**BS Praktikumsaufgabe 02**

# Thread Koordination: Die trainierenden PhilosophInnen

**Version 0.2 - Abgabe am 7. November 2016 16:00**

Alexander Mendel
Karl-Fabian Witte

erstellt am 15. November 2016

Zu Anfang wird ein Makefile zu einer gegebenen C-Datei erstellt und somit das Grundprinzip von (GNU)Make erlernt. Am Ende soll die automatische Generierung der Abhängigkeiten der C-Datein von den Headerdatein für das Makefile erklärt werden. In der Hauptaufgabe wird die Synchronisation zwischen Threads geübt. Dabei wird unter anderem das Prinzip eines Monitors verwendet.

## Inhaltsverzeichnis

**1 Vorübung: Makefile** **2**
1.1 Automatische Generierung der Headerabhängigkeiten . . . . . 2

**2 Thread-Koordination in der Muckibude** **4**
2.1 Überblick . . . . . 4
2.1.1 Programmablauf . . . . . 6
2.2 Sichtbarkeit der Objekte . . . . . 6
2.2.1 Strukturen der Objekte . . . . . 7
2.3 Die Muckibude als Monitor . . . . . 8
2.3.1 Befehl wird zur höflichen Bitte . . . . . 10
2.4 Fehlerbehandlung . . . . . 10
2.5 Sonstiges . . . . . 10

2.6 Kleine Anmerkungen aus dem Labortermin . . . . . 11
2.6.1 Funktion quid_philo in philo.c . . . . . 13

# 1 Vorübung: Makefile

Um später einmal alle Quelldateien für die Kompilierung und Verlinkung zu organisieren, wird hier (GNU)Make geübt. Im Prinzip ist es eine kleine Skriptsprache, welche die Kompilierung koordiniert und somit mit einfachen Befehlen alle abhängigen Quelldateien überprüft und gegebenenfalls diesen Zweig wieder übersetzt. Zum üben wurde uns nur eine Quelldatei gegeben und wir haben das Makefile, wie hier dargestellt, realisiert:

```
1  EXE = critsect02
2
3  CC = /usr/bin/gcc
4
5  CFLAG  = -pthread
6  CFLAG += -g
7  CFLAG += -Wall
8
9  LDFLAG = -lpthread
10
11 SRC = critsect02.c
12
13 # targets (all is default, because it is on top.)
14 all: $(EXE)
15
16 run: $(EXE)
17         ./$(EXE)
18
19 $(EXE): $(SRC)
20         $(CC) $(CFLAG) $(LDFLAG) -o $@ $<
21
22 clean:
23         rm -f $(EXE)
```

## 1.1 Automatische Generierung der Headerabhängigkeiten

Der C-Kompilierer von GNU kann uns sehr viel Arbeit abnehmen: Er erspart uns mit einem kleinen Trick, dass wir jeden Header einzeln zu der jeweiligen Objektdatei

zuordnen müssen. So können wir nämlich auch die Wildcardmethode `%.o:%.c` weiter verwenden und trotzdem die Headerabhängigkeiten berücksichtigen.

Mit dem Flag `-M` werden alle Headerdatein der jeweiligen Objektdatei zugeordnet und diese im Makefileformat (`traget.o: header1.h header2.h`) auf die Ausgabe geleitet. Diese Ausgabe leitet man in eine Datei, die in das Makefile eingefügt wird. Die beschrieben Stellen unseres Makefiles sind hier einmal dargestellt:

```
1  DEPENDFILE = .depend
2
3  ifeq ($(DEPENDFILE),$(wildcard $(DEPENDFILE)))
4  -include $(DEPENDFILE)
5  endif
6
7  dep depend $(DEPENDFILE):
8          $(CC) $(CFLAGS) -MM *.c > $(DEPENDFILE)
```

### -MM

Dieses Flag erstellt nur die Abhängigkeiten der privaten Header. Die Bibliotheksheader werden nicht mit aufgeführt.

### $(wildcard)

Diese Operation erstellt eine Liste der Dateien im selben Ordner wie das Makefile, wenn als Argument ein String mit Wildcardzeichen übergeben wird. Wenn man genau hin guckt, überprüft diese Bedingung in unserem Quellcode nur, ob sich eine Abhängigkeitsdatei im Ordner befindet und bindet diese ggf. ein.

### Fehler bei der erneuten Berechnung der Abhängigkeiten

Dieser Fehler tritt auf, wenn man keine Rechte für die neu erstellte Abhängigkeitsdatei hat, was auf der virtuellen Maschine im eingehängten Teilordner der Fall zu sein scheint.

#### Weitere Probleme

Es scheint einen Haufen von Problemen zu geben, wenn es um diese `Dependance` geht. Deswegen wird das Kapitel nur vorläufig geschlossen und weiter verfolgt.

# 2 Thread-Koordination in der Muckibude

Der folgende Abschnitt wurde kaum verändert aus unserem Entwurf entnommen.

In dieser Aufgabe soll die Thread-Koordination mit den Bibliotheken der POSIX Threads und Semaphoren unter Linux geübt werden. Dafür schicken wir 5 Philosophen (Threads) in die Muckibude. Wie wir die Koordination der Gewichte (Daten) zwischen den Philosophen realisieren, soll in den folgenden Abschnitten erläutert werden.

## 2.1 Überblick

Dieser Abschnitt dient dazu, sich schnell ein Bild von der Implementationsumsetzung machen zu können.

Das Programm wird aus folgenden Quellcodedatein zusammengestellt:

parameter.h Hier sind alle Konstanten für die Implementierung definiert.

gym.c Realisierung eines Monitors mit zwei Monitorfunktionen und den (empfindlichen) Daten: den Gewichten. Zudem weitere Hilfsfunktionen.

gym.h Das Interface für Funktionen und Typedefinitionen.

philo.c Die Threadfunktion, welche die Philosophen zu einem Zyklus des Gewichte Holen, Heben, Weglegen und Ausruhen treibt. Zudem Deklaration des globalen Befehlfeldes.

philo.h Das Interface und die externe Bekanntmachung des globalen Befehlfeldes.

global.c Die Funktion `handle_error` wird hier implementiert. Damit wird der Rückgabewert von einer Bibliotheksfunktion ausgewertet, bei einem Fehler die entsprechende Meldung ausgegeben und das Programm beendet.

global.h Das Interface für `handle_error`. Zudem werden hier `TRUE` und `FALSE` definiert.

main.c Die Mainfunktion mit weiteren Hilfsfunktionen. Der Mainthread koordiniert die Initialisierung und Zerstörung der anderen Threads und deren Objekten. Zudem steuert der Mainthread die interaktive Annahme der Befehle über die Konsole.

Es sollen kurz die Prototypen der wichtigsten Funktionen aufgelistet werden:

```
1  //     GYM
2  void init_gym_obj();  // init the stock, bags, pthread_mutex and pthread_cond
3
4  void free_gym_obj();  // destroys the pthread_mutex and pthread_cond
5  void free_gym_threads(); //  set is_quit = TRUE and pthread_cond_broadcast
6
7  int get_weights( int tid );  // monitor func
8  int put_weights( int tid );  // monitor func
9  int display_status( void );  // is called in monitor functions
10
11 int fill_bag( weight_bag_t * bag);  // get the weights form stock into bag
12 void flush_bag(weight_bag_t * bag); // put the weights back in stock
13
14 //     PHILO
15 void init_philo_obj(void);  // init philostructur, semaphore
16 void free_philo_obj(void);  // destroy the semaphore
17
18 void * philothread (void * ptr_tid); // thread function, no return, param tid
19
20 void set_train_status(train_state_t state, int tid); // used by monitor func
21
22 void unblock_philo(int tid);  // used to unblock the thread
23
24 //     GLOBAL
25 void handle_error(  int result,         // an universal tool
26                     int errnum,
27                     char const *mssge);
```

Und hier der Parameterheader:

```
1   #define NPHILO      5
2   //#define PHILO_NAMES { "Anne", "Bernd", "Clara", "Dirk", "Emma" }
3   #define PHILO_KG    {      6,       8,      12,     12,     14 }
4
5   #define NWEIGHTS_T 3
6   #define WEIGHT_KG  { 2, 3, 5 }
7   #define WEIGHT_QTY { 4, 4, 5 }
8
```

```
9   #define REST_LOOP   1000000000
10  #define WORKOUT_LOOP 500000000
```

### 2.1.1 Programmablauf

**Der Mainthread** erzeugt bzw. initialisiert alle Objekte. Dann werden die Pthreads (Philosophen) gestartet. Danach ist der Mainthread in der Controllerschleife. In dieser Endlosschleife wartet der Mainthread auf eine Eingabe aus der `stdin` und überprüft diesen auf die Befehle:

q oder Q Beendet das Programm sauber.

`tid`b Blockiert Thread Nummer tid.

`tid`u Befreit Thread Nummer tid.

`tid`p Thread Nummer tid überspringt die REST und WORK_OUT Schleifen.

Ein akzeptierter Befehl wird entschlüsselt und bearbeitet. Wenn der Befehl zum Schließen des Programms erkannt wurde, werden alle Philosophenthreads gebeten, sich zu beenden (über das globale Befehlsfeld). Alle blockierten Threads werden befreit, indem man die Blockierungsobjekte freigibt. Der Mainthread wartet nun darauf, dass alle Threads terminieren und zerstört alle erstellten Objekte. Erst dann beendet sich der Mainthread und terminiert somit das Programm.

**Einem Philothread** wird die Nummer (`tid`) mitgegeben, mit welcher er auf die für ihn wichtigen Felder zugreifen und diese manipulieren kann. Er läuft in einer Endlosschleife, in der er vier Funktionen pro Schleife ausführt. Zwei davon sind fast leere Zählerschleifen, in denen er auf die Befehle in dem globalen Array reagiert. Die anderen beiden Funktionen werden durch das Monitormodell realisiert, um die Konsistenz der Daten (Gewichte) wahren zu können. In diesen beiden Funktionen ignoriert der Philothread zunächst die Befehle des Mainthreads. In den Funktionen des Monitors wird beim Eintreten in und beim Austreten aus dem geschützten Bereich innerhalb diesem der Status auf der Konsole ausgegeben.

## 2.2 Sichtbarkeit der Objekte

Da es sich um eine statische Anzahl an Threads und Objekten handelt, können die Objekte ebenfalls statisch sein. Unsere Synchronisationsobjekte haben zudem

Dateisichtbarkeit. Somit wird für jede C-Datei eine Initialisierungs- und eine Zerstörungsfunktion implementiert, welche dann von dem Mainthread aufgerufen wird. Das globale (zwischen mehreren C-Datein sichtbar) Befehlsfeld ist von main.c und von philo.c aus sichtbar.

Wir haben uns gegen dynamischen HEAP-Speicher entschieden, da dies keinen Sinn ergebe, da kein Mehrwert für uns zu erkennen war.

### 2.2.1 Strukturen der Objekte

Gerade die zu schützenden Objekt sind hier von großer Bedeutung: die Gewichte. Dabei verwaltet gym.c alle Gewichtobjekte (weight), auch die der Philosophen. Beim ersten Blick ist es zunächst komisch, dass der Philosoph selber nicht weiß, wie viel Kilogramm Gewichte er für sein Training bekommt. Der Programmteil der Muckibude benötigt als einziger diesen Wert, um die Gewichte aus dem Lager (`stock`) in die für die Philosophen bereitgestellten Gewichtetaschen (`bag`) zu füllen:

Deswegen hat die Muckibude folgende Strukturen im statischen privaten Sichtbarkeitsfeld:

```
1  typedef struct {
2      int kg;         // type
3      int qty;        // quantity
4  } weight_t;
5
6  struct{
7      weight_t weights[ NWEIGHTS_T ]; // slots
8      int total_kg;                   // total amount of all weights in the gym
9      int available_kg;               // amount what is left
10 } stock;
11
12 typedef struct {
13     weight_t weights [ NWEIGHTS_T ];   // slots
14     int required_kg;                   // required for a effective trainigs
15 } weight_bag_t;
16
17 weight_bag_t bags[NPHILO];            // for every philo one bag
18
19 pthread_mutex_t gym_mtx;
20 pthread_cond_t gym_cv;
```

Die Philosphen haben primär die Information, in welchem Status sie sich befinden:

```
typedef enum {
    QUIT=20,
    BLOCKED,
    NORMAL
} cmd_state_t;

typedef enum {
    GET_WEIGHTS=10,
    WORKOUT,
    PUT_WEIGHTS,
    REST
} train_state_t;

typedef struct {
    sem_t block_sem;
    cmd_state_t cmd_state;
    train_state_t train_state;
    int tid;
} philo_t;

extend char cmd_arr[];
```

## 2.3 Die Muckibude als Monitor

Die Muckibude `gym.c` hat als Synchronisationsobjekte einen POSIX Mutex `gym_mtx` und eine POSIX Bedingungsvariable `gym_cv`. Beim Eintreten und Verlassen des geschützten Bereichs, wird jeweils der Trainierstatus des Philothreads entsprechend geändert und `display_status` aufgerufen. `display_status` überprüft, ob auch kein Gewicht verloren gegangen ist und gibt ggf. eine Fehlermeldung aus.

Nur die Threads, die sich Gewichte holen wollen (`GET_WEIGHTS`), können auf Grund der beschränkten verfügbaren Gewichte die erforderlichen Gewichte nicht holen. Somit warten sie auf der POSIX Bedingungsvariable in einer Whileschleife und geben den geschützen Bereich wieder frei. Wenn aber einige Gewicht von einem anderen Thread wieder zurückgelegt wurden (`PUT_WEIGHTS`), werden vor dem Austreten aus dem sicheren Bereich alle auf Bedingungsvariablen sitzenden Threads geweckt, welche dann wiederum prüfen können, ob sie diesmal ihre Gewichte erhalten können.

Das folgende Listing zeigt die beiden Monitorfunktionen. Die zweite Bedingung `is_quit` ist für die Freisetzung der blockierenden Threads, nachdem der Beendenbefehl erteilt wurde.

```c
/* ------------------------------------------------------------GET_WEIGHTS */
int get_weights(int tid){

    int res = EXIT_FAILURE;
    int no_weights_fit_cond; // condition for the failure of fill_bag

    res = pthread_mutex_lock( &gym_mtx );
    handle_error(res, errno, "phtread_mutex_lock ");

    set_train_status(GET_WEIGHTS, tid);
    printf("Tid %d GET_WEIGHTS --- ", tid  );
    display_status();

    no_weights_fit_cond = fill_bag( &bags[tid] );
    while( no_weights_fit_cond && !is_quit ){
        res = pthread_cond_wait( &gym_cv, &gym_mtx);
        handle_error(res, errno, "phtread_cond_wait ");
        no_weights_fit_cond = fill_bag( &bags[tid] );
    }

    set_train_status(WORKOUT, tid);
    printf("Tid %d WORKOUT     --- ", tid  );
    display_status();

    res = pthread_mutex_unlock( &gym_mtx );
    handle_error(res, errno, "phtread_mutex_unlock ");
    return EXIT_SUCCESS;
}

/* ------------------------------------------------------------PUT_WEIGHTS */
int put_weights(int tid){
    int res= EXIT_FAILURE;
    res = pthread_mutex_lock( &gym_mtx );
    handle_error( res, errno,"pthread_mutex_lock ");

    set_train_status(PUT_WEIGHTS, tid);
    printf("Tid %d PUT_WEIGHTS --- ", tid  );
    display_status();

    flush_bag(&bags[tid]);

    res = pthread_cond_broadcast( &gym_cv );
    handle_error( res, errno,"pthread_cond_broadcast ");

    set_train_status(REST, tid);
    printf("Tid %d REST        --- ", tid  );
    display_status();
```

```
48
49      res = pthread_mutex_unlock( &gym_mtx );
50      handle_error( res, errno,"pthread_mutex_unlock ");
51      return EXIT_SUCCESS;
52  }
```

### 2.3.1 Befehl wird zur höflichen Bitte

Die Befehle werden nach dem Entschlüssen (siehe bei Sonstiges), in ein globales Feld gelegt. Der Thread prüft in seinem `WORK_OUT` oder `REST` Zustand (in den Zählerschleifen), ob es einen Befehl gibt und blockiert sich ggf. selber mit seinem Semaphor oder überspringt die Zählerschleife oder setzt zudem seinen Befehlszustand auf `QUIT` und terminiert. Einzige Ausnahme von dieser höflichen Form ist der Befehl zum Entblocken. Dort wird direkt der Semaphor um einen erhöht und somit der Thread wieder geweckt.

## 2.4 Fehlerbehandlung

Grundsätzlich terminiert die Funktion `handle_error` das Programm. Also beendet jeder fehlgeschlagene Bibliotheksfunktionsaufruf (sofern alle abgegriffen werden) das Programm. Fehler, die wir selber definieren, wie das Eingeben eines nicht akzeptierten Befehls, werden zu Kenntnis genommen aber weitestgehend toleriert.

## 2.5 Sonstiges

Wie werden die Befehle ausgelesen: `fgets` liefert einen String zurück, der leider auch das Zeilenumbruchzeichen von dem `NULL` einfügt. Bevor man das Befehlsmuster mit dem erhaltenen Befehlsstring vergleicht, wird der Zeilenumbruch mit `NULL` ersetzt. In kurzform so:

```
1       fgets(cmd_s, NBUFF,stdin)
2       cmd_s[strcspn(cmd_s, "\n")]= '\0';
3       if (strcmp("q", cmd_s) ) ....
```

Die Threadnummer erhalten wir, indem man mit einer Zählerschleife alle Threadnummermöglichkeiten mit dem Characterwert der Null summiert und das Ergebnis mit dem ersten Character im Befehlsstring vergleicht. Nach erfolgreicher

Threadnummererkennung wird der zweite Character mit Befehlsmustern verglichen und zur richtigen Ausführung verzweigt.

```
    for(int i =0; NPHILO > i ; i++){
        int target = -1;
        if (cmd_s[0]== i + '0' ){
            target = i;
        }
    }
    if (0 <= target ){
        switch (cmd_s[1]){
            case b:
            ...
```

## 2.6 Kleine Anmerkungen aus dem Labortermin

Die Berechnung der Gewichte für die jeweilige Trainigseinheit erfolgt rekursiv mit folgenden Funktionen:

```
/*------------------------------------------------------- FILL BAG ------*/
int fill_bag_recu( const int required_kg,
                   const int index,
                   Weight_bag_t * const bag){
    int remain_kg = required_kg;
    int success = EXIT_FAILURE;

    int qty = required_kg/ stock.weights[index].kg;
    if ( stock.weights[index].qty < qty ){
        qty = stock.weights[index].qty;
    }

    while ( 0 <= qty){

        remain_kg = required_kg - stock.weights[index].kg * qty;
        if ( 0 == remain_kg ){
            success = EXIT_SUCCESS;
        } else {
            if ( 0 < index ){
                success = fill_bag_recu(remain_kg, index - 1, bag);
            } else {
                success = EXIT_FAILURE;
            }
        }

```

```
26          if ( EXIT_SUCCESS == success ){
27              bag->weights[index].qty = qty;
28              stock.weights[index].qty -= qty;
29              stock.available_kg -= qty * stock.weights[index].kg;
30              break;
31          } else if ( EXIT_FAILURE == success ){
32              qty--;
33          }
34      }
35      return success;
36  }
37  /* ------------------------------------------------ fill_bag wrapper ----- */
38  int fill_bag(Weight_bag_t * bag){
39      int success = EXIT_FAILURE;
40      int required_kg= bag->required_kg;
41      if (stock.available_kg >= required_kg && 0 < required_kg){
42      int weights_len = NWEIGHTS_T;
43          success = fill_bag_recu(required_kg, weights_len--, bag);
44      }
45      return success;
46  }
```

Es wird dabei wie folgt vorgegangen:

- Es werden von den schwersten Gewichten, so viele vorgemerkt, sodass das benötigte Gewicht sich an die Null approximiert.
- In einer Whileschleife (solange die Anzahl der vorgemerkten Gewichte größer Null ist) wird getestet, ob diese Gewichte schon ausreichen.
- Gewichte reichen aus, Rückgabevariable wird auf EXIT_SUCCESS gesetzt;
- Gewichte reichen nicht aus, es wird, falls vorhanden, auf das nächst leichtere Gewicht mit einem rekursiven Aufruf zugegriffen. Dabei wird das benötigte Restgewicht mit übergeben. Der Rückgabewert von diesem Aufruf wird auf die Rückgabevariable gelegt:
- Falls kein leichteres Gewicht verfügbar ist, wird die Rückgabevariable auf EXIT_FAILURE gesetzt.
- Die Rückgabevariable wird getestet:

- falls EXIT_SUCCESS: Gewichte werden in den Beutel gepackt und aus dem Lager entfernt und es wird die Schleife verlassen, was zu einer Rückgabe des EXIT_SUCCESS führt.

- falls EXIT_FAILURE: Die Anzahl der Gewichte wird um einen reduziert.

- whileschleife fängt von vorne an, oder falls die Anzahl null ist, endet die Funktion und ein EXIT_FAILURE wird zurückgegeben.

#### Stringmodifikation in gym.c

Es wurde zudem diskutiert, dass Anstelle von der Bibliotheksfunktion `strcat` die etwas kontrolliertere `strncat` verwedent werden sollte. Im Prinzip ist die `strncat` vorzuziehen da man hier die Anzahl der zu konkatenierenden Zeichen festlegen kann, was unerwünschtes überschreiben von Speicherbereichen verhindern kann. Der Quellcode beinhaltet jedeoch noch die `strcat` Funktion, ohne Buffergrenze, da wir sichgestellt haben, dass es nicht zu einem überlauf kommen kann. (Siehe die sehr komplizierten Defines der Stringformate und Buffergrößen in `gym.h`.)

### 2.6.1 Funktion quid_philo in philo.c

Wir sollten diese Funktion wegen diesem Schreibfehlers mittels Refactoring zu `quit_philo` ändern. Die Funktion wurde nie aufgerufen, sodass sie auskommentiert ( wegen nachweislichen Zwecken ) wurde.