Übungen Betriebssysteme (BS)

U3 – Deadlocks

https://moodle.tu-dortmund.de/course/view.php?id=34604

Peter Ulbrich

peter.ulbrich@tu-dortmund.de

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

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

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

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***

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***

Semaphor-Eselsbrücken

Semaphor-Eselsbrücken

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

  » p(b)***elegen*, ggfs. vorher warten «**

→ Danach sind weniger *Ressourcen* verfügbar, also wird *runter*gezählt.

Semaphor-Eselsbrücken

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

  » p(b)*elegen*, ggfs. vorher warten «

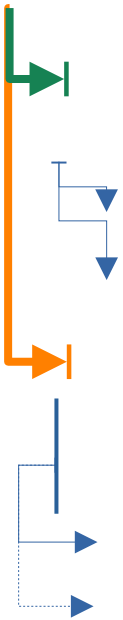
  → Danach sind weniger *Ressourcen* verfügbar, also wird *runter*gezählt.

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

  » v(f)reigeben, ggfs. Benachrichtigen «

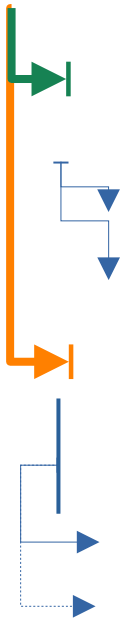
  → Danach sind mehr *Ressourcen* verfügbar, also wird *hoch*gezählt.

Mutexe vs. Semaphore



Mutexe vs. Semaphore

- ✔ Beides schützt den kritische Abschnitte
- **Mutex**: immer nur 1 Thread kann in den *kritischen Abschnitt*
- **Semaphor**: n Threads können in den *kritischen Abschnitt*
  - Nützlich für *Betriebssystemressourcen*, wo nur eine bestimmte Anzahl zur Verfügung steht.

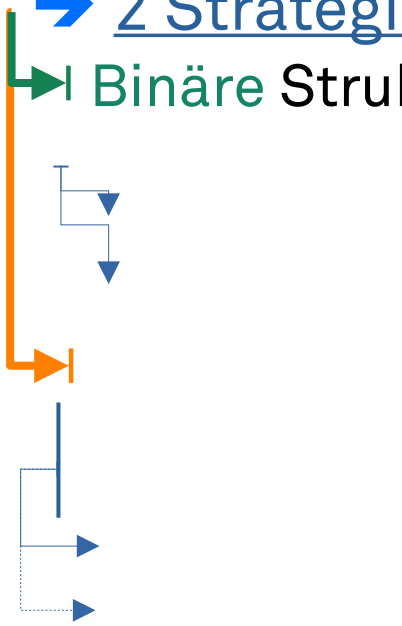


Mutexe vs. Semaphore

- ✔ Beides schützt den kritische Abschnitte
- **Mutex:** immer nur 1 Thread kann in den *kritischen Abschnitt*
- **Semaphor**: n Threads können in den *kritischen Abschnitt*
  - Nützlich für *Betriebssystemressourcen*, wo nur eine bestimmte Anzahl zur Verfügung steht.

➔ 2 Strategien: **Semaphor** *als* **Mutex**

- Binäre Struktur **behalten** *(direkte Übersetzung)*



Mutexe vs. Semaphore

- ✔ Beides schützt den kritische Abschnitte
- **Mutex**: immer nur 1 Thread kann in den *kritischen Abschnitt*
- **Semaphor**: n Threads können in den *kritischen Abschnitt*
  - Nützlich für *Betriebssystemressourcen*, wo nur eine bestimmte Anzahl zur Verfügung steht.

→ 2 Strategien: **Semaphor** *als* **Mutex**

- Binäre Struktur **behalten** *(direkte Übersetzung)*
  - ✔ `Mutex` → binären Semaphor
  - `Lock()` → `sem_wait()`
  - `Unlock()` → `sem_post()`

Mutexe vs. Semaphore

- ✔ Beides schützt den kritische Abschnitte
- **Mutex**: immer nur 1 Thread kann in den *kritischen Abschnitt*
- **Semaphor**: n Threads können in den *kritischen Abschnitt*
  - Nützlich für *Betriebssystemressourcen*, wo nur eine bestimmte Anzahl zur Verfügung steht.

→ 2 Strategien: **Semaphor** *als* **Mutex**

- Binäre Struktur **behalten** *(direkte Übersetzung)*
  - ✔ `Mutex` → binären Semaphor
  - `Lock()` → `sem_wait()`
  - `Unlock()` → `sem_post()`
- Strukturell ***anpassen*** *(Mutex + geteilte Zählvariable → Semaphor)*

Mutexe vs. Semaphore

- ✔ Beides schützt den kritische Abschnitte
- **Mutex**: immer nur 1 Thread kann in den *kritischen Abschnitt*
- **Semaphor**: n Threads können in den *kritischen Abschnitt*
  - Nützlich für *Betriebssystemressourcen*, wo nur eine bestimmte Anzahl zur Verfügung steht.

→ 2 Strategien: **Semaphor** *als* **Mutex**

- Binäre Struktur **behalten** *(direkte Übersetzung)*
  - ✔ `Mutex` → binären Semaphor
  - `Lock()` → `sem_wait()`
  - `Unlock()` → `sem_post()`
- Strukturell ***anpassen*** *(Mutex + geteilte Zählvariable → Semaphor)*
  - ✗ `Mutex` (*entfernen*)

Mutexe vs. Semaphore

- ✔ Beides schützt den kritische Abschnitte
- **Mutex**: immer nur 1 Thread kann in den *kritischen Abschnitt*
- **Semaphor**: n Threads können in den *kritischen Abschnitt*
  - Nützlich für *Betriebssystemressourcen*, wo nur eine bestimmte Anzahl zur Verfügung steht.

→ 2 Strategien: **Semaphor** *als* **Mutex**

- Binäre Struktur **behalten** *(direkte Übersetzung)*
  - ✔ `Mutex` → binären Semaphor
  - `Lock()` → `sem_wait()`
  - `Unlock()` → `sem_post()`
- Strukturell ***anpassen*** *(Mutex + geteilte Zählvariable → Semaphor)*
  - ✗ `Mutex` (*entfernen*)
  - ✔ `Counter` → zählender Semaphor
  - `sem_wait()`: Überprüfen/Dekrementieren wird **atomar**
  - [*ggf.* `sem_trywait()`, um Thread bei Zähler == `0` zu beenden]

Mutexe vs. Semaphore

✔ Beides schützt den kritische Abschnitte

- **Mutex**: immer nur 1 Thread kann in den *kritischen Abschnitt*
- **Semaphor**: n Threads können in den *kritischen Abschnitt*
  - Nützlich für *Betriebssystemressourcen*, wo nur eine bestimmte Anzahl zur Verfügung steht.

➔ 2 Strategien: **Semaphor** *als* **Mutex**

- Binäre Struktur **behalten** *(direkte Übersetzung)*
  - ✔ `Mutex` → binären Semaphor
  - `Lock()` → `sem_wait()`
  - `Unlock()` → `sem_post()`
- Strukturell ***anpassen*** *(Mutex + geteilte Zählvariable → Semaphor)*
  - ✗ `Mutex` (*entfernen*)
  - ✔ `Counter` → zählender Semaphor
  - `sem_wait()`: Überprüfen/Dekrementieren wird **atomar**
  - [*ggf.* `sem_trywait()`, um Thread bei Zähler == `0` zu beenden]

➔ Für **n=1** verhält sich ein **Semaphor** ähnlich wie ein **Mutex**.
Semaphore können auch *von* einem *anderen Prozess* *freigegeben* werden als dem, der sie belegt hat.

Semaphor – Beispiel (1)

- Gemeinsam genutzte FIFO-Queue mit maximal 100 Elementen

Semaphor – Beispiel (1)

- 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;
```

```
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 (1)

- 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;
```

```
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;
}
```

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

Semaphor – Beispiel (2)

- Zusätzlich noch einseitige Synchronisation

Semaphor – Beispiel (2)

- Zusätzlich noch einseitige Synchronisation

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

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

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

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

Semaphor – Beispiel (2)

- Zusätzlich noch einseitige Synchronisation

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

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

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

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

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

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);
```

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

- 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

Timespec - Beispiel

```c
#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

Betriebsmittel

- ***Betriebsmittel:*** werden vom Betriebssystem verwaltet und den Prozessen zugänglich gemacht.

Betriebsmittel

- ***Betriebsmittel:*** werden vom Betriebssystem verwaltet und den Prozessen zugänglich gemacht.

Man *unterscheidet zwei Arten:*

Betriebsmittel

- ***Betriebsmittel:*** werden vom Betriebssystem verwaltet und den Prozessen zugänglich gemacht.

  Man *unterscheidet zwei Arten:*

- **Wiederverwendbare Betriebsmittel**

Betriebsmittel

- ***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**

Betriebsmittel

- ***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**

Betriebsmittel

- ***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**

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:

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

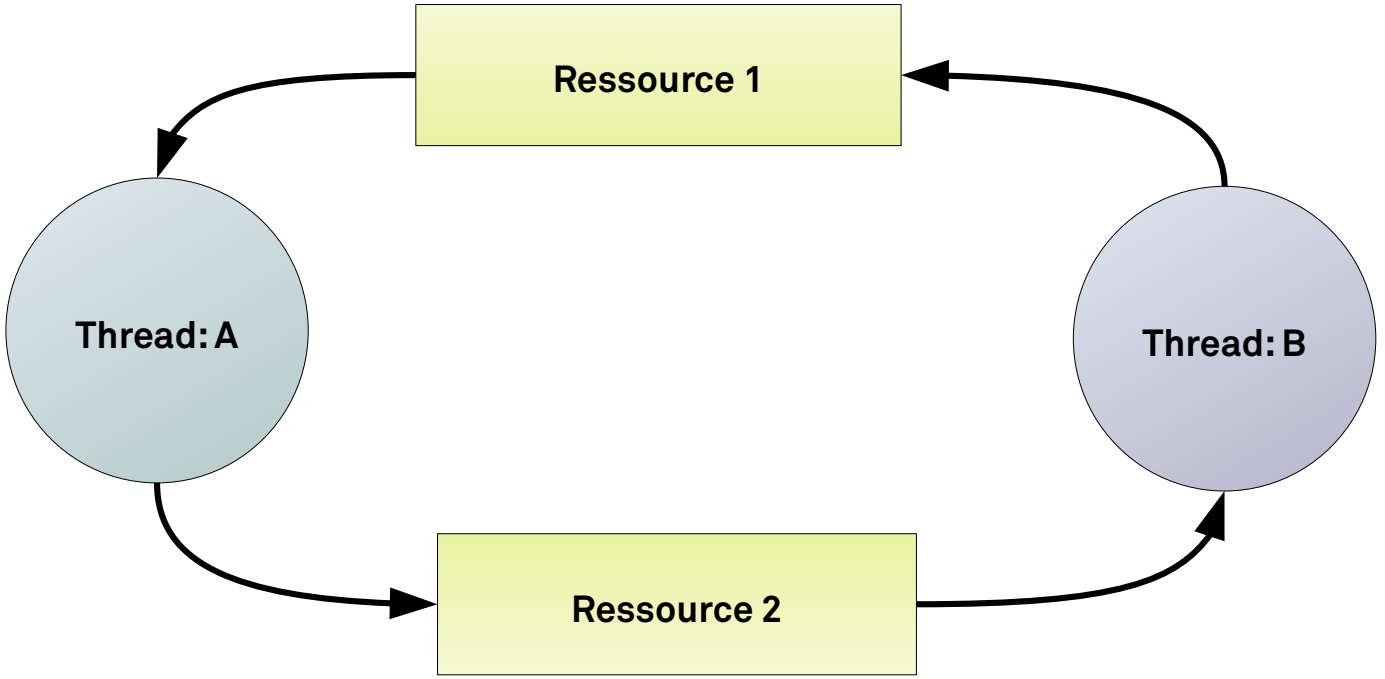
4. »circular wait«

- eine geschlossene Kette wechselseitig wartender Prozesse

Deadlock-Voraussetzungen

4. »circular wait«

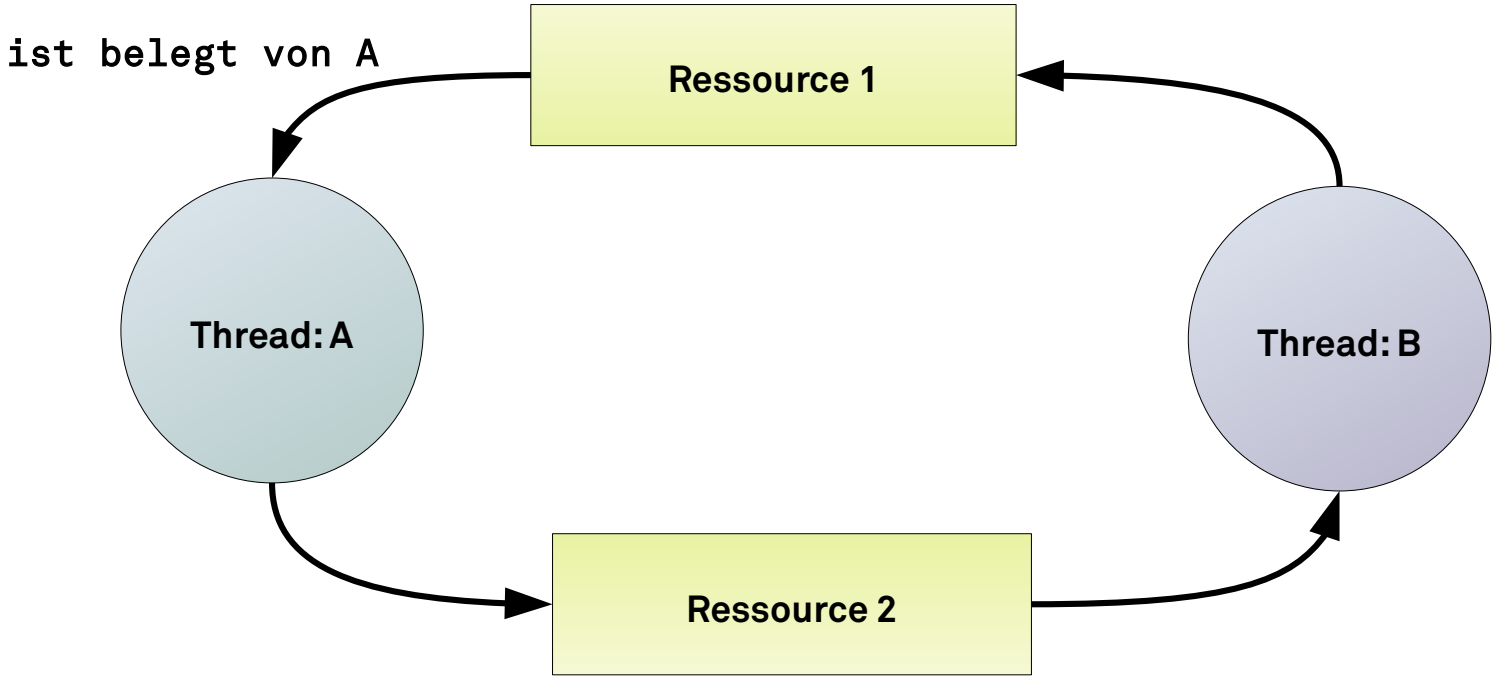
- eine geschlossene Kette wechselseitig wartender Prozesse



Deadlock-Voraussetzungen

4. »circular wait«

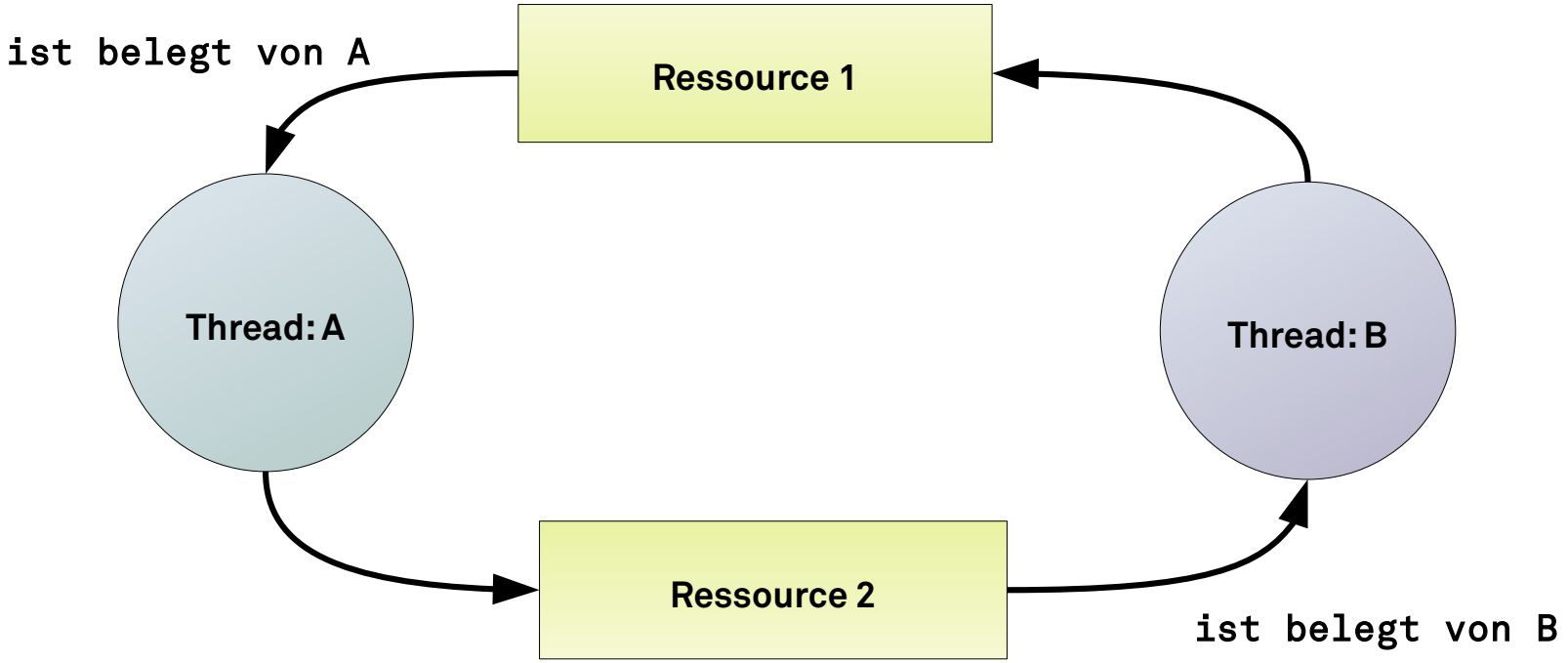
- eine geschlossene Kette wechselseitig wartender Prozesse



Deadlock-Voraussetzungen

4. »circular wait«

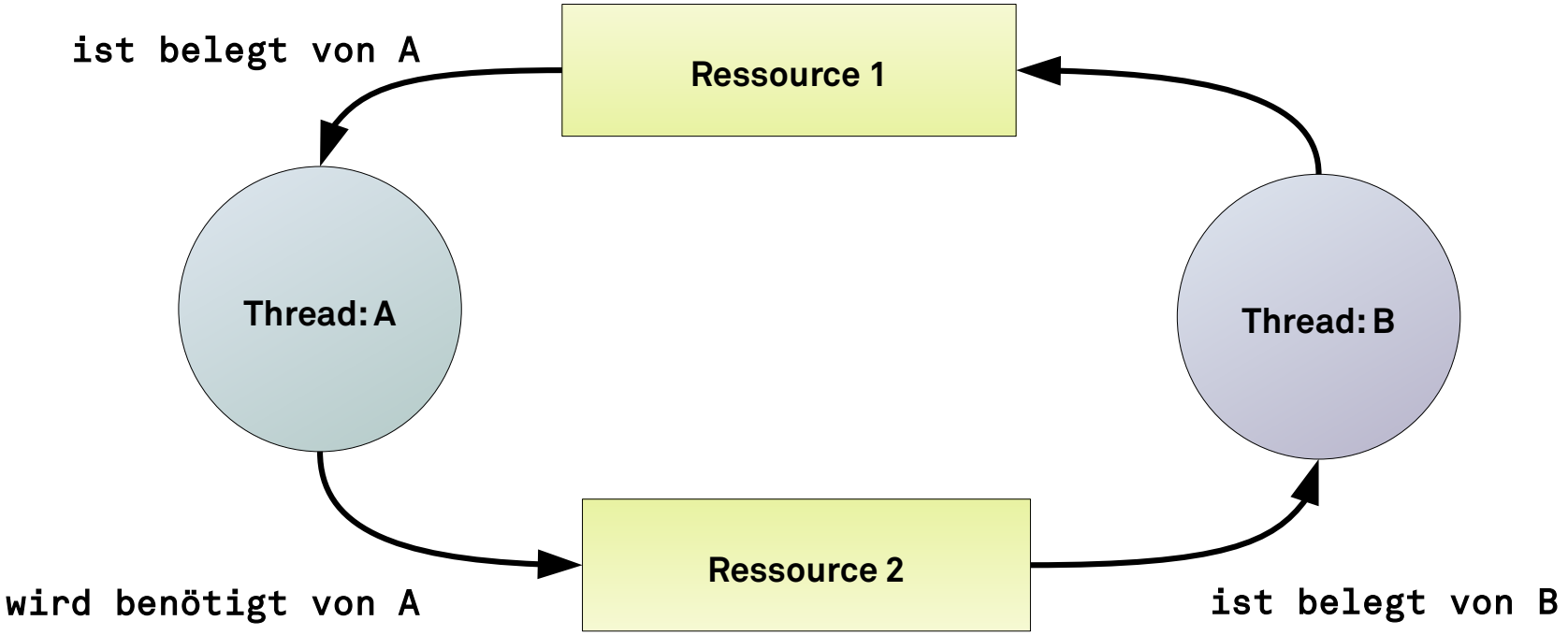
- eine geschlossene Kette wechselseitig wartender Prozesse



Deadlock-Voraussetzungen

4. »circular wait«

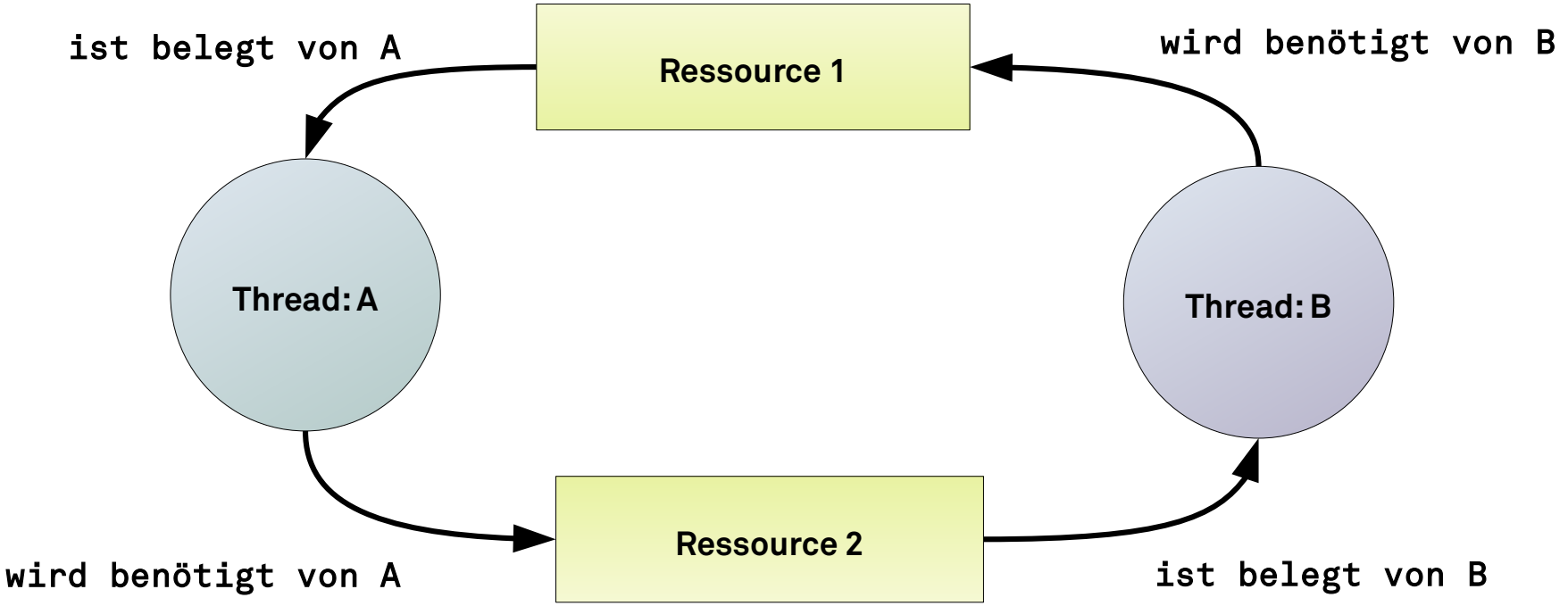
- eine geschlossene Kette wechselseitig wartender Prozesse



Deadlock-Voraussetzungen

4. »circular wait«

- eine geschlossene Kette wechselseitig wartender Prozesse



Verklemmungsauflösung

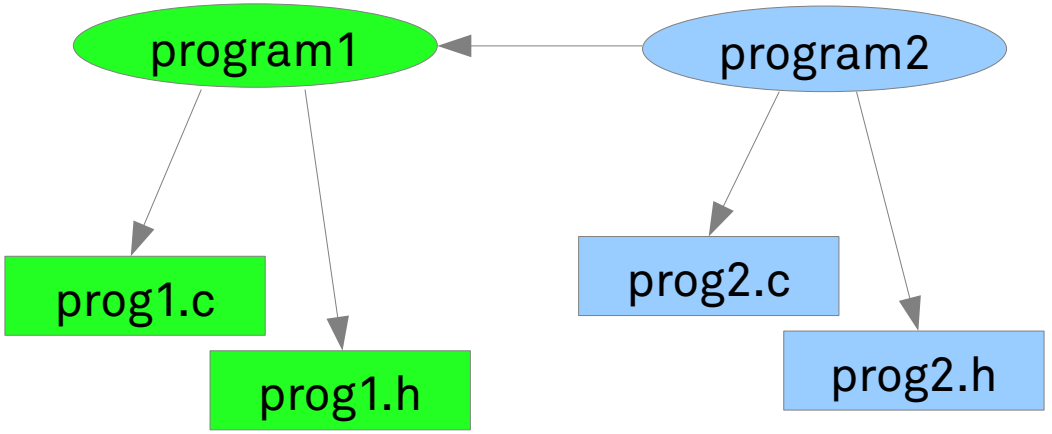
- Die „einfachste“ Variante: **Prozesse abbrechen** und so Betriebsmittel frei bekommen
  - Verklemmte Prozesse schrittweise abbrechen (*großer Aufwand*)
  - Mit dem **„*effektivsten Opfer*“** (**?**) beginnen
  - Oder: alle verklemmten Prozesse terminieren (*großer Schaden*)

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)`).