Übungen Betriebssysteme (BS)

U3 – Verklemmungen

Alwin Berger

TU Dortmund - AG Systemsoftware

Agenda

- Besprechung von A2
- Fortsetzung der C-Einführung
- Aufgabe 3: Deadlock
  - Semaphor
  - Verschiedene Arten von Betriebsmitteln
  - Verklemmungen und deren Auflösung
  - Makefiles
- Alte Klausuraufgabe zur Synchronisation

Semaphore

- Eine „nicht-negative ganze Zahl”, für die zwei unteilbare Operationen definiert sind:
  P (niederländisch prolaag, „erniedrigen”, auch *down*, *wait*)
  - hat der Semaphore den Wert 0, wird der laufende Prozess blockiert
  - ansonsten wird der Semaphor um 1 dekrementiert

  V (niederländisch verhoog, „erhöhe”, auch *up*, *signal*)
  - auf den Semaphor ggf. blockierte Prozesse wird deblockiert
  - ansonsten wird der Semaphor um 1 inkrementiert
- Eine *Betriebssystemabstraktion* zum Austausch von Synchronisationssignalen zwischen nebenläufigen arbeitenden Prozessen.

Eselsbrückchen zu den Semaphor-Operationen

- Mit *P* wartet man auf eine Ressource und belegt diese:
  „**p**elegen, ggf. vorher warten”
  - Danach sind **weniger Ressourcen** verfügbar, also wird runtergezählt.

- Mit *V* wird eine Ressource wieder freigegeben, ggf. wird der nächste wartende Thread benachrichtigt:
  „**v**reigeben, ggf. benachrichtigen”
  - Danach sind **mehr Ressourcen** verfügbar, also wird hochgezählt.

Unterschied zwischen Mutex und Semaphor

- Beide Synchronisationsmittel schützen einen kritischen Abschnitt
- **Mutex**
  - Immer **nur 1 Faden** kann in den kritischen Abschnitt
  - Sperren und Entsperren müssen von [**demselben**] Faden erfolgen
- **Semaphor**
  - **n Fäden** können in den kritischen Abschnitt

Mutex durch Semaphor darstellen:

- Realisiert durch **binären Semaphor** (n=1)
  - Initialisierung mit 1

```
lock() -> sem_wait()
unlock() -> sem_post()
```

Semaphor-Beispiel 1 (1)

- Gemeinsam genutzte FIFO-Queue mit max. 100 Elementen
- Gleichzeitig darf max. 1 Faden zugreifen

```
01 /* gem. Speicher */
02 Semaphore lock;
03 Semaphore freiePlaetze;
04
05 struct List * queue;
06 /* Initialisierung */
07 lock = 1;
08 freiePlaetze = 100;
09
10 queue->head = NULL;
11 queue->tail = NULL;
12
13 void enqueue(element *item){
14   if (item != NULL){
15     p(&freiePlaetze);
16     p(&lock);
17     queue->tail = item;
18     // ...
19     v(&lock);
20   }
21 }
22
23 element * dequeue(){
24   p(&lock);
25   element *item = queue->head;
26   // ...
27   if (item != NULL){
28     v(&freiePlaetze);
29   }
30   v(&lock);
31   return item;
32 }
```

Semaphor-Beispiel 1 (2)

- Gemeinsam genutzte FIFO-Queue mit max. 100 Elementen
- Gleichzeitig darf max. 1 Faden zugreifen

```
01 /* gem. Speicher */
02 Semaphore lock;
03 Semaphore freiePlaetze;
04
05 struct List * queue;
06 /* Initialisierung */
07 lock = 1;
08 freiePlaetze = 100;
09
10 queue->head = NULL;
11 queue->tail = NULL;
12
13 void enqueue(element *item){
14   if (item != NULL){
15     p(&freiePlaetze);
16     p(&lock);
17     queue->tail = item;
18     // ...
19     v(&lock);
20   }
21 }
22
23 element * dequeue(){
24   p(&lock);
25   element *item = queue->head;
26   // ...
27   if (item != NULL){
28     v(&freiePlaetze);
29   }
30   v(&lock);
31   return item;
32 }
```

Semaphor-Beispiel 1 (3)

- Gemeinsam genutzte FIFO-Queue mit max. 100 Elementen
- Gleichzeitig darf max. 1 Faden zugreifen

```
01 /* gem. Speicher */
02 Semaphore lock;
03 Semaphore freiePlaetze;
04
05 struct List * queue;
06 /* Initialisierung */
07 lock = 1;
08 freiePlaetze = 100;
09
10 queue->head = NULL;
11 queue->tail = NULL;
12
13 void enqueue(element *item){
14   if (item != NULL){
15     p(&freiePlaetze);
16     p(&lock);
17     queue->tail = item;
18     // ...
19     v(&lock);
20   }
21 }
22
23 element * dequeue(){
24   p(&lock);
25   element *item = queue->head;
26   // ...
27   if (item != NULL){
28     v(&freiePlaetze);
29   }
30   v(&lock);
31   return item;
32 }
```

Semaphor-Beispiel 2 (1)

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

```
01 /* gem. Speicher */
02 Semaphore verfuegbar;
03 /* Initialisierung */
04 verfuegbar = 0;
05
06 void producer(){
07   while(1){
08     element *e = produce();
09     enqueue(e);
10     v(&verfuegbar);
11   }
12 }
```

```
13 void consumer(){
14   while(1){
15     p(&verfuegbar);
16     element *e = dequeue();
17     consume(e);
18   }
19 }
```

Semaphor-Beispiel 2 (2)

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

```
01 /* gem. Speicher */
02 Semaphore verfuegbar;
03 /* Initialisierung */
04 verfuegbar = 0;
05
06 void producer(){
07   while(1){
08     element *e = produce();
09     enqueue(e);
10     v(&verfuegbar);
11   }
12 }
```

```
13 void consumer(){
14   while(1){
15     p(&verfuegbar);
16     element *e = dequeue();
17     consume(e);
18   }
19 }
```

sem_init(3)

```
#include <semaphore.h>
int sem_init(sem_t *sem, int pshared, unsigned int value);
```

- Initialisiert einen Semaphore mit dem Wert von value.
- Argumente:
  - **sem**: Zeiger auf zu initialisierende Semaphor
  - **pshared**: 0, falls sem nur zwischen Threads eines Prozesses verwendet wird
  - **value**: Initialer Wert des Semaphors, entspricht dem n
- Rückgabewerte:
  - =0, wenn erfolgreich
  - !=0, wenn ein Fehler aufgetreten ist
- Gegenstück: int sem_destroy(sem_t *sem);

sem_wait(3)

```
#include <semaphore.h>
int sem_wait(sem_t *sem);
```

- Dekrementiert den Semaphor und betritt den kritischen Abschnitt
- Legt den aufrufenden Thread schlafen, wenn der Wert bereits 0 ist
- Argumente:
  - **sem**: Ein Zeiger auf den zu verwendenden Semaphor
- Rückgabewerte:
  - =0, wenn erfolgreich
  - !=0, wenn ein Fehler aufgetreten ist
- Gegenstück: int sem_post(sem_t *sem);

Betriebsmittel…

… werden vom Betriebssystem verwendet 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
  - Typisches Synchronisationsschema: **Gegenseitiger Ausschluss**
- **Konsumierbare Betriebsmittel**
  - Werden im laufenden System erzeugt (produziert) und zerstört (konsumiert)
  - Beispiel: Unterbrechungsanforderungen, Signale, Nachrichten, Daten von Eingabegeräten
  - Typisches Synchronisationsschema: **Einseitige Synchronisation**

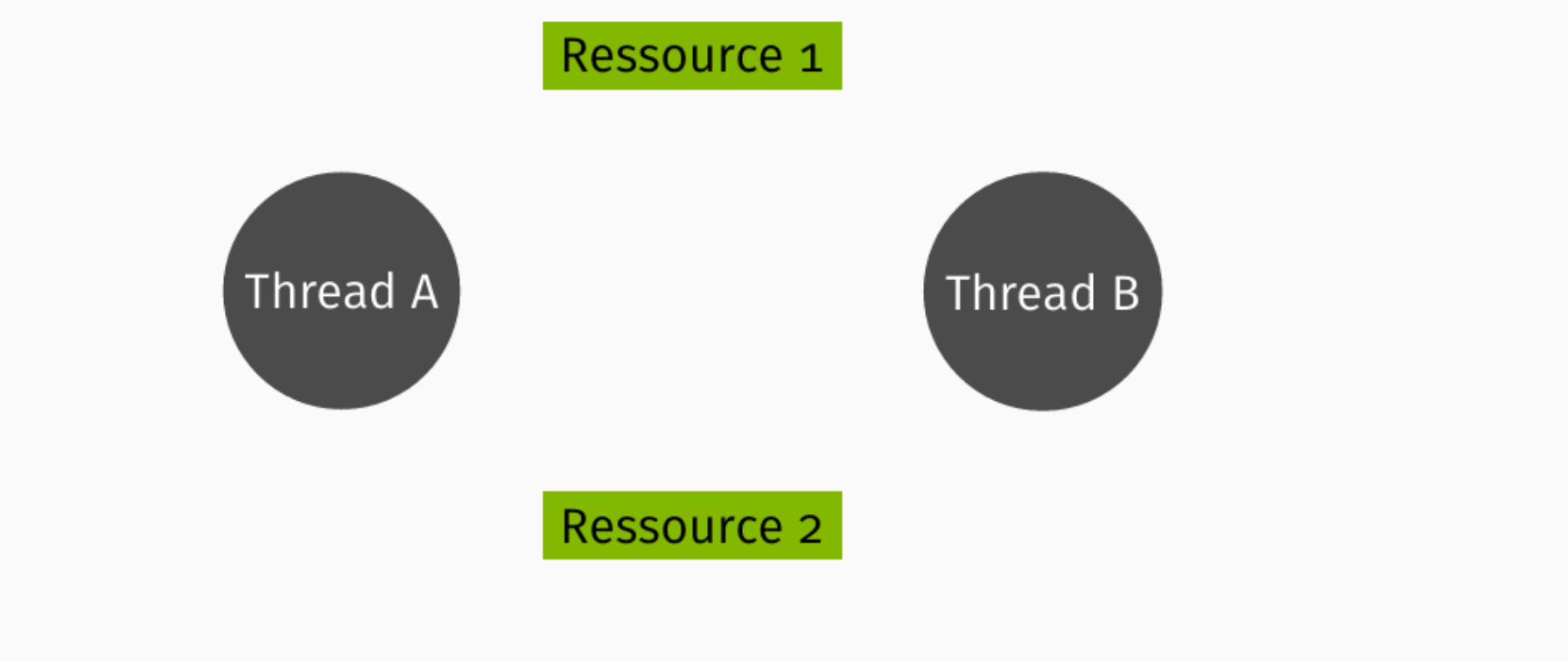
Vorraussetzungen für eine Verklemmung (1)

Die notwendigen Bedingungen für eine Verklemmung:

- **mutual exclusion:** Betriebsmittel sind nur **unteilbar** nutzbar
- **hold and wait:** Betriebsmittel sind nur **schrittweise** belegbar
- **no preemption:** Betriebsmittel können **nicht zurückgefordert** werden
- **circular wait:** eine geschlossene Kette wechselseitig wartender Fäden

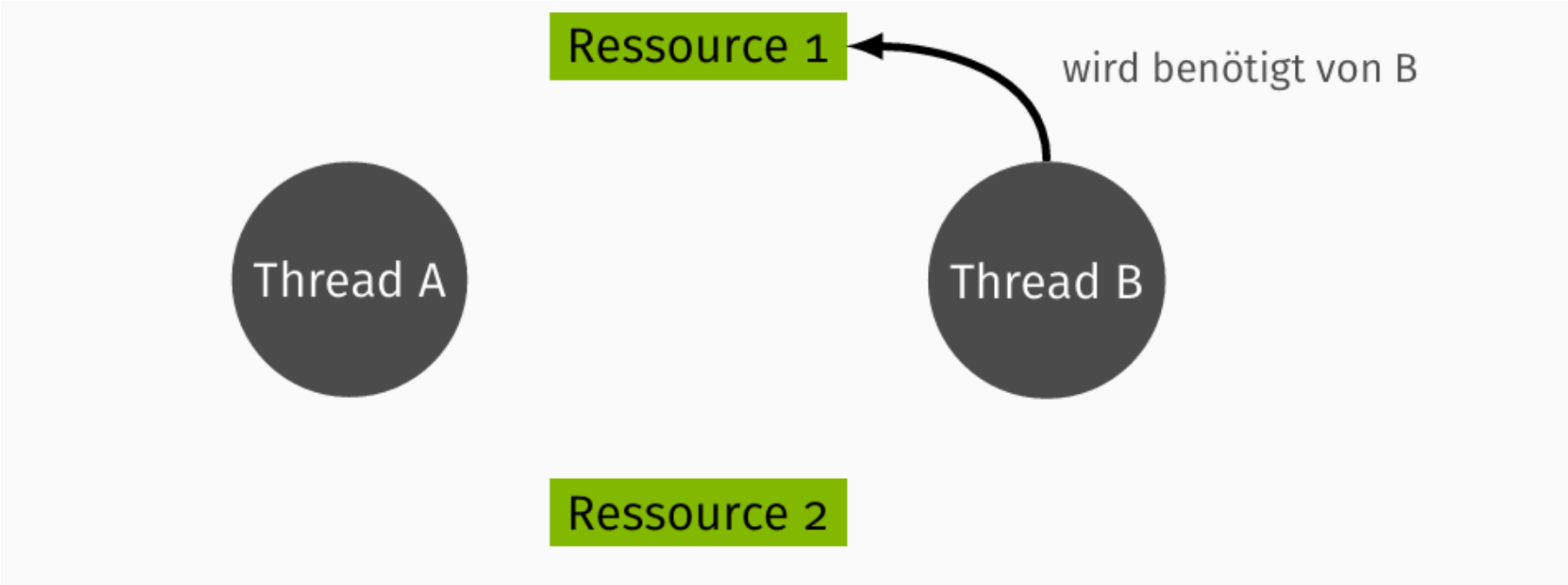
Vorraussetzungen für eine Verklemmung (2)

4. **circular wait:** eine geschlossene Kette wechselseitig wartender Fäden



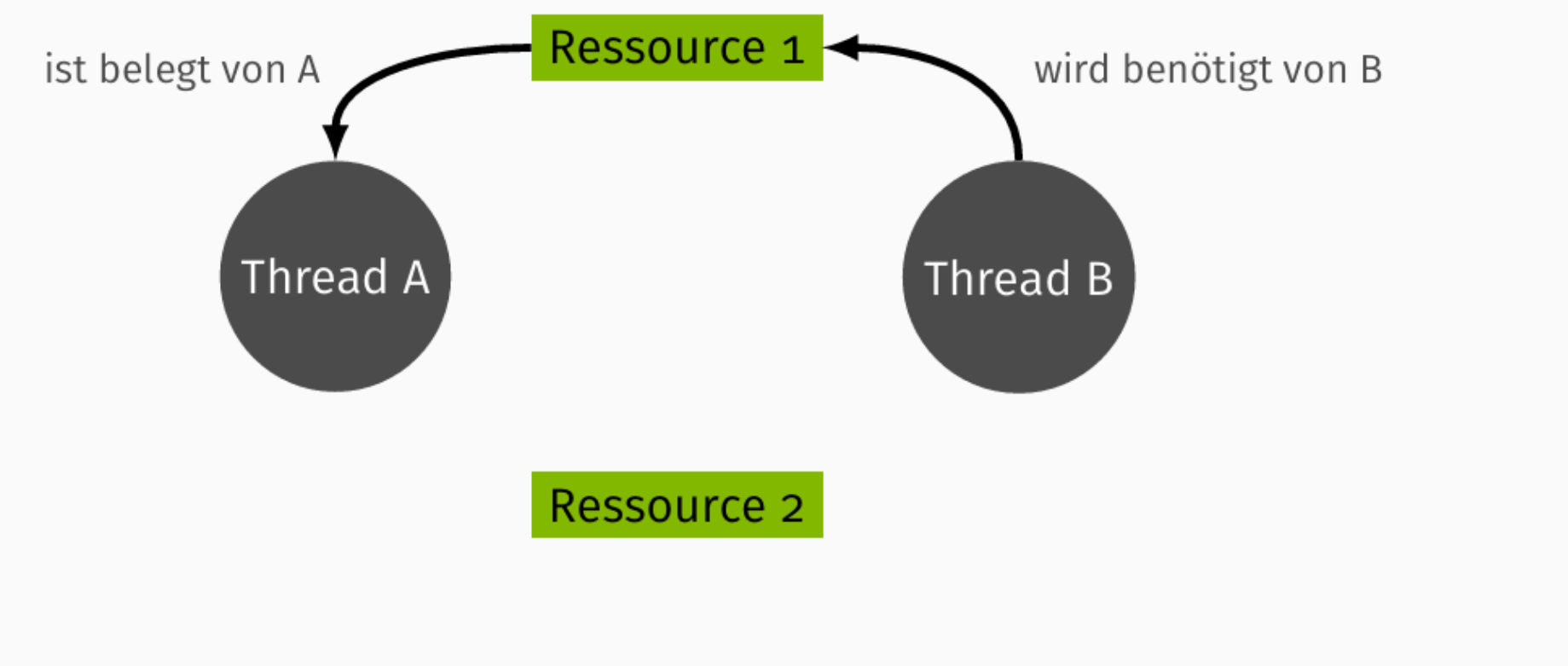
Vorraussetzungen für eine Verklemmung (2)

4. **circular wait:** eine geschlossene Kette wechselseitig wartender Fäden



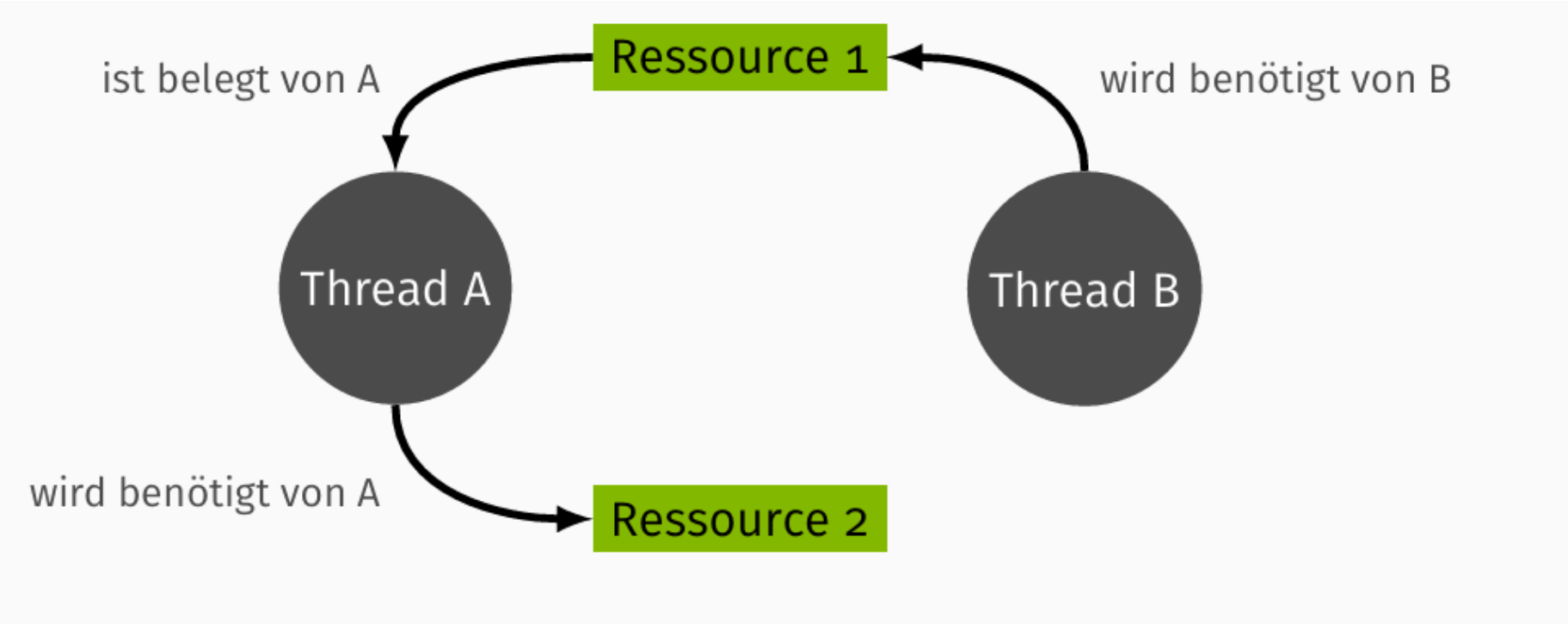
Vorraussetzungen für eine Verklemmung (2)

4. **circular wait:** eine geschlossene Kette wechselseitig wartender Fäden



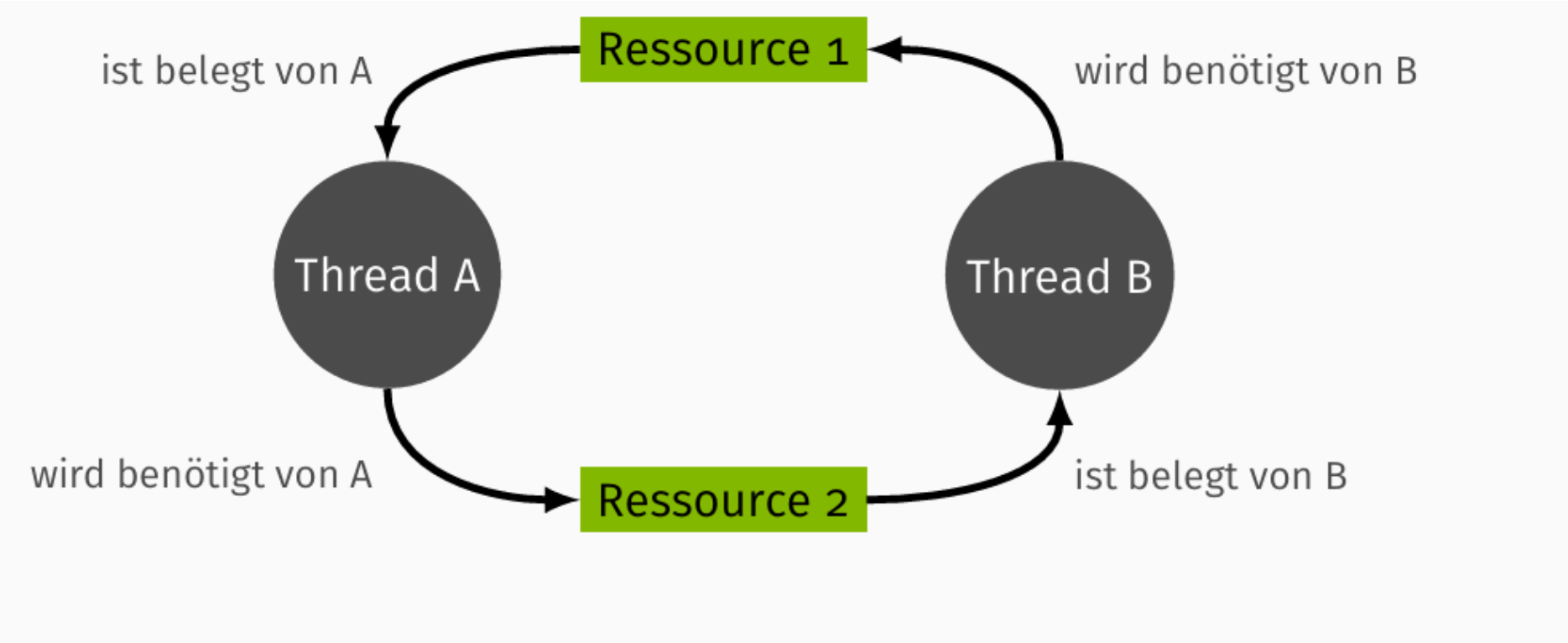
Vorraussetzungen für eine Verklemmung (2)

4. **circular wait:** eine geschlossene Kette wechselseitig wartender Fäden



Vorraussetzungen für eine Verklemmung (2)

4. **circular wait:** eine geschlossene Kette wechselseitig wartender Fäden



Verklemmungsauflösung

- **Prozess abbrechen** und so Betriebsmittel frei bekommen
  - Verklemmte Prozesse schrittweise abbrechen (großer Aufwand)
  - Alle verklemmten Prozesse terminieren (großer Schaden)
- **Betriebsmittel entziehen** und mit dem effektivsten Opfer (?) beginnen
  - Betreffenden Prozess zurückfahren bzw. wieder aufsetzen (Transaktionen, checkpointing/recovery [großer Aufwand])
  - Ein Aushungern der zurückgefahrenen Prozesse ist zu vermeiden
  - Außerdem Vorsicht vor Livelocks!
- **Gratwanderung** zwischen Schaden und Aufwand

Makefiles

- Enthält Informationen, welche Dateien zu einem Projekt gehören und deren Abhängigkeiten
- Bauen von Projekten mit mehreren Dateien
- Makefiles ausführen mit **make**
  - bei fehlendem Target wird das Standard-Target ausgeführt (erstes Target in der Datei)
  - Ein Target entspricht (meistens) der zu erstellenden Datei
  - Optionen: -f gibt das Makefile an, -j Anzahl gleichzeitger Jobs
- Syntax

```
# Name: <benötigte Dateien und/oder andere Targets>
# <TAB> Kommando
```

Beispiel für ein Makefile

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

all: programm1 programm2    # erstes Tagret = Default-Target
programm1: prog1.c prog1.h
        $(CC) $(CFLAGS) -o prog1 prog1.c
programm2: prog2.c prog2.h prog1 # Abhängigkeit: benötigt programm1!
        $(CC) $(CFLAGS) -o prog2 prog2.c
```

```
01 studi@bsvm:~$ make program1
02 gcc -Wall -ansi -pedantic -D_XOPEN_SOURCE -D_POSIX_SOURCE -o prog1 prog1.c
03 studi@bsvm:~$ make
04 gcc -Wall -ansi -pedantic -D_XOPEN_SOURCE -D_POSIX_SOURCE  -o prog1 prog1.c
05 gcc -Wall -ansi -pedantic -D_XOPEN_SOURCE -D_POSIX_SOURCE  -o prog2 prog2.c
```

Klausuraufgabe zur Synchronisation

Situation:

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)).

Klausuraufgabe zur Synchronisation

Zieltext: “**to get to the other side**”. 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 = [ ]

```
01 chicken1() {
02   printf("to ");
03
04
05   printf("to ");
06
07
08   printf("other ");
09
10 }
```

```
12 chicken2() {
13
14   printf("get ");
15
16 }
17 chicken3() {
18
19   printf("the ");
20
21
22   printf("side");
23 }
```

Klausuraufgabe zur Synchronisation

Zieltext: “**to get to the other side**”. 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

```
01 chicken1() {
02   printf("to ");
03
04
05   printf("to ");
06
07
08   printf("other ");
09
10 }
```

```
12 chicken2() {
13
14   printf("get ");
15
16 }
17 chicken3() {
18
19   printf("the ");
20
21
22   printf("side");
23 }
```

Klausuraufgabe zur Synchronisation

Zieltext: “**to get to the other side**”. 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

```
01 chicken1() {
02   printf("to ");
03   v(s2);
04
05   printf("to ");
06
07
08   printf("other ");
09
10 }
```

```
12 chicken2() {
13   p(s2);
14   printf("get ");
15
16 }
17 chicken3() {
18
19   printf("the ");
20
21
22   printf("side");
23 }
```

Klausuraufgabe zur Synchronisation

Zieltext: “**to get to the other side**”. 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

```
01 chicken1() {
02    printf("to ");
03    v(s2);
04    p(s1);
05    printf("to ");
06
07
08    printf("other ");
09
10 }
```

```
12 chicken2() {
13    p(s2);
14    printf("get ");
15    v(s1);
16 }
17 chicken3() {
18
19    printf("the ");
20
21
22    printf("side");
23 }
```

Klausuraufgabe zur Synchronisation

Zieltext: “**to get to the other side**”. 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

```
01 chicken1() {
02    printf("to ");
03    v(s2);
04    p(s1);
05    printf("to ");
06    v(s3);
07
08    printf("other ");
09
10 }
```

```
12 chicken2() {
13    p(s2);
14    printf("get ");
15    v(s1);
16 }
17 chicken3() {
18    p(s3);
19    printf("the ");
20
21
22    printf("side");
23 }
```

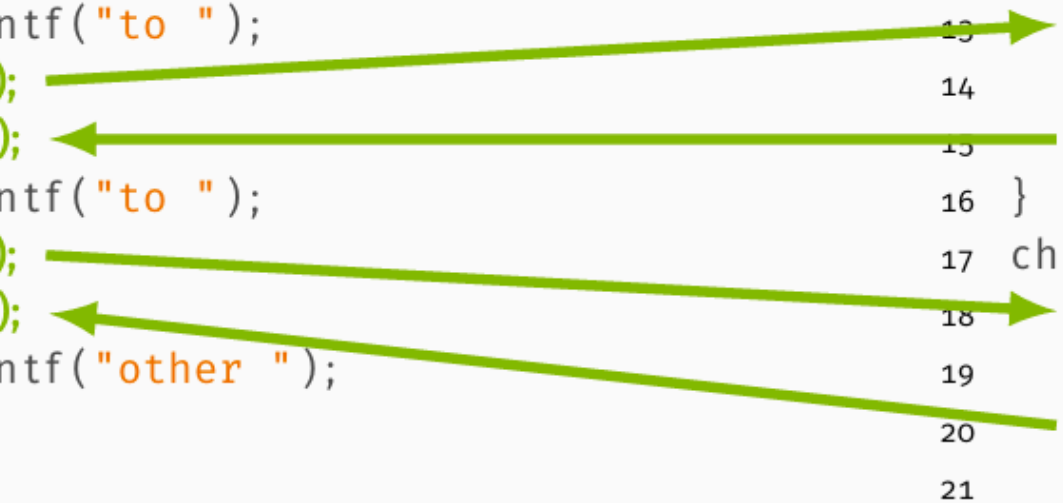
Klausuraufgabe zur Synchronisation

Zieltext: “**to get to the other side**”. 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

```
01 chicken1() {
02    printf("to ");
03    v(s2);
04    p(s1);
05    printf("to ");
06    v(s3);
07    p(s1);
08    printf("other ");
09
10 }
```

```
12 chicken2() {
13    p(s2);
14    printf("get ");
15    v(s1);
16 }
17 chicken3() {
18    p(s3);
19    printf("the ");
20    v(s1);
21
22    printf("side");
23 }
```



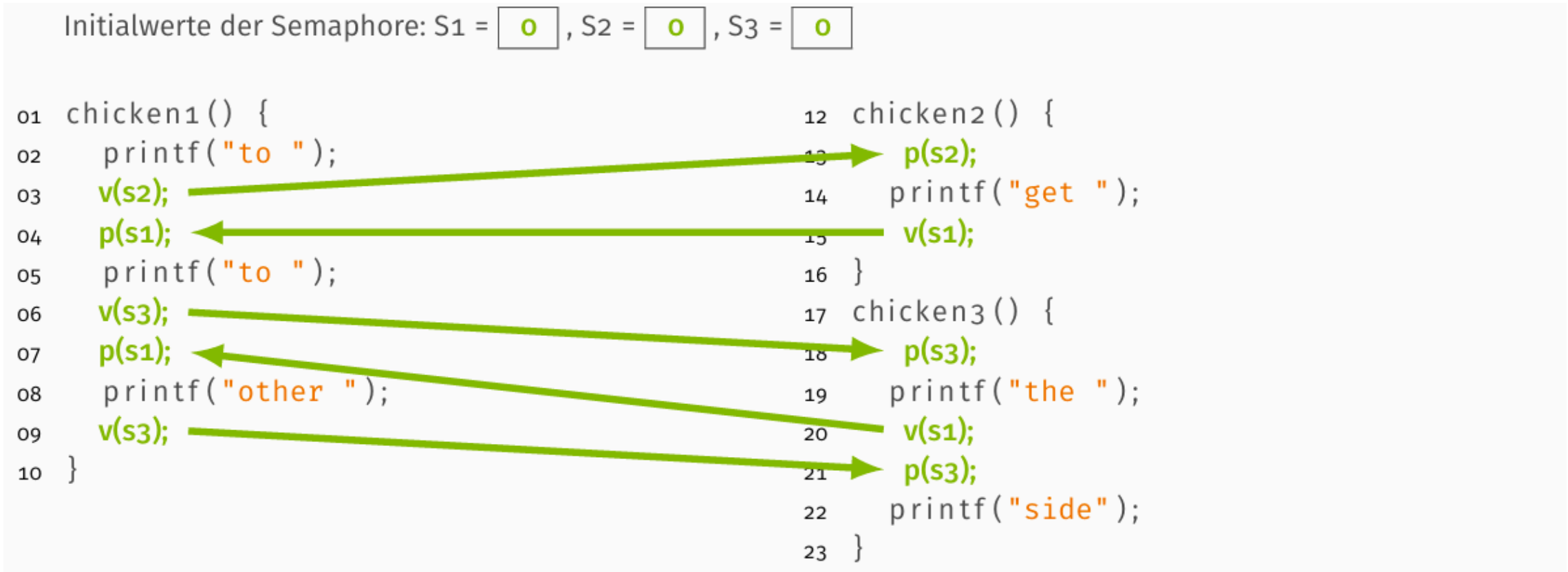
Klausuraufgabe zur Synchronisation

Zieltext: **to get to the other side**. 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

```
01 chicken1() {
02    printf("to ");
03    v(s2);
04    p(s1);
05    printf("to ");
06    v(s3);
07    p(s1);
08    printf("other ");
09    v(s3);
10 }
```

```
12 chicken2() {
13    p(s2);
14    printf("get ");
15    v(s1);
16 }
17 chicken3() {
18    p(s3);
19    printf("the ");
20    v(s1);
21    p(s3);
22    printf("side");
23 }
```



Was haben wir heute gelernt?

- Zusammenfassung
  - Semaphore und deren Synchronisationsmuster
  - Zustandekommen von Verklemmungen
  - Verwendung von Makefiles