0 FAILED
    1. tests/tests.c:62  - CU_ASSERT_EQUAL_FATAL(instance->board[row][col],model_state_none)
  Test: test_model_get_state ...FAILED
    1. tests/tests.c:62  - CU_ASSERT_EQUAL_FATAL(instance->board[row][col],model_state_none)
  Test: test_model_get_winner ...FAILED
    1. tests/tests.c:62  - CU_ASSERT_EQUAL_FATAL(instance->board[row][col],model_state_none)
  Test: test_model_can_move ...FAILED
    1. tests/tests.c:62  - CU_ASSERT_EQUAL_FATAL(instance->board[row][col],model_state_none)
  Test: test_model_move ...FAILED
    1. tests/tests.c:62  - CU_ASSERT_EQUAL_FATAL(instance->board[row][col],model_state_none)
  Test: test_model_get_win_line ...FAILED
    1. tests/tests.c:62  - CU_ASSERT_EQUAL_FATAL(instance->board[row][col],model_state_none)

Run Summary:    Type  Total    Ran Passed Failed Inactive
              suites      1      1    n/a      0        0
               tests      6      6      0      6        0
             asserts      6      6      0      6      n/a
```

1. Konzentrieren Sie sich auf den ersten Test der fehlschlägt. Dies ist ein Unit Test, welcher die Funktion **model_init()** prüft. Suchen Sie die Funktion in **src/model.h** und **src/model.c**.

2. Was ist die geforderte Funktionalität und wie ist sie implementiert?

Suchen Sie die darin aufgerufene **model_init()** Funktion und implementieren Sie diese.

```
void model_init(model_t *instance)
{
    assert(instance);

    // Instructions to the students:
    // set all fields of the board to model_state_none
    // BEGIN-STUDENTS-TO-ADD-CODE



    // END-STUDENTS-TO-ADD-CODE
}
```

1. Führen Sie `make test` und korrigieren Sie obige Funktion, bis der Test nicht mehr fehlschlägt.

5.4.2 4.2 Teilaufgabe test_model_get_state und test_model_get_winner

Gehen Sie analog zur ersten Teilaufgabe vor:

1. Führen Sie `make test` aus.
2. Suchen Sie die Funktion **model_get_state()** in **model.h** und **model.c**.
3. Implementieren Sie die intern benutzte Funktion **get_state()** gemäss der Anleitung im Code.

```
model_state_t model_get_state(model_t *instance, model_pos_t pos)
{
    assert(instance);
    assert_pos(pos);

    // Instructions to the students:
    // replace the stub implementation my access to the field at the given position.
    // BEGIN-STUDENTS-TO-ADD-CODE


    return model_state_none; // stub


    // END-STUDENTS-TO-ADD-CODE
}
```

5.4.3 4.3 Teilaufgabe test_model_can_move

Gehen Sie analog den obigen Teilaufgaben vor und implementieren Sie, gemäss Vorgaben im Code, die Funktion **model_can_move()**.

```
int model_can_move(model_t *instance)
{
    assert(instance);
    if (model_get_winner(instance) == model_state_none) {
        // Instructions to the students:
        // scan all fields: return 1 with first field which equals model_state_none
        // BEGIN-STUDENTS-TO-ADD-CODE



        // END-STUDENTS-TO-ADD-CODE
    }
    return 0;
}
```

5.4.4 4.4 Teilaufgabe test_model_move und test_model_get_win_line

Schliesslich gehen Sie auch hier analog den obigen Teilaufgaben vor und implementieren Sie, gemäss Vorgaben im Code, die Funktion **set_state()**.

```
/**
 * @brief                   Sets the field on the board to the given state.
 * @param  instance [INOUT]  The instance which holds the state.
 * @param  pos      [IN]     The affected field.
 * @param  state    [IN]     The new state of the field.
 */
static void set_state(model_t *instance, model_pos_t pos, model_state_t state)
{
    assert_pos(pos);

    // Instructions to the students:
    // set the field of the board to the new state
    // BEGIN-STUDENTS-TO-ADD-CODE




    // END-STUDENTS-TO-ADD-CODE
}
```

Wenn die beiden obigen Teilaufgaben erfolgreich umgesetzt sind, laufen die Tests ohne Fehler durch und das Spiel kann gespielt werden.

5.5 5. Bewertung

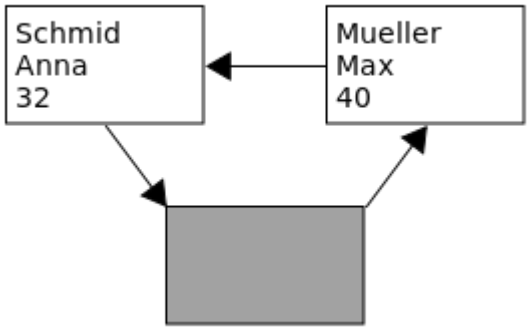
Der funktionierende Programmcode muss der Praktikumsbetreuung gezeigt werden. Die Lösungen müssen mündlich erklärt werden.

Aufgabe	Kriterium	Punkte
Sortieren von Strings	Sie können das funktionierende Programm demonstrieren und erklären.	2
TicTacToe	Sie können das funktionierende Programm inklusive funktionierende Tests demonstrieren und erklären.	
TicTacToe	Teilaufgabe test_model_init	0.5
TicTacToe	Teilaufgabe test_model_get_state und test_model_get_winner	0.5
TicTacToe	Teilaufgabe test_model_can_move	0.5
TicTacToe	Teilaufgabe test_model_move und test_model_get_win_line	0.5

Version: 14.02.2022

Chapter 6

06 - Personen Verwaltung – Linked List



6.1 1. Übersicht

In diesem Praktikum schreiben Sie eine einfache Personenverwaltung. Dabei werden Sie etliche Elemente von C anwenden:

- Header Files selber schreiben, inklusive Include Guard
- Typen definieren
- Funktionen mit `by value` und `by reference` Parametern deklarieren und definieren
- einfache Variablen, Pointer Variablen, struct Variablen und Array Variablen benutzen
- Strukturen im Speicher dynamisch allozieren und freigeben
- I/O und String Funktionen aus der Standard Library anwenden
- Anwender Eingaben verarbeiten
- Fehlerbehandlung

6.2 2. Lernziele

In diesem Praktikum wenden Sie viele der bisher gelernten C Elemente an.

- Sie können anhand dieser Beschreibung ein vollständiges C Programm schreiben.
- Sie können Unit Tests schreiben welche die wesentlichen Funktionen des Programms individuell testen.
-

Die Bewertung dieses Praktikums ist am Ende angegeben.

Erweitern Sie die vorgegebenen Code Gerüste, welche im `git` Repository `snp-lab-code` verfügbar sind.

6.3 3. Personenverwaltung

6.3.1 3.1 Programmfunktion

Das Programm soll in einer Schleife dem Benutzer jeweils folgende Auswahl bieten, wovon eine Aktion mit Eingabe des entsprechenden Buchstabens ausgelöst wird:

I(nsert), **R**(emove), **S**(how), **C**(lear), **E**(nd):

- **Insert**: der Benutzer wird aufgefordert, eine Person einzugeben
- **Remove**: der Benutzer wird aufgefordert, die Daten einer zu löschenden Person einzu-geben
- **Show**: eine komplette Liste aller gespeicherten Personen wird in alphabetischer Rei-henfolge ausgegeben
- **Clear**: alle Personen werden gelöscht
- **End**: das Programm wird beendet

6.3.2 3.2 Designvorgaben

Verkettete Liste Da zur Kompilierzeit nicht bekannt ist, ob 10 oder 10'000 Personen eingegeben werden, wäre es keine gute Idee, im Programm einen statischen Array mit z.B. 10'000 Personen-Einträgen zu allozieren. Dies wäre ineffizient und umständlich beim sortierten Einfügen von Personen. In solchen Situationen arbeitet man deshalb mit dynamischen Datenstrukturen, die zur Laufzeit beliebig (solange Speicher vorhanden ist) wachsen und wieder schrumpfen können. Eine sehr populäre dynamische Datenstruktur ist die **verkettete Liste** und genau die werden wir in diesem Praktikum verwenden.

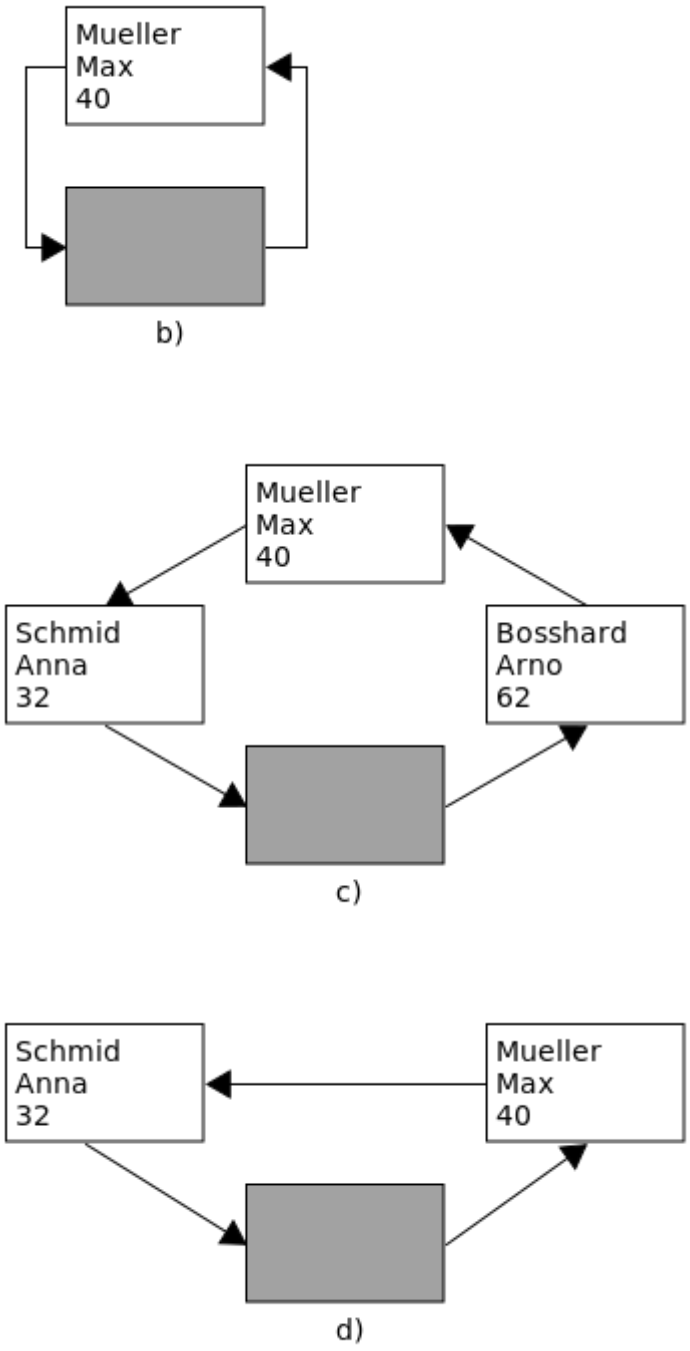


Abbildung 1: Zyklisch verkettete Liste

Eine verkettete Liste bedeutet, dass ein Knoten der verketten Liste einen Datensatz einer Person speichert und zusätzlich einen Pointer auf den nächsten Knoten in der Liste aufweist (siehe Abbildung 1). In dieser Pointer Variablen (`next` in der `node_t` Struktur unten) steht also einfach die Adresse des nächsten Knotens.

Die leere Liste besteht aus einem einzelnen Element, welches keine spezifische Person abspeichert und welches auf sich selbst zeigt (Abbildung 1 a). Dieses Element ist der Einstiegspunkt der Liste (auch Anker oder Wurzel genannt) und ist das einzige Element, das Sie im Programm direkt kennen und einer Variablen zuweisen. Dieses Element können Sie statisch allozieren (z.B. `node_t anchor;`, siehe Details weiter unten), denn es existiert während der gesamten Ausführungszeit. Alle anderen Elemente erreichen Sie ausgehend vom Anker, indem Sie einmal, den Pointern folgend, im Kreis herum gehen. Abbildung 1 b zeigt die Liste nach dem Einfügen der Person `Max Mueller, 40` Jahre. Nach dem Einfügen von zwei weiteren Personen sieht die Datenstruktur aus wie in Abbildung 1 c. Das Entfernen der Person `Arno Bosshard` führt zu Abbildung 1 d.

Eine Person kann **zugefügt** werden, indem dynamisch ein neuer Knoten erzeugt wird und dieser in die verkettete Liste eingefügt wird. Beim Einfügen müssen die Adressen der Knoten so den Pointern zugewiesen werden, dass die Kette intakt bleibt.

Ein Knoten wird **entfernt**, indem der entsprechende Knoten aus der Verkettung herausgelöst wird (`next` des Vorgängerknotens soll neu auf `next` des herauszulösenden Knotens zeigen) und dann der Speicher des entsprechenden Knotens freigegeben wird.

Personen und Knoten Records

Die für je eine Person zu speichernden Daten sollen in folgendem C `struct` zusammengefasst sein.

```
#define NAME_LEN 20

typedef struct {
  char         name[NAME_LEN];
  char         first_name[NAME_LEN];
  unsigned int age;
} person_t;
```

Jeder Knoten der verketteten Liste soll aus folgendem C `struct` bestehen.

```
typedef struct node {
  person_t     content;        // in diesem Knoten gespeicherte Person
  struct node *next;           // Pointer auf den nächsten Knoten in der Liste
} node_t;
```

Vorschlag: zyklisch verkettete Liste

Erkennen des Endes der Liste: bei der zyklisch verketteten Liste zeigt das letzte Element wie-der auf den Anker, die Liste bildet also einen Kreis. Dies ist in Abbildung 1 so abgebildet.

Alternativ könnte man das Ende erkennbar machen, indem die Kette anstelle von zyklisch, mit einem NULL Pointer endet.

Die Wahl ist ihnen überlassen ob sie die eine oder andere Art der End-Erkennung implementieren. In der Beschreibung wird angenommen, dass es sich um eine zyklisch verkettete Liste handelt.

Sortiertes Einfügen

Die Personen Records sollen sortiert in die Liste eingefügt werden. Dies bedeutet, dass vom Anker her gesucht werden soll, bis der erste Knoten gefunden wurde dessen Nachfolgeknoten entweder „grösser“ ist als der einzufügende Knoten, oder wo das Ende der Liste erreicht ist. Die Ordnung (grösser, gleich, kleiner) soll so definiert sein:

```
// if (p1 > p2) { ... }
if (person_compare(&p1, &p2) > 0) { ... }
/**
 * @brief  Compares two persons in this sequence: 1st=name, 2nd=first_name, 3rd=age
 * @param  a [IN] const reference to 1st person in the comparison
 * @param  b [IN] const reference to 2nd person in the comparison
 * @return =0 if all record fields are the same
 *         >0 if all previous fields are the same, but for this field, a is greater
 *         <0 if all previous fields are the same, but for this field, b is greater
 * @remark strncmp() is used for producing the result of string field comparisons
 * @remark a->age - b->age is used for producing the result of age comparison
 */
int person_compare(const person_t *a, const person_t *b);
```

Eingabe

Fehlerhafte Wahl der Operation in der Hauptschleife soll gemeldet werden, ansonsten aber ignoriert werden.

Fehlerhafte Eingabe der Personenangaben sollen gemeldet werden und die gesamte Operation (z.B. Insert) verworfen werden.

Zu prüfende Fehler bei Personeneingaben:

- für die Namen
  - zu lange Namen
- für das Alter
  - keine Zahl
- Duplikat
  - derselbe Record soll nicht doppelt in der Liste vorkommen

Weitergehende Prüfungen sind nicht erwartet.

Zu beachten: bei fehlerhafter Eingabe darf kein „Memory Leak“ entstehen, d.h. potentiell auf dem Heap allozierter Speicher muss im Fehlerfall freigegeben werden.

6.3.3 3.3 Bestehender Programmrahmen

Der Programmrahmen besteht aus den unten aufgelisteten Files. Es sollen weitere Module in `src` hinzugefügt werden und die bestehenden Files ergänzt werden gemäss den Aufgaben.

Makefile	-> **zu ergänzen** mit neuen Modulen
tests/tests.c	-> **zu ergänzen** gemäss Aufgaben (implementieren von Unit Tests)
src/main.c	-> **zu ergänzen** gemäss Aufgaben (Hauptprogramm)

6.4 4. Aufgabe 1: Modularisierung – API und Implementation main.c

Kreieren Sie folgende Files in `src` und implementieren Sie `main.c` basierend auf dem unten von Ihnen gegebenen API.

File person.h

Typ Definitionen:

```
person_t... // siehe Beschreibung oben
```

Funktionsdeklarationen:

```
// siehe Beschreibung oben
int person_compare(const person_t *a, const person_t *b);
```

- gegebenenfalls weitere Funktionen für die Bearbeitung von Personen

File list.h

Typ Definitionen:

```
person_t... // siehe Beschreibung oben
```

Funktionsdeklarationen:

- Funktionen für `insert`, `remove`, `clear` Operationen auf der Liste

Das Hauptprogramm soll die Eingabeschleife implementieren und die obigen Funktionen (wo angebracht) aufrufen.

6.5 5. Aufgabe 2: Implementierung von person.c und list.c

Fügen Sie die beiden Implementationsfiles `person.c` und `list.c` zu `src`. Fügen Sie die beiden Module im `Makefile` zu der vorgegebenen Variablen `MODULES` hinzu, so dass sie beim `make` Aufruf auch berücksichtigt werden.

6.5.1 5.1 Teilaufgabe: Implementierung von person.c

Implementieren Sie die Funktionen aus `person.h`.

Falls nötig, stellen Sie weitere statische Hilfsfunktionen in `person.c` zur Verfügung.

6.5.2 5.2 Teilaufgabe: Implementierung von list.c

Implementieren Sie die Funktionen aus `list.h`.

Falls nötig, stellen Sie weitere statische Hilfsfunktionen in `list.c` zur Verfügung.

6.6 6. Aufgabe 3: Unit Tests

Schreiben Sie Unit Tests für mindestens die folgenden Funktionen

- `person.h`:
  - `person_compare`
- `list.h`:
  - `list_insert`
  - `list_remove`
  - `list_clear`

Es existieren in `tests/tests.c` schon vier Test Rahmen für diese Test Cases.

In diese Test Cases sollen die entsprechenden Funktionen unter verschiedenen Bedingungen isoliert aufgerufen werden und deren Verhalten überprüft werden.

Verwenden Sie für die Überprüfung die CUnit `CU_ASSERT_...` Makros.

Siehe dazu auch `man CUnit`.

Wenn die obigen Teilaufgaben erfolgreich umgesetzt sind, laufen die Tests ohne Fehler durch.

6.7 7. Bewertung

Aufgabe	Kriterium	Punkte
	Sie können das funktionierende Programm inklusive funktionierende Tests demonstrieren und erklären.	
1	API von list.h und person.h plus die Implementation von main.c	2
2	Teilaufgabe: person.c	2
2	Teilaufgabe: list.c	2
3	Unit Tests	2

Version: 11.01.2022

Chapter 7

07 - Prozesse und Threads

Quelle: https://www.wikiwand.com/de/Ein-Mann-Orchester

7.1 1. Übersicht

In diesem Praktikum werden wir uns mit Prozessen, Prozesshierarchien und Threads beschäftigen, um ein gutes Grundverständnis dieser Abstraktionen zu erhalten. Sie werden bestehenden Code analysieren und damit experimentieren. D.h. dies ist nicht ein «Codierungs»-Praktikum, sondern ein «Analyse»- und «Experimentier»-Praktikum.

7.1.1 1.1 Nachweis

Dieses Praktikum ist eine leicht abgewandelte Variante des ProcThreads Praktikum des Moduls BSY, angepasst an die Verhältnisse des SNP Moduls. Die Beispiele und Beschreibungen wurden, wo möglich, eins-zu-ein übernommen.

Als Autoren des BSY Praktikums sind genannt: M. Thaler, J. Zeman.

7.2 2. Lernziele

In diesem Praktikum werden Sie sich mit Prozessen, Prozesshierarchien und Threads beschäftigen. Sie erhalten einen vertieften Einblick und Verständnis zur Erzeugung, Steuerung und Terminierung von Prozessen unter Unix/Linux und Sie werden die unterschiedlichen Eigenschaften von Prozessen und Threads kennenlernen.

- Sie können Prozesse erzeugen und die Prozesshierarchie erklären
- Sie wissen was beim Erzeugen eines Prozesses vom Elternprozess vererbt wird
- Sie wissen wie man auf die Terminierung von Kindprozessen wartet
- Sie kennen die Unterschiede zwischen Prozessen und Threads

7.3 3. Aufgaben

Das Betriebssystem bietet Programme um die aktuellen Prozesse und Threads darzustellen.

Die Werkzeuge kommen mit einer Vielzahl von Optionen für die Auswahl und Darstellung der Daten, z.B. ob nur Prozesse oder auch Threads aufgelistet werden sollen, und ob alle Prozesse oder nur die «eigenen» Prozesse ausgewählt werden sollen, etc.

Siehe die entsprechenden `man` Pages für weitere Details.

Eine Auswahl, welche unter Umständen für die folgenden Aufgaben nützlich sind:

`ps`	Auflisten der Prozess Zustände zum gegebenen Zeitpunkt.
`pstree`	Darstellung der gesamten Prozesshierarchie.
`top`	Wie `ps`, aber die Darstellung wird in Zeitintervallen aufdatiert.
`htop`	Wie `top`, aber zusätzlich dazu die Auslastung der CPU in einem System mit mehreren CPUs.
`lscpu`	Auflisten der CPUs.
`cat/proc/cpuinfo`	Ähnlich zu `lscpu`, aber mit Zusatzinformationen wie enthaltene CPU Bugs (z.B. `bugs: cpu_meltdown spectre_v1 spect-re_v2 spec_store_bypass l1tf mds swapgs itlb_multihit`)

7.3.1 3.1 Aufgabe 1: Prozess mit fork() erzeugen

Ziele

- Verstehen, wie mit `fork()` Prozesse erzeugt werden.
- Einfache Prozesshierarchien kennenlernen.
- Verstehen, wie ein Programm, das `fork()` aufruft, durchlaufen wird.

Aufgaben

1. Studieren Sie zuerst das Programm `ProcA1.c` und beschrieben Sie was geschieht.

2. Notieren Sie sich, was ausgegeben wird. Starten Sie das Programm und vergleichen Sie die Ausgabe mit ihren Notizen? Was ist gleich, was anders und wieso?

7.3.2 3.2 Aufgabe 2: Prozess mit fork() und exec(): Programm Image ersetzen

Ziele

- An einem Beispiel die Funktion `execl()` kennenlernen.
- Verstehen, wie nach `fork()` ein neues Programm gestartet wird. **Aufgaben**

1. Studieren Sie zuerst die Programme `ProcA2.c` und `ChildProcA2.c`.
2. Starten Sie `ProcA2.e` und vergleichen Sie die Ausgabe mit der Ausgabe unter Aufgabe 1. Diskutieren und erklären Sie was gleich ist und was anders.

3. Benennen Sie `ChildProcA2.e` auf `ChildProcA2.f` um (Shell Befehl `mv`) und überlegen Sie, was das Programm nun ausgibt. Starten Sie `ProcA2.e` und vergleichen Sie Ihre Überlegungen mit der Programmausgabe.

4. Nennen Sie das Kindprogramm wieder `ChildProcA2.e` und geben Sie folgenden Befehl ein: `chmod -x ChildProcA2.e`. Starten Sie wiederum `ProcA2.e` und analysieren Sie die Ausgabe von `perror("...")`. Wieso verwenden wir `perror()`?

7.3.3 3.3 Aufgabe 3: Prozesshierarchie analysieren

Ziele

- Verstehen, was `fork()` wirklich macht.
- Verstehen, was Prozesshierarchien sind.

Aufgaben

1. Studieren Sie zuerst Programm `ProcA3.c` und zeichnen Sie die entstehende Prozesshierarchie (Baum) von Hand auf. Starten Sie das Programm und verifizieren Sie ob Ihre Prozesshierarchie stimmt.
2. Mit dem Befehl `ps f` oder `pstree` können Sie die Prozesshierarchie auf dem Bildschirm ausgeben. Damit die Ausgabe von `pstree` übersichtlich ist, müssen Sie in dem Fenster, wo Sie das Programm `ProcA3.e` starten, zuerst die PID der Shell erfragen, z.B. über `echo $$`. Wenn Sie nun den Befehl `pstree -n -p pid-von-oben` eingeben, wird nur die Prozesshierarchie ausgehend von der Bash Shell angezeigt: `-n` sortiert die Prozesse numerisch, `-p` zeigt für jeden Prozess die PID an.

Hinweis: alle erzeugten Prozesse müssen arbeiten (d.h. nicht terminiert sein), damit die Darstellung gelingt. Wie wird das im gegebenen Programm erreicht?

7.3.4 3.4 Aufgabe 4: Zeitlicher Ablauf von Prozessen

Ziele

- Verstehen, wie Kind- und Elternprozesse zeitlich ablaufen.

Aufgaben

1. Studieren Sie Programm `ProcA4.c`. Starten Sie nun mehrmals hintereinander das Programm `ProcA4.e` und vergleichen Sie die jeweiligen Outputs (leiten Sie dazu auch die Ausgabe auf verschiedene Dateien um). Was schliessen Sie aus dem Resultat?

Anmerkung: Der Funktionsaufruf `selectCPU(0)` erzwingt die Ausführung des Eltern- und Kindprozesses auf CPU 0 (siehe Modul `setCPU.c`). Die Prozedur `justWork(HARD_WORK)` simuliert CPU-Load durch den Prozess (siehe Modul `workerUtils.c`).

7.3.5 3.5 Aufgabe 5: Waisenkinder (Orphan Processes)

Ziele

- Verstehen, was mit verwaisten Kindern geschieht.

Aufgaben

1. Analysieren Sie Programm `ProcA5.c`: was läuft ab und welche Ausgabe erwarten Sie?
2. Starten Sie `ProcA5.e`: der Elternprozess terminiert: was geschieht mit dem Kind?
3. Was geschieht, wenn der Kindprozess vor dem Elternprozess terminiert? Ändern Sie dazu im `sleep()` Befehl die Zeit von 2 Sekunden auf 12 Sekunden und verfolgen Sie mit top das Verhalten der beiden Prozesse, speziell auch die Spalte S.

7.3.6 3.6 Aufgabe 6: Terminierte, halbtote Prozesse (Zombies)

Ziele

- Verstehen, was ein Zombie ist.
- Eine Möglichkeit kennenlernen, um Zombies zu verhindern.

Aufgaben

1. Analysieren Sie das Programm `ProcA6.c`.
2. Starten Sie das Script `mtop` bzw. `mtop aaaa.e`. Es stellt das Verhalten der Prozesse dynamisch dar.
   Hinweis: `<defunct>` = Zombie.
3. Starten Sie `aaaa.e` und verfolgen Sie im `mtop`-Fenster was geschieht. Was beachten Sie?

4. In gewissen Fällen will man nicht auf die Terminierung eines Kindes mit `wait()`, bzw. `waitpid()` warten. Überlegen Sie sich, wie Sie in diesem Fall verhindern können, dass ein Kind zum Zombie wird.

7.3.7 3.7 Aufgabe 7: Auf Terminieren von Kindprozessen warten

Vorbemerkung: Diese Aufgabe verwendet Funktionen welche erst in der Vorlesung über *Inter-Process-Communication (IPC)* im Detail behandelt werden.

Sie können diese Aufgabe bis dann aufsparen oder die verwendeten Funktionen selber via `man` Pages im benötigten Umfang kennenlernen: `man 2 kill` und `man 7 signal`.

Ziele

- Verstehen, wie Informationen zu Kindprozessen abgefragt werden können.
- Die Befehle `wait()` und `waitpid()` verwenden können.

Aufgaben

1. Starten Sie das Programm `ProcA7.e` und analysieren Sie wie die Ausgabe im Hauptprogramm zustande kommt und was im Kindprozess `ChildProcA7.c` abläuft.

2. Starten Sie `ProcA7.e` und danach nochmals mit `1` als erstem Argument. Dieser Argument Wert bewirkt, dass im Kindprozess ein "Segmentation Error" erzeugt wird, also eine Speicherzugriffsverletzung. Welches Signal wird durch die Zugriffsverletzung an das Kind geschickt? Diese Information finden Sie im Manual mit `man 7 signal`. Schalten Sie nun core dump ein (siehe README) und starten Sie `ProcA7.e 1` erneut und analysieren Sie die Ausgabe.

Hinweis: ein core Dump ist ein Abbild des Speichers z.B. zum Zeitpunkt, wenn das Programm abstürzt (wie oben mit der Speicher Zugriff Verletzung). Der Dump wird im File **core** abgelegt und kann mit dem **gdb** (GNU-Debugger) gelesen werden (siehe `README`). Tippen Sie nach dem Starten des Command Line UI des `gdb where` gefolgt von list ein, damit sie den Ort des Absturzes sehen. Mit `quit` verlassen Sie **gdb** wieder.

1. Wenn Sie `ProcA7.e 2` starten, sendet das Kind das Signal 30 an sich selbst. Was geschieht?

2. Wenn Sie `ProcA7.e 3` starten, sendet ProcA7.e das Signal SIGABRT (abort) an das Kind: was geschieht in diesem Fall?

3. Mit `ProcA7.e 4` wird das Kind gestartet und terminiert nach 5 Sekunden. Analysieren Sie wie in ProcA7.e der Lauf- bzw. Exit-Zustand des Kindes abgefragt wird (siehe dazu auch `man 3 exit`).

7.3.8 3.8 Aufgabe 8: Kindprozess als Kopie des Elternprozesses

Ziele

- Verstehen, wie Prozessräume vererbt werden.
- Unterschiede zwischen dem Prozessraum von Eltern und Kindern erfahren.

Aufgaben

1. Analysieren Sie Programm `ProcA8_1.c`: was gibt das Programm aus?
   - Starten Sie `ProcA8_1.e` und überprüfen Sie Ihre Überlegungen.
   - Waren Ihre Überlegungen richtig? Falls nicht, was könnten Sie falsch überlegt haben?

2. Analysieren Sie Programm `ProcA8_2.c`: was gibt das Programm aus?
   - Starten Sie `ProcA8_2.e` und überprüfen Sie Ihre Überlegungen.
   - Waren Ihre Überlegungen richtig? Falls nicht, was könnten Sie falsch gemacht haben?
   - Kind und Eltern werden in verschiedener Reihenfolge ausgeführt: ist ein Unterschied ausser der Reihenfolge festzustellen?

3. Analysieren Sie Programm `ProcA8_3.c` und Überlegen Sie, was in die Datei `AnyOutPut.txt` geschrieben wird, wer schreibt alles in diese Datei (sie wird ja vor `fork()` geöffnet) und wieso ist das so?
   - Starten Sie `ProcA8_3.e` und überprüfen Sie Ihre Überlegungen.
   - Waren Ihre Überlegungen richtig? Falls nicht, wieso nicht?

7.3.9 3.9 Aufgabe 9: Unterschied von Threads gegenüber Prozessen

Ziele

- Den Unterschied zwischen Thread und Prozess kennenlernen.
- Problemstellungen um Threads kennenlernen.
- Die `pthread`-Implementation kennen lernen.

Aufgaben

1. Studieren Sie Programm `ProcA9.c` und überlegen Sie, wie die Programmausgabe aussieht. Vergleichen Sie Ihre Überlegungen mit denjenigen aus Aufgabe 8.2 b) (`Pro-cA8_2.e`).
   - Starten Sie `ProcA9.e` und vergleichen das Resultat mit Ihren Überlegungen.
   - Was ist anders als bei `ProcA8_2.e`?

1. Setzen Sie in der Thread-Routine vor dem Befehl `pthread_exit()` eine unendliche Schleife ein, z.B. `while(1) { };` .
   - Starten Sie das Programm und beobachten Sie das Verhalten mit `top`. Was beobachten Sie und was schliessen Sie daraus?
     Hinweis: wenn Sie in `top` den Buchstaben H eingeben, werden die Threads einzeln dargestellt.
   - Kommentieren Sie im Hauptprogram die beiden `pthread_join()` Aufrufe aus und starten Sie das Programm. Was geschieht? Erklären Sie das Verhalten.

7.3.10 3.10 Aufgabe 10 (optional):

7.3.10.1 3.10.1 Übersicht

Dieser Teil des Praktikums behandelt spezielle Prozesse: die Dämon Prozesse («daemon pro-cesses»). Es ist gedacht als Zusatz zum Basis Praktikum über Prozesse und Threads.

Auch dieser Teil ist ein «Analyse»- und «Experimentier»-Praktikum.

3.10.1.1 Nachweis

Dieses Praktikum ist eine leicht abgewandelte Variante des ProcThreads Praktikum des Moduls BSY, angepasst an die Verhältnisse des SNP Moduls. Die Beispiele und Beschreibungen wurden, wo möglich, eins-zu-ein übernommen.

Als Autoren des BSY Praktikums sind genannt: M. Thaler, J. Zeman.

7.3.10.2 3.10.2 Lernziele

In diesem Praktikum werden Sie sich mit Dämon Prozessen beschäftigen.

- Sie können die Problemstellung der Dämon Prozesse erklären
- Sie können einen Dämon Prozess kreieren
- Sie können aus dem Dämon Prozess mit der Umgebung kommunizieren
-

7.3.10.3 3.10.3 Aufgabe: Dämon Prozesse

Ziele

- Problemstellungen um Daemons kennenlernen:
  - wie wird ein Prozess zum Daemon?
  - wie erreicht man, dass nur ein Daemon vom gleichen Typ aktiv ist?
  - wie teilt sich ein Daemon seiner Umwelt mit?
  - wo "lebt" ein Daemon?

Einleitung

Für diese Aufgabe haben wir einen Daemon implementiert: **MrTimeDaemon** gibt auf Anfrage die Systemzeit Ihres Rechners bekannt. Abfragen können Sie diese Zeit mit dem Programm `WhatsTheTimeMr localhost`. Die Kommunikation zwischen den beiden Prozessen haben wir mit TCP/IP Sockets implementiert. Weitere Infos zum Daemon finden Sie nach den Aufgaben.

Im Abschnitt 4 finden Sie Zusatzinformationen über diese Implementation eines Dämon Prozesses plus weiterführende Informationen.

Aufgaben

1. Für die folgende Aufgabe benötigen Sie mindestens zwei Fenster (Kommandozeilen-Konsolen). Übersetzen Sie die Programme mit `make` und starten Sie das Programm **PlapperMaul** in einem der Fenster. Das Programm schreibt (ca.) alle 0.5 Sekunden *Hallo, ich bins. . . . Pidi* plus seine Prozess-ID auf den Bildschirm. Mit dem Shell Befehl `ps` können Sie Ihre aktiven Prozesse auflisten, auch **PlapperMaul**. Überlegen Sie sich zuerst, was mit **PlapperMaul** geschieht, wenn Sie das Fenster schliessen: läuft **PlapperMaul** weiter? Was geschieht mit **PlapperMaul** wenn Sie sich ausloggen und wieder einloggen? Testen Sie Ihre Überlegungen, in dem Sie die entsprechenden Aktionen durchführen. Stimmen Ihre Überlegungen?

2. Starten Sie nun das Programm bzw. den Daemon **MrTimeDaemon**. Stellen Sie die gleichen Überlegungen an wie mit **PlapperMaul** und testen Sie wiederum, ob Ihre Überlegungen stimmen. Ob **MrTimeDaemon** noch läuft können Sie feststellen, indem Sie die Zeit abfragen oder den Befehl `ps ajx | grep MrTimeDaemon` eingeben: was fällt Ihnen am Output auf? Was schliessen Sie aus Ihren Beobachtungen?
3. Starten Sie **MrTimeDaemon** erneut, was geschieht?
4. Stoppen Sie nun **MrTimeDaemon** mit `killall MrTimeDaemon`.
5. Starten Sie **MrTimeDaemon** und fragen Sie mit `WhatsTheTimeMr localhost` oder mit `WhatsTheTimeMr 127.0.0.1` die aktuelle Zeit auf Ihrem Rechner ab.
   Optional: Fragen Sie die Zeit bei einem Ihrer Kollegen ab. Dazu muss beim Server (dort wo **MrTimeDaemon** läuft) ev. die Firewall angepasst werden. Folgende Befehle müssen dazu mit **root-Privilegien** ausgeführt werden:

   ```
   iptables-save > myTables.txt # sichert die aktuelle Firewall
   iptables -I INPUT 1 -p tcp --dport 65534 -j ACCEPT
   iptables -I OUTPUT 2 -p tcp --sport 65534 -j ACCEPT
   ```

   Nun sollten Sie über die IP-Nummer oder über den Rechner-Namen auf den **TimeServer** mit `WhatsTheTimeMr` zugreifen können. Die Firewall können Sie mit folgendem Befehl wiederherstellen:

   ```
   iptables-restore myTables.txt
   ```

6. Studieren Sie `MrTimeDaemon.c`, `Daemonizer.c` und `TimeDaemon.c` und analysieren Sie, wie die Daemonisierung abläuft. Entfernen Sie die Kommentare im Macro `Out-PutPIDs` am Anfang des Moduls `Daemonizer.c`. Übersetzen Sie die Programme mit make und starten Sie `MrTimeDaemon` erneut. Analysieren Sie die Ausgabe, was fällt Ihnen auf? Notieren Sie alle für die vollständige Daemonisierung notwendigen Schritte.
7. Setzen Sie beim Aufruf von `Daemonizer()` in `MrTimeDaemon.c` anstelle von `lock-FilePath` den Null-Zeiger `NULL` ein. Damit wird keine lock-Datei erzeugt. Übersetzen Sie die Programme und starten Sie erneut `MrTimedaemon`. Was geschieht bzw. wie können Sie feststellen, was geschehen ist?
   Hinweis: lesen Sie das log-File: `/tmp/timeDaemon.log`.

   Wenn Sie noch Zeit und Lust haben: messen Sie die Zeit, zwischen Start der Zeitanfrage und Eintreffen der Antwort. Dazu müssen Sie die Datei `WhatsTheTimeMr.c` entsprechend anpassen.

7.3.10.4 3.10.4 Zusatzinformationen

3.10.4.1 Diese Implementation

Dieser Daemon besteht aus den 3 Komponenten.

Hauptprogramm: MrTimeDaemon.c

Hier werden die Pfade für die lock-Datei, die log-Datei und der "Aufenthaltsort" des Daemons gesetzt. Die lock-Datei wird benötigt um sicherzustellen, dass der Daemon nur einmal gestartet werden kann. In die lock-Datei schreibt der Daemon z.B. seine PID und sperrt sie dann für Schreiben. Wird der Daemon ein zweites Mal gestartet und will seine PID in diese Datei schreiben, erhält er eine Fehlermeldung und terminiert (es soll ja nur ein Daemon arbeiten). Terminiert der Daemon, wird die Datei automatisch freigegeben. Weil Daemonen sämtliche Kontakte

mit ihrer Umwelt im Normalfall abbrechen und auch kein Kontrollterminal besitzen, ist es sinnvoll, zumindest die Ausgabe des Daemons in eine log-Datei umzuleiten. Dazu stehen einige Systemfunktionen für Logging zur Verfügung. Der Einfachheit halber haben wir hier eine normale Datei im Verzeichnis `/tmp` gewählt.

> **Anmerkung:** die Wahl des Verzeichnisses `/tmp` für die lock- und log-Datei ist für den normalen Betrieb problematisch, weil der Inhalt dieses Verzeichnisses jederzeit gelöscht werden kann, bzw. darf. Wir haben dieses Verzeichnis gewählt, weil wir die beiden Dateien nur für die kurze Zeit des Praktikums benötigen.

Der Daemon erbt sein Arbeitsverzeichnis vom Elternprozesse, er sollte deshalb in ein festes Verzeichnis des Systems wechseln, um zu verhindern, dass er sich in einem montierten (gemounteten) Verzeichnis aufhält, das dann beim Herunterfahren nicht demontiert werden könnte (wir haben hier wiederum `/tmp` gewählt).

Daemonizer: Daemonizer.c

Der Daemonizer macht aus dem aktuellen Prozess einen Daemon. Z.B. sollte er Signale (eine Art Softwareinterrupts) ignorieren: wenn Sie die CTRL-C Taste während dem Ausführen eines Vordergrundprozess drücken, erhält dieser vom Betriebssystem das Signal SIGINT und bricht seine Ausführung ab. Weiter sollte er die Dateierzeugungsmaske auf 0 setzen (Dateizugriffsrechte), damit kann er beim Öffnen von Dateien beliebige Zugriffsrechte verlangen (die Dateierzeugungsmaske erbt er vom Elternprozess). Am Schluss startet der Daemonizer das eigentliche Daemonprogramm: TimeDaemon.e.

Daemonprogramm: TimeDaemon.c

Das Daemonprogramm wartet in einer unendlichen Schleife auf Anfragen zur Zeit und schickt die Antwort an den Absender zurück. Die Datenkommunikation ist, wie schon erwähnt, mit Sockets implementiert, auf die wir aber im Rahmen dieses Praktikums nicht weiter eingehen wollen (wir stellen lediglich Hilfsfunktionen zur Verfügung).

3.10.4.2 Zusatzinformation zu Dämon Prozessen

Dämonen oder englisch Daemons sind eine spezielle Art von Prozessen, die vollständig unabhängig arbeiten, d.h. ohne direkte Interaktion mit dem Anwender. Dämonen sind Hintergrundprozesse und terminieren i.A. nur, wenn das System heruntergefahren wird oder abstürzt. Dämonen erledigen meist Aufgaben, die periodisch ausgeführt werden müssen, z.B. Überwachung von Systemkomponenten, abfragen, ob neue Mails angekommen sind, etc.

Ein typisches Beispiel unter Unix ist der Printer Daemon `lpd`, der periodisch nachschaut, ob ein Anwender eine Datei zum Ausdrucken hinterlegt hat. Wenn ja, schickt er die Datei auf den Drucker.

Hier wird eine weitere Eigenschaft von Daemons ersichtlich: meist kann nur ein Dämon pro Aufgabe aktiv sein: stellen Sie sich vor, was passiert, wenn zwei Druckerdämonen gleichzeitig arbeiten. Andererseits muss aber auch dafür gesorgt werden, dass ein Dämon wieder gestartet wird, falls er stirbt.

7.4 4. Bewertung

Die gegebenenfalls gestellten Theorieaufgaben und der funktionierende Programmcode müssen der Praktikumsbetreuung gezeigt werden. Die Lösungen müssen mündlich erklärt werden.

Aufgabe	Kriterium	Punkte
	Sie können die gestellten Fragen erklären.	
1	Prozess mit `fork()` erzeugen	0.5
2	Prozess mit `fork()` und `exec()`: Programm Image ersetzen	0.5
3	Prozesshierarchie analysieren	0.5
4	Zeitlicher Ablauf von Prozessen	0.5
5	Waisenkinder (Orphan Processes)	0.5
6	Terminierte, halbtote Prozesse (Zombies)	0.5
7	Auf Terminieren von Kindprozessen warten	0.5
8	Kindprozess als Kopie des Elternprozesses	0.5
9	Unterschied von Threads gegenüber Prozessen	0.5
10	Dämon Prozesse	(4)

Version: 11.01.2022

Chapter 8

08 - Synchronisationsprobleme

8.1 1. Übersicht

Quelle: https://commons.wikimedia.org/wiki/File:Velgast-suedbahn.jpg

In diesem Praktikum lernen sie zuerst am Beispiel eines Kaffee-Automaten verschiedene grundlegende Synchronisationsprobleme kennen und mit Hilfe von Locks (Mutexes) und Semaphoren lösen:

- gegenseitiger Ausschluss mit einem Lock

- Erzwingen einer einfachen Reihenfolge
- Erzwingen einer erweiterten Reihenfolge

Im zweiten Teil werden sie auf Basis dieser Grundlagen ein komplexeres Synchronisationsproblem bearbeiten, diesmal am Beispiel von Bank Transaktionen.

8.1.1 1.1 Nachweis

Dieses Praktikum ist eine leicht abgewandelte Variante des Sync Praktikum des Moduls BSY, angepasst an die Verhältnisse des SNP Moduls. Die Beispiele und Beschreibungen wurden, wo möglich, eins-zu-ein übernommen.

Als Autor des BSY Praktikums ist genannt: M. Thaler.

8.2 2. Lernziele

In diesem Praktikum werden sie Synchronisationsprobleme lösen

- Sie wissen wie man systematisch Synchronisationsprobleme analysiert
- Sie wissen wann ein potentieller Deadlock entstehen kann
- Sie können Mutex mit Threads anwenden
- Sie können Semaphoren mit Prozessen anwenden

8.3 3. Einführung

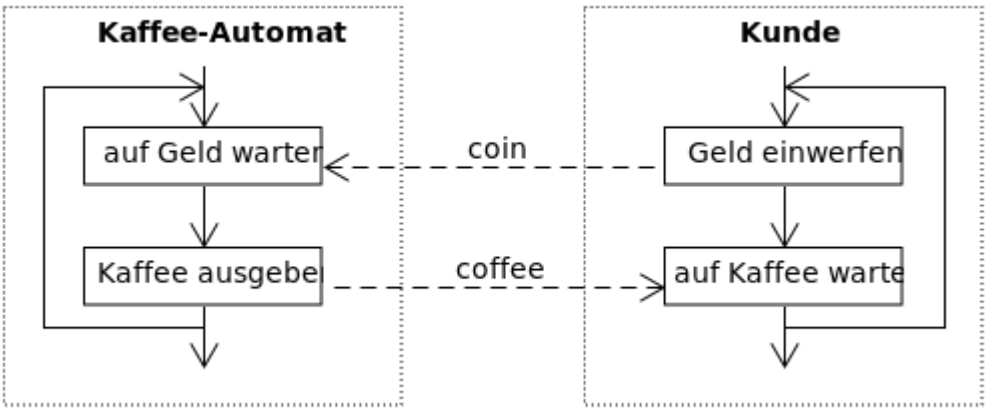
Das Lösen von Synchronisationsproblemen ist oft nicht einfach, weil Prozesse bzw. Threads gleichzeitig ablaufen, ihre Aktivitäten jedoch nach Vorgaben koordiniert werden müssen: man verliert schnell den Überblick. Systematisches Vorgehen mit Aufzeichnen der Abläufe und Synchronisationsbedingungen bewährt ich sich in diesem Fall.

8.3.1 3.1 Wie löst man Synchronisationsprobleme?

Gehen sie beim Lösen von Synchronisationsproblemen in folgenden Schritten vor:

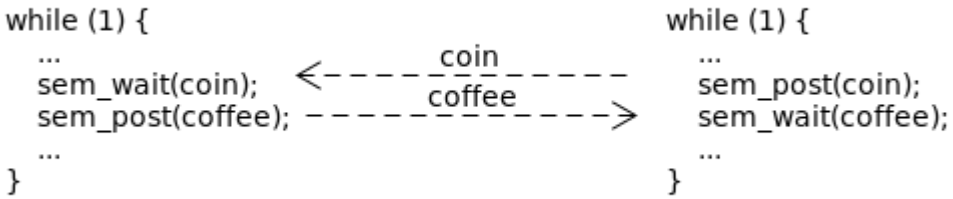
- **Schritt 1: Prozesse (Threads) der Problemstellung identifizieren.**
  Prozesse sind die Aktivitäten, die gleichzeitig ausgeführt werden. In diesem Sinne sind sie eigenständige Ausführungs-Einheiten, deren zeitliches Verhalten synchronisiert werden muss.
- **Schritt 2: Ausführungsschritte der einzelnen Prozesse (Threads) ermitteln.**
  Erstellen sie eine Liste mit einer Spalte für jeden Prozess. Notieren sie für jeden Prozess stichwortartig die wesentlichen Aktionen in der gewünschten zeitlichen Reihenfolge. Tragen sie noch keine Synchronisationsoperationen ein, sondern Texte wie warten auf Geld, etc. Übertragen sie anschliessend die Liste in einen Ablaufgraphen (Siehe Beispiel in Abbildung 1).
- **Schritt 3: Synchronisationsbedingungen ermitteln.**
  Eine Synchronisationsbedingung ist eine zeitliche Beziehung (Abhängigkeit) zwischen Aktionen verschiedener Prozesse, die für das korrekte Arbeiten erforderlich ist. Zeichnen sie diese Beziehungen mit Pfeilen in den Ablaufgraphen aus Schritt 2 ein (Siehe Abbildung 1).
- **Schritt 4: Benötigte Semaphore definieren.**
  Für jede Synchronisationsbedingung wird ein eigener Semaphor benötigt. Notieren sie für jeden Semaphor einen Namen und den Wert, mit dem er initialisiert werden muss.

- **Schritt 5: Prozesse mit Semaphore Operationen ergänzen.**
  Erweitern sie nun alle Prozesse aus Schritt 2 mit den notwendigen Semaphore Operati-onen (Siehe Pseudocode in Abbildung 1).
- **Schritt 6: Implementation.**
  Implementieren und testen sie das vollständige Programm.




```
coin   = sem_open(...,0);
coffee = sem_open(...,0);
```

Ablaufgraph und Pseudocode für 2 Prozesse und zwei Semaphore



8.4 4. Der Kaffee-Automat

Als Beispiel verwenden wir einen Automaten, der Kaffee verkauft. Der Kunde muss zum Kauf eines Kaffees zuerst eine bzw. mehrere Münzen einwerfen und anschliessend den gewünsch-ten Kaffee wählen. Der Automat gibt dann das entsprechende Getränk aus.

Im ersten Beispiel werden der Automat und die Kunden mit Threads modelliert und tauschen Daten über gemeinsame Speichervariablen aus. Im zweiten und dritten Beispiel werden der Automat und die Kunden mit Prozessen modelliert, dabei wird der Ablauf mit Hilfe von Sema-phoren gesteuert bzw. erzwungen.

Hinweis: die Programme zu den folgenden Aufgaben können alle mit **startApp.e** gestartet werden. Dieses Programm startet und stoppt Threads und Prozesse, alloziert und dealloziert die Ressourcen (Mutexes, Semaphore).

8.4.1 4.1 Aufgabe: Mutual Exclusion

Greifen mehrere Threads (oder Prozesse) auf gemeinsame Daten zu, können sogenannte Race Conditions entstehen. Das Resultat ist in diesem Fall abhängig von der Reihenfolge, in der die Threads (Prozesse) ausgeführt werden.

Im vorliegenden Beispiel wirft der Kunde eine 1 Euro Münze ein und drückt anschliessend auf eine von zwei Kaffeewahltasten. Dabei wird die Anzahl Münzen (*coinCount*) und die gewählte Kaffeesorte (*selCount1*, *selCount2*) inkrementiert. Diese Variablen sind in der Datenstruktur *cData* abgelegt, auf die gemeinsam Kaffee-Automat und Kunden zugreifen können. Der Auto-mat überprüft, ob die Anzahl Münzen und die Anzahl der Kaffeewahlen gleich gross sind, falls nicht, wird eine Fehlermeldung ausgegeben und alle Zähler auf *Null* gesetzt.

8.4.1.1 Aufgaben

1. Übersetzen sie die Programme im Verzeichnis *mutex* mit *make* und starten sie den Kaffee-Automaten mit **startApp.e** mehrmals hintereinander. Analysieren sie die Datenwerte in den Fehlermeldungen, beschreiben sie was die Gründe dafür sind bzw. sein können.
2. Schützen sie nun den Zugriff auf die gemeinsamen Daten mit einem Mutex so, dass alle Threads eine konsistente Sicht der Daten haben. Wir haben für sie einen Mutex vorbereitet: die Datenstruktur *cData* enthält die Mutex-Variable *mutex*, die in **startApp.c** initialisiert wird. Die Funktionen für das Schliessen und das Öffnen des Mutex (Locks) aus der *pthread* Bibliothek sind:

```
pthread mutex lock(&(cD->lock));
```

- und

```
pthread mutex unlock(&(cD->lock));
```

Überprüfen sie, ob der Kaffee-Automat nun keine Fehlermeldungen mehr ausgibt. Erhö-hen sie dazu auch die Anzahl Kunden *CUSTOMERS* in **commonDefs.h**, z.B. auf 10.

1. Im Thread des Kaffee-Automaten wird an verschiedenen Orten mehrmals auf die gemeinsamen Daten in *cD* zugegriffen. Wenn sie die gemeinsamen Daten in lokale Variablen kopieren und dann nur noch auf diese lokalen Variablen zugreifen würden, könn-ten sie dann auf die Synchronisation mit dem Mutex verzichten?
2. Wie oft kann ein einzelner Kunde einen Kaffee beziehen, bis der nächste Kunde an die Reihe kommt? Hier reicht eine qualitative Aussage.

8.4.2 4.2 Aufgabe: Einfache Reihenfolge

Wie sie im ersten Beispiel festgestellt haben, verhindert ein Mutex zwar, dass Race Conditions auftreten, die Verarbeitungsreihenfolge der Threads lässt sich jedoch nicht beeinflussen und ist zufällig. Im Folgenden soll eine erzwungene Verarbeitungsreihenfolge implementiert werden:

- Ein Kunde benutzt den Automat für einen Kaffeekauf exklusiv, d.h. alle Schritte des Kunden werden innerhalb eines Mutexes ausgeführt. Ist ein Kunde an der Reihe, wartet er bis der Automat bereit ist, wirft eine Münze ein, wartet auf den Kaffee und gibt anschlies-send den Automaten für den nächsten Kunden frei.
- Der Automat meldet zuerst in einer Endlos-Schleife, dass er für die Geld-Eingabe bereit ist, wartet dann auf die Eingabe einer Münze, gibt den Kaffee aus und meldet anschliessend wieder, wenn er bereit ist, etc.

Für die Lösung dieses Problems benötigen wir Semaphore, die, im Gegensatz zu Mutexes, auch in verschiedenen Prozessen gesetzt bzw. zurückgesetzt werden dürfen. Den Kaffee-Automat und die Kunden implementieren wir mit Prozessen. sie finden die entsprechenden Prozesse im Verzeichnis **basicSequence**.

8.4.2.1 Aufgaben

1. Beschreiben sie den Kaffee-Automaten mit Hilfe der 6 Schritte aus Abschnitt 3 auf Papier, dokumentieren sie dabei alle Schritte schriftlich.
2. Implementieren sie nun den Kaffee-Automaten. Ergänzen sie dazu den *coffeeTeller* und den *customer* Prozess so mit vier Semaphoren, dass die vorgegebenen Ablaufbedingungen eingehalten werden. Mit welchen Werten müssen die Semaphore initialisiert werden? Wir haben für sie vier Semaphore vorbereitet: Achtung, sie sind aber noch auskommentiert (siehe commonDefs.h und startApp.c. Die benötigten Semaphor-Funktionen aus der POSIX Bibliothek sind:

```
sem_wait(&semaphor);
```

und

```
sem_post(&semaphor);
```

Analysieren sie die Ausgabe der Prozesse (mehrmals starten). Was fällt auf?

1. Gibt Ihr Programm den Output in der korrekten Reihenfolge aus? Falls nicht, wie könnte das gelöst werden?

8.4.3 4.3 Aufgabe: Erweiterte Reihenfolge

Die Preise steigen dauernd ... auch der Kaffee wird immer teurer, er kostet nun 3 Euro. Da der Automat nur 1 Euro Stücke annehmen kann, muss der Kunde 3 Münzen einwerfen. Erweitern sie die Prozesse aus Aufgabe 4.2 so, dass eine vordefinierte Anzahl Münzen eingegeben werden muss (die Anzahl Münzen ist in *commonDefs.h* als *NUM_COINS* definiert). Verwenden sie keine zusätzlichen Semaphore, sondern nutzen sie, dass wir Counting Semaphore verwenden. Die vordefinierten Prozesse finden sie im Verzeichnis *advancedSequence*.

8.4.3.1 Aufgabe

- Passen sie den coffeeTeller und den customer Prozess so an, dass der Kunde mehrere Münzen einwerfen muss, bis der Automat einen Kaffee ausgeben kann.

Hinweis: POSIX Semaphore sind Counting Semaphore, können aber nicht auf vordefinierte Werte gesetzt werden (ausser bei der Initialisierung). Abhilfe schafft hier das mehrmalige Aufrufen von *sem_post()*, z.B. in einer for-Schleife.

8.4.4 4.4 Zusammenfassung

Wir haben drei grundlegenden Typen von Synchronisationsproblemen kennen gelernt:

- **Mutex** nur ein Prozess bzw. Thread kann gleichzeitig auf gemeinsame Daten zugreifen.
  - Beispiel: entweder liest der Kaffee-Automat die Daten oder ein Kunde verändert sie.
- **Einfache Reihenfolge** ein Prozess wartet auf die Freigabe durch einen anderen Prozess.
  - Beispiel: der Kaffee-Automat wartet auf die Eingabe einer Münze.
- **Erweiterte Reihenfolge** ein Prozess wartet auf mehrere Freigaben durch einen anderen Pro-zess.
  - Beispiel: der Kaffee-Automat wartet auf die Eingabe von drei Münzen.

8.5 5. International Banking

Die International Bank of Transfer (IBT) besitzt in 128 Ländern Filialen und stellt für 2048 spezielle Handels-Kunden in jeder Filiale ein Konto zur Verfügung. Gelder dieser Kunden werden dauernd zwischen den Filialen hin und her transferiert, dazu beschäftigt die Bank sogenannte Pusher. Pusher heben Geldbeträge von Konten in einer Filiale ab und buchen sie auf den entsprechenden Konten in irgendeiner (auch in der eigenen) Filiale wieder ein. Die Beträge liegen zwischen 1000 und 100'000 Dollar und werden zufällig ausgewählt, die Wahl der beiden Filialen ist ebenfalls zufällig.

8.5.1 5.1 Implementation

Im Folgenden arbeiten wir mit einer *pthread*-basierten Implementation der IBT, die Pusher werden dabei mit Threads implementiert. Die Filialen der Bank sind als Array von Strukturen implementiert, wobei pro Filiale ein Lock (*branchLock*) und ein Array von Konten (Accounts) definiert ist. Die Konten sind wiederum Strukturen mit dem Kontostand (*account*) und dem Lock (*acntLock*), siehe dazu auch den Source Code. Die Zugriffe auf die Gelder sind imple-mentiert (Funktionen *withdraw()*, *deposit()*, *transfer()*), aber nicht synchronisiert. **Hinweis:** es ist von Vorteil hier mit mehreren CPUs zu arbeiten. Falls sie eine VM verwenden, setzen sie die Anzahl CPUs auf das Maximum.

8.5.2 5.2 Aufgabe: Konto Synchronisation

1. Wechseln sie ins Verzeichnis **banking/a1**, übersetzen sie das Programm und starten sie es mit dem Skript `./startApp`. Analysieren und erklären sie die Resultate. Notie-ren sie sich zudem die Laufzeiten für 1, 2 und 4 Threads.
2. Synchronisieren sie die Kontenzugriffe so, dass möglichst viele Zugriffe gleichzeitig ausgeführt werden können und die Zugriffe atomar sind. Sie dürfen nur eines der beiden Locks *branchLock* bzw. *acntLock* verwenden: welches wählen sie und wieso? Be-gründen sie ihre Antwort und testen sie ihre Lösung.

8.5.3 5.3 Aufgabe: Filialen Zugriff in Critical Section

Ihr Chef meint, dass es wohl aus Sicherheitsgründen besser wäre, sowohl die Filialen und die jeweiligen Kontenzugriffen zu "locken".

1. Wechseln sie ins Verzeichnis banking/a2 und kopieren sie banking.c aus Aufgabe 5.2. Implementieren sie diese zusätzlichen Anforderungen. Analysieren sie die Resultate. Was stellen sie fest im Vergleich mit den Resultaten aus der Aufgabe 5.2? Was raten sie ihrem Chef?
2. Ein Kollege meint, es wäre effizienter beim Abheben des Betrags zuerst das Konto zu locken und dann die Filiale, hingegen beim Einbuchen zuerst die die Filiale und dann das Konto. Was für eine Antwort geben sie ihrem Kollegen?**Hinweis:** falls sie nicht sicher sind: probieren sie es aus.

8.5.4 5.4 Aufgabe: Refactoring der Synchronisation

Das International Banking Committe (IBC) erlässt neue Richtlinien, die unter anderem fordern, dass die Gesamtbilanz einer Bank über sämtliche Filialen zu jeder Zeit konsistent sein muss.

1. Erklären sie wieso die Implementationen aus Aufgabe 5.2 und 5.3 diese Anforderungen nicht erfüllen.
2. Ihr Entwicklungsteam kommt zum Schluss, dass den Pushern neu nur noch eine Funktion *transfer()* für die Überweisung von Beträgen zwischen den Filialen und Konten zur Verfügung gestellt werden darf. Welche Locks bzw. welches Lock muss verwendet werden, damit die Forderung des IBC erfüllt werden kann? Wechseln sie ins Verzeichnis *banking/a3* und ergänzen sie die Funktion *transfer()* in banking.c um die entsprechenden Lock-Funktionen. Wichtiger **Hinweis:** es darf kein neues Lock eingeführt werden und die Gesamtbilanz über sämtliche Filialen muss jederzeit konsistent sein.
3. Testen und analysieren sie das Programm und vergleichen sie die Resultate (Funktionalität, Laufzeit) mit den Lösungen aus Aufgabe 5.2 und 5.3. Notieren sie sich, was ihnen bei dieser Aufgabe wichtig erscheint.
4.

8.6 6. Bewertung

Die gegebenenfalls gestellten Theorieaufgaben und der funktionierende Programmcode müssen der Praktikumsbetreuung gezeigt werden. Die Lösungen müssen mündlich erklärt werden.

Aufgabe	Kriterium	Gewicht
	Sie können die gestellten Fragen erklären.	
4	4.1 Aufgabe: Mutual Exclusion4.2 Aufgabe: Einfache Reihenfolge4.3 Aufgabe: Erweiterte Reihenfolge	4
5	5.2 Aufgabe: Konto Synchronisation5.3 Aufgabe: Filialen Zugriff in Critical Section5.4 Aufgabe: Refactoring der Synchronisation	4

Version: 18.08.2021

Chapter 9

09 - File Operations

9.1 1. Übersicht

9.2 2. Lernziele

9.3 3. Aufgabe 1:

9.4 4. Bewertung

Die gegebenenfalls gestellten Theorieaufgaben und der funktionierende Programmcode müssen der Praktikumsbetreuung gezeigt werden. Die Lösungen müssen mündlich erklärt werden.

Aufgabe	Kriterium	Punkte
	Sie können das funktionierende Programm inklusive funktionierende Tests demonstrieren und erklären.	
1	-	-

Version: 16.02.2022

Chapter 10

10 - IPC

10.1 1. Übersicht

10.2 2. Lernziele

10.3 3. Aufgabe 1:

10.4 4. Bewertung

Die gegebenenfalls gestellten Theorieaufgaben und der funktionierende Programmcode müssen der Praktikumsbetreuung gezeigt werden. Die Lösungen müssen mündlich erklärt werden.

Aufgabe	Kriterium	Punkte
	Sie können das funktionierende Programm inklusive funktionierende Tests demonstrieren und erklären.	
1	-	-

Version: 16.02.2022