technische universität
dortmund

Betriebssysteme

Tafelübung 3. Deadlock

https://ess.cs.tu-dortmund.de/DE/Teaching/SS2019/BS/

Horst Schirmeier

horst.schirmeier@tu-dortmund.de
https://ess.cs.tu-dortmund.de/~hsc

AG Eingebettete Systemsoftware
Informatik 12, TU Dortmund

mit Material von Olaf Spinczyk,
Universität Osnabrück

Agenda

- Besprechung Aufgabe 2: Threadsynchronisation
- Aufgabe 3: Deadlock
  - Makefiles
  - Enumeration
  - Problemvorstellung
  - Wiederverwendbare/Konsumierbare Betriebsmittel
  - Voraussetzungen für Verklemmungen
  - Verklemmungsauflösung
- Alte Klausuraufgabe zu Semaphoren

Besprechung Aufgabe 2

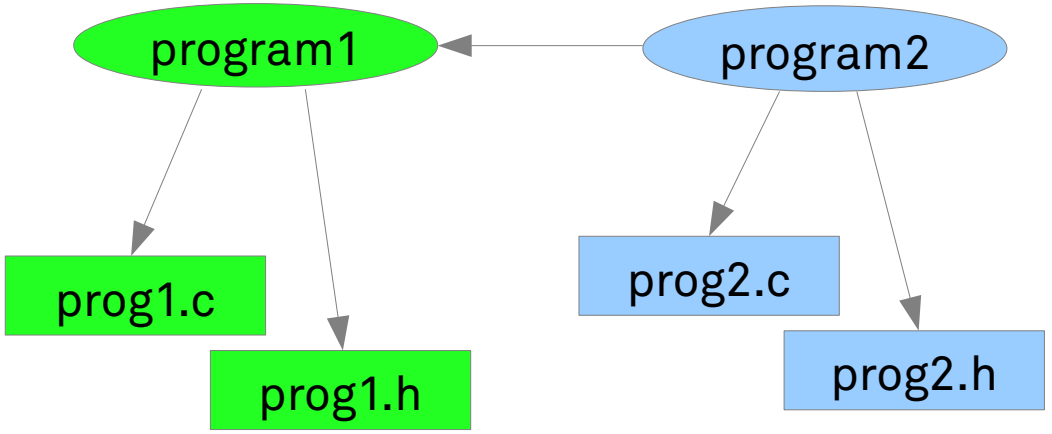
- → Foliensatz Besprechung

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**

Enumeration

- ***enum*** ist ein benutzerdefinierter Aufzählungstyp
- eine mittels ***enum*** definierte Variable ist vom „Typ ***int***“
  - intern Ganzzahl, aber durch Wertbezeichner visualisiert
  - kann dort verwendet werden, wo ein ***int*** erlaubt ist

```
#include <stdio.h>

enum farbpalette {ROT, GRUEN, BLAU, GELB};
enum farbpalette farbe;

int main(void) {
    farbe = GRUEN;
    printf("gewaehlte Farbe: %d\n", farbe);
    return 0;
}
```

```
mm@ios:~$ ./enum
gewaehlte Farbe: 1
```

Problemstellung: Szenario

- 2 Mitarbeiter (in der Vorgabe: **Pthreads** A + B)
  - teilen sich die Arbeit, eine Klausur zu korrigieren
  - machen regelmäßig Pausen
- Zum Korrigieren benötigte Ressourcen (**beide** benötigt!):
  - R1: Karton mit Klausuren (**Semaphore**)
  - R2: Liste für Noten (**Semaphore**)
- Ressourcennutzungsstrategie:
  - Ressource nicht frei → **Warten**, bis sie freigegeben wird
  - Dabei werden bereits erhaltene Ressourcen **nicht zurückgegeben**.
  - Pause / Korrektur fertig → **Freigabe** beider Ressourcen

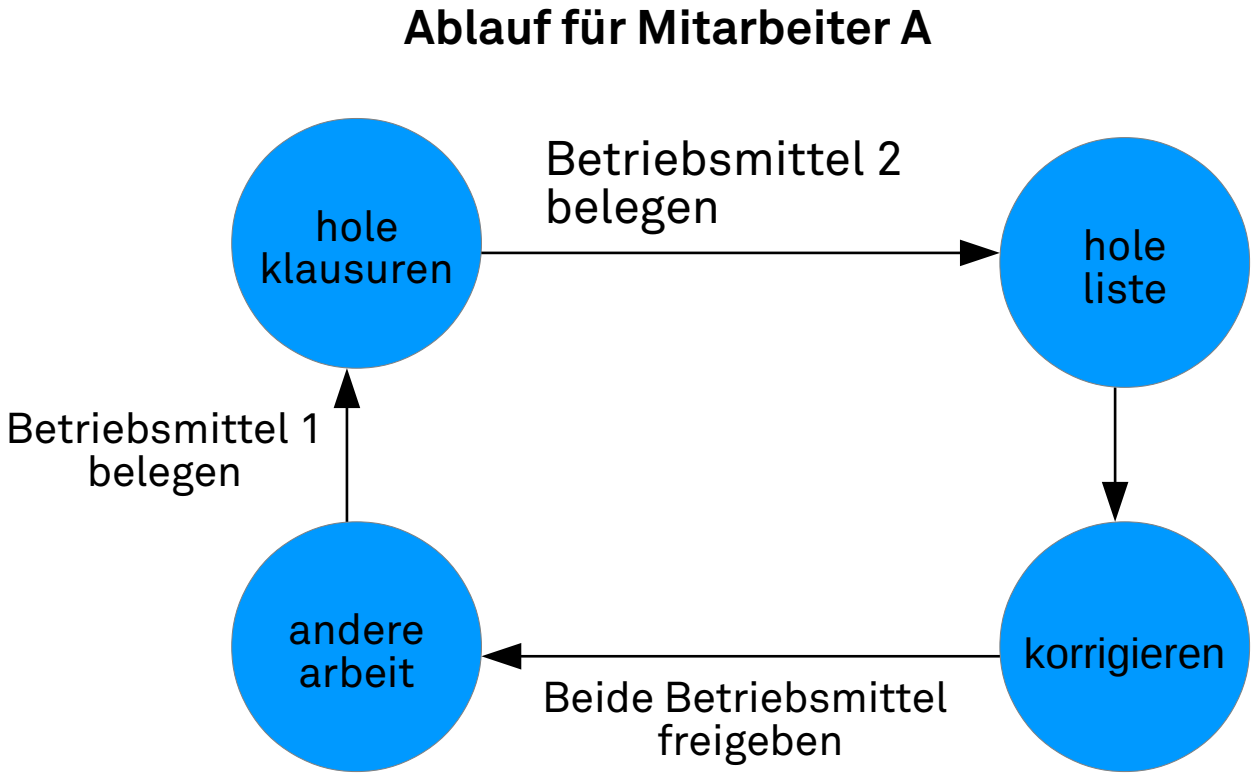
Problemstellung: Ablauf für Mitarbeiter

Die Threads führen folgende Aufgaben regelmäßig durch:

Schritt	**Mitarbeiter A**	**Mitarbeiter B**
1	Andere Arbeit erledigen	
2	Semaphore Klausuren belegen	Semaphore Liste belegen
3	Semaphore Liste belegen	Semaphore Klausuren belegen
4	Korrigieren	
5	Klausuren und Liste freigeben	
6	Weiter mit Schritt 1	

Problemstellung: Zustände

- 4 Zustände: andere_arbeit, hole_klausuren, hole_liste und korrigieren



Betriebsmittel ...

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

- **Wiederverwendbare Betriebsmittel**
  - Werden für eine bestimmte Zeit belegt und anschließend wieder freigegeben.
  - Beispiele: CPU, Haupt- und Hintergrundspeicher
  - Typische Zugriffssynchronisation: **Gegenseitiger Ausschluss**
- **Konsumierbare Betriebsmittel**
  - Werden im laufenden System erzeugt (produziert) und zerstört (konsumiert)
  - Beispiele: Unterbrechungsanforderungen, Daten von Eingabegeräten
  - Typische Zugriffssynchronisation: **Einseitige Synchronisation**

Siehe auch Vorlesung 6 S. 14

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.

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

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)

Pthreads abbrechen

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

- Sendet eine Anfrage zum Abbrechen des Pthreads
  - Parameter
    - thread: Thread-Objekt
  - Rückgabewert:
    - 0, wenn erfolgreich
    - ≠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***).

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)`).