Lehrstuhl für Connected Mobility
School of Computation, Information and Technology
Technische Universität München

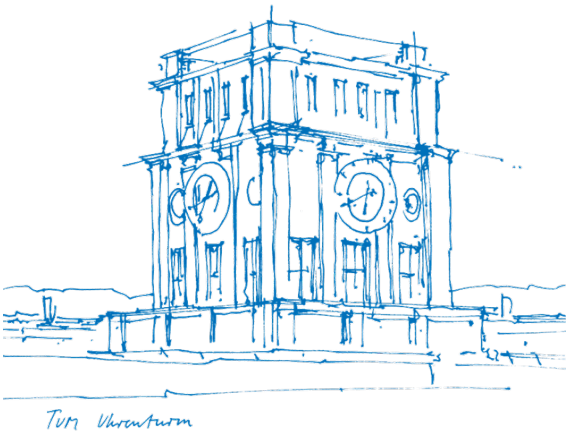
Grundlagen: Betriebssysteme und Systemsoftware

Tutorübung

Mario Delic

Lehrstuhl für Connected Mobility
School of Computation, Information and Technology
Technische Universität München

Übungswoche 5

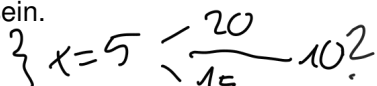


Synchronisation
Begrifflichkeiten

Nebenläufigkeit
Programmteile können unabhängig voneinander in beliebiger Reihenfolge oder gleichzeitig ausgeführt werden

Echte Parallelität
Echt gleichzeitige Ausführung z.B. Multi-Prozessor Systeme

Problem: **Nichtdeterminismus:**
Das Verhalten des Programms kann bei gleichen Bedingungen/Eingaben dennoch unterschiedlich sein.
P1: x = x + 5;
P2: x = x * 2;
↪ Ergebnis nicht eindeutig.



Synchronisation
Begrifflichkeiten

Verklemmung (Deadlock)

1) In einer Menge an Prozesses warten alle Prozesse auf bestimmte Ereignisse
2) Diese Ereignisse können aber nur Prozessen aus derselben Menge auslösen
3) Alle Prozesse warten ohne Ausweg

Verhungern

Prozesse, die nicht im Deadlock sind, aber deren Ausführung (aus anderen Gründen) unendlich lange hinausgezögert/vermieden wird

$\hookrightarrow$ Ein System sollte verklemmungsfrei (keine Deadlocks) und fair (kein Vehungern) sein.

Synchronisation
Begrifflichkeiten

Livelock

Prozesse sind nicht im Deadlock, aber machen trotzdem keinen Fortschritt (’klemmen trotzdem’)

1) Zwei Prozesse A und B locken jeweils die Ressource R_a und R_b
2) Zur weiteren Ausführung braucht A aber zusätzlich R_b, und B R_a
3) A und B sehen, dass sie R_b und R_a aber nicht bekommen können, da belegt
4) A und B geben ihre zuvor belegten Ressourcen R_a und R_b frei
5) Der Vorgang wiederholt sich (unendlich)

Synchronisation
Begrifflichkeiten

Warten

Aktives Warten (z. B. Spinlock):
- In einer (Endlos)schleife wird dauernd auf die Verfügbarkeit der Ressource geprüft

Passives Warten:
- Ein Prozess legt sich (z.B mittels sleep()) schlafen
- Bei Freigabe der nötigen Ressource wird der Prozess wieder geweckt

Synchronisation
Umsetzung

Semaphor
- Zählvariable
- Unterstützt Operationen up und down
- down kann nicht ausgeführt werden wenn Semaphor==0

Mutex
- binäres Semaphor
- Unterstützt Operationen up/unlock und down/lock
- lock kann nicht ausgeführt werden wenn Mutex==locked
- (unlock auf ein unlocked Mutex ist i.d.R. undefined behavior)

Aufgabe 2
Netzwerkkarte

Eine Datei soll über ein Netzwerk auf einen Computer transferiert werden. Die Netzwerkkarte `N` des Computers empfängt blockweise Datenpakete und legt diese im Buffer `B` (Kapazität: `n`) ab, von wo aus sie nach und nach entnommen und auf die Festplatte `F` gespeichert werden. Es sei folgender Lösungsversuch mit dem Mutex `wa` als Pseudocode gegeben:

a) Laufen beide Prozesse verklemmungsfrei? Welche Situationen führen zu Verklemmungen?

b) Geben Sie eine verbesserte Version an, in der keine Probleme mehr auftreten, indem Sie zwei Semaphore geeignet deklarieren und geeignete Aufrufe von `down` und `up` einfügen.

c) Welche Probleme treten auf, wenn Sie in Ihrer verbesserten Lösung die Reihenfolge der `down`-Operationen für `wa` und Ihrer beiden zusätzlichen Semaphore vertauschen?

Aufgabe 2

Netzwerkkarte

a) endloses warten im kritischen Bereich/nach sperren des Mutex → DEADLOCK

```
1    Deklaration:
2    wa(1);
3
4    Netzwerkkarte N:
5    while(true) {
6      <empfange Datenblock>;
7      down(wa);
8      <schreibe Datenblock in B, falls Platz frei, sonst warte>;
9      up(wa);
10   }
11
12   Festplatte F:
13   while(true) {
14     down(wa);
15     <entnimm Datenblock aus B, falls vorhanden, sonst warte>;
16     up(wa);
17     <schreibe Datenblock auf Festplatte>;
18   }
```

Aufgabe 2

Netzwerkkarte

c) Nach vertauschen der down-Operationen kann es wieder zu einem Deadlock kommen. z.B. wenn Buffer voll, dann betritt N den kritischen Bereich (down(wa)), aber blockiert anschließend bei down(frei)

b)

```
Deklaration:
wa(1);
belegt(0);
frei(n);
```

```
Netzwerkkarte N:
while(true) {
  <empfange Datenblock>;
  down(frei);
  down(wa);
  <schreibe Datenblock in B>;
  up(wa);
  up(belegt);
}
```

```
Festplatte F:
while(true) {
  down(belegt);
  down(wa);
  <entnimm Datenblock aus B, falls vorhanden, sonst warte>;
  up(wa);
  up(frei);
  <schreibe Datenblock auf Festplatte>;
}
```

2 Semaphore sind nötig, da ein Semaphor nur eine Eigenschaft gleichzeitig steuern kann (hier: entweder die Leere oder die Völle). Das liegt daran, dass ein Semaphor lediglich nach unten beschränkt ist und nur bei „0" blockiert (nur down blockiert). Nach oben (up) gibt es keine Grenze. Diese Grenze muss durch ein 2. Semaphor realisiert werden.

Aufgabe 3

Wir betrachten die Strecke zwischen Garching-Forschungszentrum (**GF**) und Fröttmaning (**F**). Da zur Zeit gebaut wird, herrscht zwischen Garching-Hochbrück (**GH**) und F eingleisiger Betrieb. Im Folgenden modellieren wir die Synchronisation der Strecke GF $\Longleftrightarrow$ F. Gegeben ist: im Bahnhof GF haben nur zwei Züge Platz, die Kapazität des Bahnhofs F ist unbegrenzt.

a) Fügen Sie einen Mutex hinzu, sodass es auf dem eingleisigen Abschnitt zu **keiner Kollision** kommen kann. Ist aktuell ein Zug im eingleisigen Abschnitt, so muss der nächste im letzten Bahnhof vor der Baustelle warten.

b) Führen Sie mittels Semaphoren Zähler ein, die dafür sorgen, dass in den Bahnhöfen GF und F **jeweils niemals weniger als null** Züge sind. Sorgen Sie dafür, dass in GF **niemals mehr als zwei** Züge sind. Sind in GF bereits zwei Züge, so darf in F kein weiterer Richtung GF ausfahren. **Am Anfang seien in GF ein Zug, in F drei**.

c) Verhindern Sie, dass auf dem Streckenabschnitt GF $\Longleftrightarrow$ GH in beiden Richtungen zusammen mehr als **zwei** Züge unterwegs sind.

Benennungsschema aus der Übung

```
// Prozess
Fahre_in_richtung_F
{
  down (GF belegt)
  down (GFGH Frei)
  <Fahre aus GF aus>
  up (GF leer)

  <Fahre in GH ein>
  up (GFGH Frei)
  down (ein)
  <Fahre aus GH aus>

  <Fahre durch eingleisigen Abschnitt>

  <Fahre in F ein>
  up (ein)
  up (F belegt)
}
```

```
// Prozess
Fahre_in_richtung_GF
{
  down (GF leer)
  down (F belegt)
  down (ein)
  <Fahre aus F aus>

  <Fahre durch eingleisigen Abschnitt>

  <Fahre in GH ein>
  up (ein)
  down (GFGH Frei)
  <Fahre aus GH aus>

  <Fahre in GF ein>
  up (GF belegt)
  up (GFGH Frei)
}
```

Deklarationen:

ein (1)	a)
F belegt (3)	b)
GF belegt (1)	b)
GF leer (1)	b)
GFGH Frei (2)	c)

down/up in natürlicher Sprache:

down (X_voll) ≙ X wird weniger voll!
up (X_voll) ≙ X wird mehr voll!
down (X_frei) ≙ X wird weniger frei!
up (X_frei) ≙ X wird mehr frei!

Alternatives Benennungsschema

```
// Prozess
Fahre_in_richtung_F
{
    down (GFbesetzte Plätze)
    down (GFGHfreie Gleise)
<Fahre aus GF aus>
    up (GFfreie Plätze)

<Fahre in GH ein>
    up (GFGHfreie Gleise)
    down (eingleisig FreiesGleis)
<Fahre aus GH aus>

<Fahre durch eingleisigen Abschnitt>

<Fahre in F ein>
    up (eingleisig FreiesGleis)
    up (FbesetztePlätze)
}
```

```
// Prozess
Fahre_in_richtung_GF
{
    down (GFfreie Plätze)
    down (FbesetztePlätze)
    down (eingleisig FreiesGleis)
<Fahre aus F aus>

<Fahre durch eingleisigen Abschnitt>

<Fahre in GH ein>
    up (eingleisig FreiesGleis)
    down (GFGHfreie Gleise)
<Fahre aus GH aus>

<Fahre in GF ein>
    up (GFbesetzte Plätze)
    up (GFGHfreie Gleise)
}
```

Deklarationen:

eingleisigFreiesGleis(1) a)

FbesetztePlätze(3) b)

GFbesetztePlätze(1) b)

GFfreiePlätze(1) b)

GFGHfreie Gleise(2) c)

down/up in natürlicher Sprache:

down(X_voll) ≙ X wird weniger voll!

up(X_voll) ≙ X wird mehr voll!

down(X_frei) ≙ X wird weniger frei!

up(X_frei) ≙ X wird mehr frei!