Übungen Betriebssysteme (BS)

U3 – Deadlocks

CORONA-EDITION

https://sys.cs.tu-dortmund.de/DE/Teaching/SS2021/BS/

Peter Ulbrich

peter.ulbrich@tu-dortmund.de

https://sys.cs.tu-dortmund.de/EN/People/ulbrich/

technische universität
dortmund

arbeitsgruppe
systemsoftware

Agenda

- Besprechung Aufgabe 2: Threadsynchronisation
- Fortsetzung Grundlagen C-Programmierung
- Aufgabe 3: Deadlock
  - Semaphore
  - Wiederverwendbare/Konsumierbare Betriebsmittel
  - Problemvorstellung
  - Voraussetzungen für Verklemmungen
  - Verklemmungsauflösung
  - Makefiles
- Alte Klausuraufgabe zum Thema Semaphore

Besprechung Aufgabe 2

- → Foliensatz Besprechung A2

Grundlagen C-Programmierung

- → Foliensatz C-Einführung (Folie 43-46)

Semaphore

- Ein Sempahor ist eine **Betriebssystemabstraktion** zum Austausch von Synchronisierungssignalen zwischen nebenläufig arbeitenden Prozessen.
- Steht für „Signalgeber“
- E. Dijkstra: Eine „nicht-negative ganze Zahl“, für die folgenden zwei **unteilbaren Operationen** definiert sind.

Semaphor-Operationen

- **P** (holländisch prolaag, „erniedrige“; auch *down* oder ***wait***)
  - hat der Semaphor den Wert 0, wird der laufende Prozess blockiert
  - ansonsten wird der Semaphor um 1 dekrementiert
- **V** (holländisch verhoog, „erhöhe“; auch *up* oder ***signal***)
  - auf den Semaphor ggf. blockierter Prozess wird deblockiert
  - ansonsten wird der Semaphor um 1 inkrementiert

Eselsbrücken für Semaphore

- Mit **P** wartet man auf eine Ressource und belegt diese.

  „p(b)elegen, ggfs. vorher warten"

  Es sind danach weniger Ressourcen verfügbar, also wird *runter*gezählt.

- Mit **V** gibt man eine Ressource wieder frei, ggfs. wird der nächste wartende Thread benachrichtigt.

  „v(f)reigeben, ggfs. Benachrichtigen"

  Es sind danach wieder mehr Ressourcen verfügbar, also wird *hoch*gezählt.

Mutexe vs. Semaphore

- Mit beiden können kritische Abschnitte geschützt werden.
- Beim **Mutex** kann immer **nur ein Thread** den kritischen Abschnitt betreten.
- Mit einem **Semaphor** können **n Threads** den kritischen Abschnitt betreten.
  - Nützlich für Betriebssystemressourcen, bei denen eine bestimmte Anzahl zur Verfügung steht.
- Für **n=1** verhält sich ein Semaphor ähnlich wie ein Mutex. Semaphore können aber auch von einem anderen Prozess freigegeben werden als dem, der sie belegt hat.

Semaphor – Beispiel

- Gemeinsam genutzte FIFO-Queue mit maximal 100 Elementen

```
/* gem. Speicher */
Semaphore lock;
Semaphore freiePlaetze;
struct List * queue;
```

```
/* Initialisierung */
lock = 1;
freiePlaetze = 100;
queue->head = NULL;
queue->tail = NULL;
```

Mehrere Threads können Elemente in die Queue schreiben, andere Threads können dann Elemente wieder herausnehmen.

```
void enqueue(element *item){
  if (item != NULL){
    p(&freiePlaetze);
    p(&lock);
    queue->tail = item;
    ...
    v(&lock);
  }
}
element * dequeue(){
  p(&lock);
  element *item = queue->head;
  ...
  if (item != NULL){
    v(&freiePlaetze);
  }
  v(&lock);
  return item;
}
```

Semaphor – Beispiel

- Zusätzlich noch einseitige Synchronisation

```
/* gem. Speicher */
Semaphore verfuegbar;
```

```
/* Initialisierung */
verfuegbar = 0;
```

Die Consumer lesen nur, wenn etwas in der Queue steht.

```
void producer(){
  while(1){
    element *e = produce();
    enqueue(e);
    v(verfuegbar);
  }
}
void consumer(){
  while(1){
    p(verfuegbar);
    element *e = dequeue();
    consume(e);
  }
}
```

POSIX Semaphore

```
int sem_init(sem_t *sem, int pshared, unsigned int value);
```

```
#include <semaphore.h>
```

- Anlegen einer Semaphor
  - Parameter
    - sem: Adresse des Semaphor-Objekts
    - pshared: 0, falls nur zwischen Threads eines Prozesses verwendet
    - value: Initalwert des Semaphors, entspricht dem **n**
  - Rückgabewert:
    - 0, wenn erfolgreich
    - -1 im Fehlerfall
  - ***Bsp***.:

```
sem_t semaphor;
if (sem_init(&semaphor, 0, 1) == -1) {
   /* Fehlerbehandlung */
}
```


POSIX Semaphore

- Belegen eines Semaphors (**P**), ggf. müssen wir vorher warten:

```
int sem_wait(sem_t *sem);
```

- Freigeben eines Semaphors (**V**), ggf. wird der nächste Thread benachrichtigt:

```
int sem_post(sem_t *sem);
```

- Entfernen eines Semaphors:

```
int sem_destroy(sem_t *sem);
```

  - Parameter
    - sem: Adresse des Semaphor-Objekts
  - Rückgabewert:
    - 0, wenn erfolgreich
    - -1 im Fehlerfall

POSIX Semaphore

```
int sem_timedwait(sem_t *sem, struct timespec *abs_timeout);
```

- Belegen eines Semaphors (**P**), mit Angabe der maximalen Wartezeit.
  - Parameter
    - sem: Adresse des Semaphor-Objekts
    - abs_timeout: Adesse eines timespec sturcts mit dem Zeitpunkt bis zu dem gewartet werden soll.
  - Rückgabewert:
    - 0, wenn erfolgreich
    - -1 im Fehlerfall, **errno** beachten

Time

```
int clock_gettime(clockid_t clockid, struct timespec *tp);
```

- Gibt die aktuelle Zeit zurück
  - Parameter
    - clockid: Id der Uhr die verwendet werden soll (`Bei uns CLOCK_REALTIME`)
    - tp: timespec struct, in dem die aktuelle Zeit gespeichert werden soll.
  - Rückgabewert:
    - 0, wenn erfolgreich
    - -1 im Fehlerfall

```
#include <time.h>
```

Timespec - Beispiel

```
#include <time.h>
sem_t sem; /*Semaphor*/

 …

 struct timespec waittime;

 if(clock_gettime(CLOCK_REALTIME, &waittime)) {
    perror("clock_gettime");
    exit(1);
 }
 waittime.tv_sec += 10;
 int status = sem_timedwait(&sem, &waittime);
 /* Fehlerbehandlung hier!*/
```

Betriebsmittel ...

werden vom Betriebssystem verwaltet und den Prozessen zugänglich gemacht. Man unterscheidet zwei Arten:

- **Wiederverwendbare Betriebsmittel**
  - Werden von Prozessen für eine bestimmte Zeit belegt und anschließend wieder freigegeben.
  - **Beispiele:** CPU, Haupt- und Hintergrundspeicher, E/A-Geräte, Systemdatenstrukturen wie Dateien, Prozesstabelleneinträge, …
  - Typische Zugriffssynchronisation: **Gegenseitiger Ausschluss**
- **Konsumierbare Betriebsmittel**
  - Werden im laufenden System erzeugt (produziert) und zerstört (konsumiert)
  - **Beispiele:** Unterbrechungsanforderungen, Signale, Nachrichten, Daten von Eingabegeräten
  - Typische Zugriffssynchronisation: **Einseitige Synchronisation**

A3: Problemvorstellung

Das Szenario:

Die Archäologen des Kronos Instituts führen Ausgrabungen an verschiedenen Ausgrabungsstätten durch. Jedem Archäologen wird eine Ausgrabungsstätte zugeteilt. Leider sind die Fördermittel für die Ausgrabung begrenzt, daher sind manche Messgeräte und Werkzeuge nur in geringen Stückzahlen verfügbar und lagern zentral im Institut.

Welche Geräte für die jeweilige Ausgrabungsstätte benötigt werden, ergibt sich erst nach und nach im Laufe der Ausgrabung. Wird ein neues Gerät benötigt, so holt der Archäologe dieses vom Institut und fährt mit der Ausgrabung fort. Ist das Gerät nicht im Institut, dann wartet er darauf, dass es zurückgebracht wird. Solange die Ausgrabung nicht abgeschlossen ist, befinden sich noch alle Geräte in Benutzung und können daher nicht zurück zum Institut gebracht werden. Ist eine Ausgrabung beendet, dann bringt der Archäologe alle Geräte zum Institut zurück, und ihm wird eine neue Ausgrabungsstätte zugeteilt.

Deadlock-Voraussetzungen

Die notwendigen Bedingungen für eine Verklemmung:

Deadlock-Voraussetzungen

Die notwendigen Bedingungen für eine Verklemmung:

1. ***„mutual exclusion“***
   - die umstrittenen Betriebsmittel sind nur unteilbar nutzbar

Deadlock-Voraussetzungen

Die notwendigen Bedingungen für eine Verklemmung:

1. ***„mutual exclusion“***
   - die umstrittenen Betriebsmittel sind nur unteilbar nutzbar
2. ***„hold and wait“***
   - die umstrittenen Betriebsmittel sind nur schrittweise belegbar

Deadlock-Voraussetzungen

Die notwendigen Bedingungen für eine Verklemmung:

1. ***„mutual exclusion“***
   - die umstrittenen Betriebsmittel sind nur unteilbar nutzbar
2. ***„hold and wait“***
   - die umstrittenen Betriebsmittel sind nur schrittweise belegbar
3. ***„no preemption“***
   - die umstrittenen Betriebsmittel sind nicht rückforderbar

Deadlock-Voraussetzungen

Die notwendigen Bedingungen für eine Verklemmung:

1. ***„mutual exclusion“***
   - die umstrittenen Betriebsmittel sind nur unteilbar nutzbar
2. ***„hold and wait“***
   - die umstrittenen Betriebsmittel sind nur schrittweise belegbar
3. ***„no preemption“***
   - die umstrittenen Betriebsmittel sind nicht rückforderbar

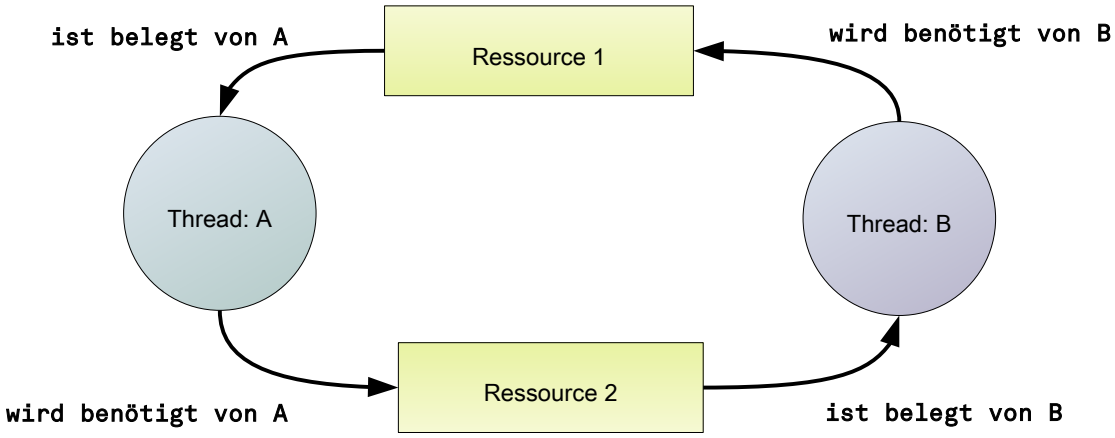
Weitere **Bedingung zur Laufzeit**:

4. ***“circular wait”***
   - eine geschlossene Kette wechselseitig wartender Prozesse

Deadlock-Voraussetzungen

4. *„circular wait“*

- eine geschlossene Kette wechselseitig wartender Prozesse



Verklemmungsauflösung

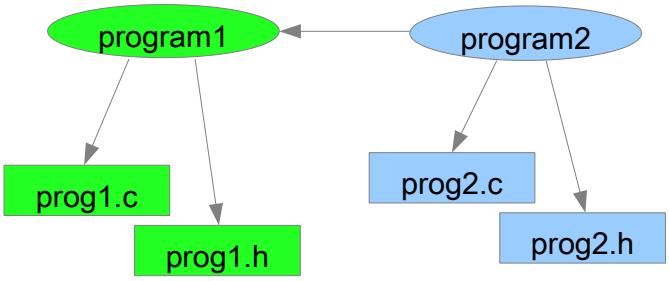
- Die „einfachste“ Variante: **Prozesse abbrechen** und so Betriebsmittel frei bekommen
  - Verklemmte Prozesse schrittweise abbrechen (großer Aufwand)
  - Mit dem „effektivsten Opfer“ (**?**) beginnen
  - Oder: alle verklemmten Prozesse terminieren (großer Schaden)

Makefiles

- Bauen von Projekten mit mehreren Dateien
- Makefile → Informationen wie eine Projektdatei beim Bauen des Projektes zu behandeln ist

```
# -= Variablen  =-
# Name=Wert  oder auch
# Name+=Wert  für Konkatenation

CC=gcc
CFLAGS=-Wall -ansi -pedantic -D_XOPEN_SOURCE -D_POSIX_SOURCE

# -= Targets =-
# Name: <benötigte "Dateien" und/oder andere Targets>
# <TAB> Kommando
# <TAB> Kommando ...  (ohne <TAB> beschwert sich make!)

all: program1 program2     # erstes Target = Default-Target

program1: prog1.c prog1.h
    $(CC) $(CFLAGS) -o program1 prog1.c
program2: prog2.c prog2.h program1 # Abhängigkeit: benötigt program1!
    $(CC) $(CFLAGS) -o program2 prog2.c
```


Targets & Abhängigkeiten




```
# target program1
program1: prog1.c prog1.h
...
# target program2
program2: prog2.c prog2.h program1
...
```

- Vergleich von Änderungsdatum der Quell- und Zieldateien
  - Quelle jünger? → Neu übersetzen!
- make durchläuft Abhängigkeitsgraph
- Java-Pendant: Apache **Ant**

Makefile

- Makefiles ausführen mit make <target>
  - bei fehlendem <target> wird das Default-Target (das 1.) ausgeführt
  - Optionen
    - -f: Makefile angeben; make -f <makefile>
    - -j: Anzahl der gleichzeitig gestarteten Jobs, z.B. make -j 3

Klausuraufgabe: Synchronisierung

Why did the multithreaded chicken cross the road? Die drei Funktionen des folgenden Programms werden in jeweils eigenen Prozessen ausgeführt, die alle zur selben Zeit laufbereit werden. Sorgen Sie durch geeignete Synchronisation der Prozesse dafür, dass das Programm

to get to the other side

ausgibt. Dafür stehen Ihnen drei Semaphore zur Verfügung, die Sie geeignet initialisieren müssen. Setzen Sie an den freien Stellen Semaphor-Operationen (P, V) auf die Semaphore (S1, S2, S3) ein (z.B. `P(S1)`).

Initialwerte der Semaphore: S1 = ☐ S2 = ☐ S3 = ☐

```
chicken1() {
   printf("to");

   printf("to");

   printf("other");
}
```

```
chicken2() {
   printf("get");
}
```

```
chicken3() {
   printf("the");

   printf("side");
}
```

Klausuraufgabe: Synchronisierung

Why did the multithreaded chicken cross the road? Die drei Funktionen des folgenden Programms werden in jeweils eigenen Prozessen ausgeführt, die alle zur selben Zeit laufbereit werden. Sorgen Sie durch geeignete Synchronisation der Prozesse dafür, dass das Programm

to get to the other side

ausgibt. Dafür stehen Ihnen drei Semaphore zur Verfügung, die Sie geeignet initialisieren müssen. Setzen Sie an den freien Stellen Semaphor-Operationen (P, V) auf die Semaphore (S1, S2, S3) ein (z.B. `P(S1)`).

Initialwerte der Semaphore: S1 = 0 S2 = 0 S3 = 0

```
chicken1() {

   printf("to");
   V(S2);
   P(S1);
   printf("to");
   V(S3);
   P(S1);
   printf("other");
   V(S3);
}
```

```
chicken2() {
  P(S2);
  printf("get");
  V(S1);
}
```

```
chicken3() {
  P(S3);
  printf("the");
  V(S1);
  P(S3);
  printf("side");
}
```


Klausuraufgabe: Synchronisierung

Why did the multithreaded chicken cross the road? Die drei Funktionen des folgenden Programms werden in jeweils eigenen Prozessen ausgeführt, die alle zur selben Zeit laufbereit werden. Sorgen Sie durch geeignete Synchronisation der Prozesse dafür, dass das Programm

to get to the other side

ausgibt. Dafür stehen Ihnen drei Semaphore zur Verfügung, die Sie geeignet initialisieren müssen. Setzen Sie an den freien Stellen Semaphor-Operationen (P, V) auf die Semaphore (S1, S2, S3) ein (z.B. `P(S1)`).