technische universität
dortmund

Betriebssysteme

Tafelübung 3. Deadlock

http://ess.cs.tu-dortmund.de/DE/Teaching/SS2017/BS/

Olaf Spinczyk

olaf.spinczyk@tu-dortmund.de
http://ess.cs.tu-dortmund.de/~os

AG Eingebettete Systemsoftware
Informatik 12, TU Dortmund

Agenda

- Besprechung Aufgabe 2: Threadsynchronisation
- Aufgabe 3: Deadlock
  - Semaphore
  - Problemvorstellung
  - Erkennen von Verklemmungen
  - Auflösen von Verklemmungen
  - Pthreads abbrechen
  - Makefiles
  - Alte Klausuraufgabe zu Semaphoren

Besprechung Aufgabe 2

- → Foliensatz Besprechung

Wiederholung Mutex

- Mutex dienen zum Schutz von kritischen Abschnitten
  - Nebenläufige Prozesse können hierdurch auf eine gemeinsame Datenstruktur zugreifen, ohne dass der Zustand der Datenstruktur inkonsisten wird.

```
/* Schlossvariable (Initialwert 0) */
pthread_mutex_t rangmutex;
…
int main(void){
   …
   pthread_mutex_init(&rangmutex, NULL);
   …
}

void* run(void* arg) {
   …
   pthread_mutex_lock(&rangmutex);
   rangliste[rang++] = team;
   pthread_mutex_unlock(&rangmutex);
   …
}
```

Kritische Abschnitte können auch durch Semaphore geschützt werden.

Dazu müssen wir aber zunächst wissen, was Semaphore sind und wie wir sie benutzen können.

Mutexe vs. Semaphoren

- Mit beiden können kritische Abschnitte geschützt werden.
- Beim Mutex kann immer nur ein Thread den kritischen Abschnitt betreten.
- Mit Semaphoren 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. Semaphoren können aber auch von unterschiedlichen Prozessen freigegeben werden.

Semaphoren

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

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 zu Semaphoren

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

  Eselsbrücke: „p(b)elegen, ggfs. vorher warten"

  Es sind danach weniger Ressourcen verfügbar, also wird runtergezählt.

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

  Eselsbrücke: „v(f)reigeben, ggfs. benachrichtigen"

  Es sind danach wieder mehr Ressourcen verfügbar, also wird hochgezählt.

Semaphor – Komplexere Interaktion

- Das erste Leser/Schreiber-Problem aus der Vorlesung

In diesem Beispiel soll ein kritischer Abschnitt geschützt werden. Es gibt zwei Klassen von konkurrierenden Prozessen:

- **Schreiber:** Sie ändern Daten und müssen daher gegenseitigen Ausschluss garantiert bekommen.
- **Leser:** Da sie nur Lesen, dürfen mehrere Leser auch gleichzeitig den kritischen Abschnitt betreten

Semaphor – Komplexes Beispiel

- Das erste Leser/Schreiber-Problem aus der Vorlesung

```
/* gem. Speicher */
Semaphore rcMutex;
Semaphore wrtMutex;
int readcount;
```

```
/* Initialisierung */
rcMutex = 1;
wrtMutex = 1;
readcount = 0;
```

```
/* Schreiber */
p(&wrtMutex);
... schreibe
v(&wrtMutex);
```

```
/* Leser */
p(&rcMutex);
readcount++;
/* Der erste Leser sperrt wrtMutex */
if (readcount == 1)
  p(&wrtMutex);
v(&rcMutex);

... lese

p(&rcMutex);
readcount--;
/* Der Letzte gibt wrtMutex frei */
if (readcount == 0)
   v(&wrtMutex);
v(&rcMutex):
```

POSIX Semaphoren

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

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

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

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

POSIX Semaphoren

- Belegen einer Semaphore (**P**), ggfs. müssen wir vorher warten:

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

- Freigeben einer Semaphore (**V**), ggfs. wird der nächste Thread benachrichtigt:

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

- Entfernen einer Semaphore:

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

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

POSIX Semaphore

- Beispiel: Gegenseitiger Ausschluss mit Semaphore

```
#include <semaphore.h>
/* Konsumierbare Rangsemaphore */
sem_t rangsemaphore;
...
int main(void){
   sem_init(&rangsemaphore, 0, 1);
   ...
   sem_destroy(&rangsemaphore);
   return 0;
}

void* run(void* arg) {
   ...
   sem_wait(&rangsemaphore);
   rangliste[rang++] = team;
   sem_post(&rangsemaphore);
   ...
}
```

globale Deklaration, damit auch Threads Zugriff haben

Fehlerbehandlung!

Wiederverwendbare Betriebsmittel

- Es kommt zu einer Verklemmung, wenn zwei Prozesse ein wiederwendbares Betriebsmittel belegt haben, das vom jeweils anderen hinzugefordert wird.
- Beispiel: Ein Rechnersystem hat **200 GByte** Hauptspeicher. Zwei Prozesse belegen den Speicher schrittweise. Die Belegung erfolgt blockierend.

Prozess 1

```
...
Belege 80 GByte;
...
Belege 60 GByte;
```

Prozess 2

```
...
Belege 70 GByte;
...
Belege 80 GByte;
```

Wenn beide Prozesse ihre erste Anforderung ausführen bevor Speicher nachgefordert wird, ist eine Verklemmung unvermeidbar.

Problemstellung

Das Szenario:

Ein Programm mit mehreren Threads erstellt neue Objekte. Dafür ist regelmäßig neuer Speicher notwendig.

Dafür fordert jeder Thread schrittweise den Speicher an, den er für dieses Objekt benötigt.

Eine Speicherverwaltung kümmert sich um das Zuweisen und Freigeben von Speicher.

Fordern die Threads größere Speicherbereiche an, als ihnen zugewiesen werden kann, warten sie darauf, dass genug Speicher verfügbar ist.

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:

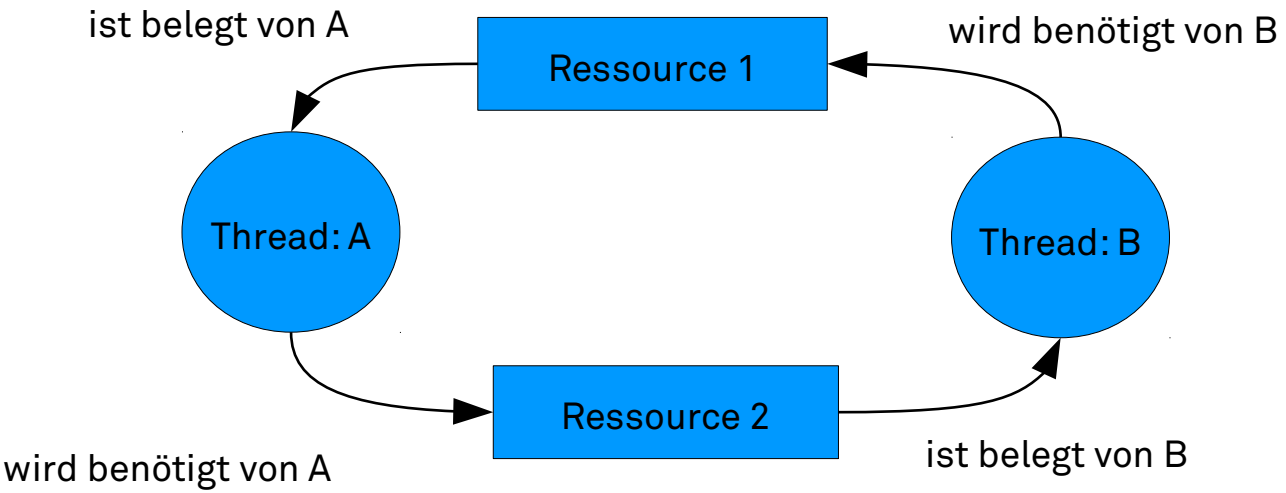
1. ***„mutual exclusion“***
   - die umstrittenen Betriebsmittel sind nur unteilbar nutzbar
2. ***„hold and wait“***
   - die umstrittenen Betriebsmittel sind nur schrittweise belegbar
3. ***„no preemption“***
   - die umstrittenen Betriebsmittel sind nicht rückforderbar

Erst wenn zur Laufzeit **eine weitere Bedingung** eintritt, liegt tatsächlich eine Verklemmung vor:

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

Deadlock-Voraussetzungen

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



Erkennen von Verklemmungen

- Analyse der Zustände aller Threads → mögliche Art der Deadlockerkennung
  - Die Größe des angefragten Speichers wird gespeichert, wenn die Anforderung nicht erfüllt werden kann.
  - Kann keinem Thread Speicher zugewiesen werden, aber alle fordern Speicher nach handelt es sich um eine Verklemmung
  - **Wichtig:** Der Zugriff auf solche Zustände muss synchronisiert werden! **(Race Conditions)**

Pthreads abbrechen

```
int pthread_cancel(pthread_t thread);
```

- Sendet eine Anfrage zum Abbrechen des Pthreads
  - Parameter
    - thread: Thread-Objekt
  - Rückgabewert:
    - 0, wenn erfolgreich
    - eine Fehlernummer ungleich 0 im Fehlerfall

- Wann der Thread auf den Abbruch reagiert hängt vom:
  - gesetzten Abbruchzustand (***enabled*** oder *disabled*)
  - und dem gesetzten Abbruchtyp ab (*asynchronous* oder ***deferred***).

Verklemmungsvermeidung (1)

- engl. deadlock aviodance
- Verhinderung von zirkulärem Warten (**im laufenden System**) durch strategische Maßnahmen:
  - keine der ersten drei notwendigen Bedingungen wird entkräftet
  - Fortlaufende **Bedarfsanalyse** schließt zirkuläres Warten aus

Verklemmungsvermeidung (2)

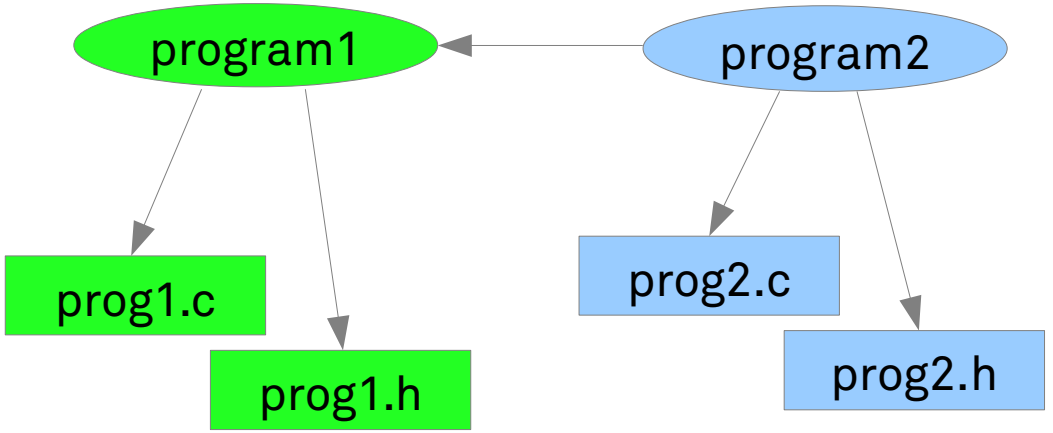
- Betriebsmittelanforderungen der Prozesse sind zu steuern:
  - **„sicherer Zustand“** muss immer beibehalten werden:
    - es existiert eine Prozessabfolge, bei der jeder Prozess seinen maximalen Betriebsmittelbedarf decken kann
  - **„unsichere Zustände“** werden umgangen:
    - Zuteilungsablehnung im Falle nicht abgedeckten Betriebsmittelbedarfs
    - anfordernde Prozesse nicht bedienen bzw. frühzeitig suspendieren
- Problem: À priori Wissen über den maximalen Betriebsmittelbedarf ist erforderlich.
  - In unserem Beispiel: Von jedem Thread muss die Gesamtgröße des reservierten Speichers bekannt sein.

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**

Client-/Server-Beispiel

```
# -= 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: chatclient chatserver     # erstes Target = Default-Target

chatclient: chatclient.c chatshm.c chatshm.h sync.c sync.h
    $(CC) $(CFLAGS) -o chatclient chatclient.c chatshm.c sync.c
chatserver: chatserver.c chatshm.c chatshm.h sync.c sync.h
    $(CC) $(CFLAGS) -o chatserver chatserver.c chatshm.c sync.c

clean:
    rm -f chatclient chatserver *.o

.PHONY: all clean              # “unechte” (phony) Targets
```

Makefiles und make(1)

```
hsc@ios:~/A2/vorgaben$ ls
chatclient  chatclient.c  chatserver  chatserver.c  chatshm.c  chatshm.h
Makefile  sync.c  sync.h
hsc@ios:~/A2/vorgaben$ make clean
rm -f chatclient chatserver *.o
hsc@ios:~/A2/vorgaben$ make
gcc -Wall (...) -o chatclient chatclient.c chatshm.c sync.c
gcc -Wall (...) -o chatserver chatserver.c chatshm.c sync.c
hsc@ios:~/A2/vorgaben$ touch chatserver.c
hsc@ios:~/A2/vorgaben$ make
gcc -Wall (...) -o chatserver chatserver.c chatshm.c sync.c
hsc@ios:~/A2/vorgaben$ make
make: Nothing to be done for `all'.
hsc@ios:~/A2/vorgaben$ touch sync.h
hsc@ios:~/A2/vorgaben$ make
gcc -Wall (...) -o chatclient chatclient.c chatshm.c sync.c
gcc -Wall (...) -o chatserver chatserver.c chatshm.c sync.c
hsc@ios:~/A2/vorgaben$
```

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

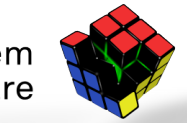
ausgibt. Dafür stehen Ihnen drei Semaphore zur Verfügung, die Sie geeignet initialisieren müssen. Setzen Sie an den freien Stellen Semaphor-Operationen (P, V) auf die Semaphore (S1, S2, S3) ein (z.B. `P(S1)`).

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)`).