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

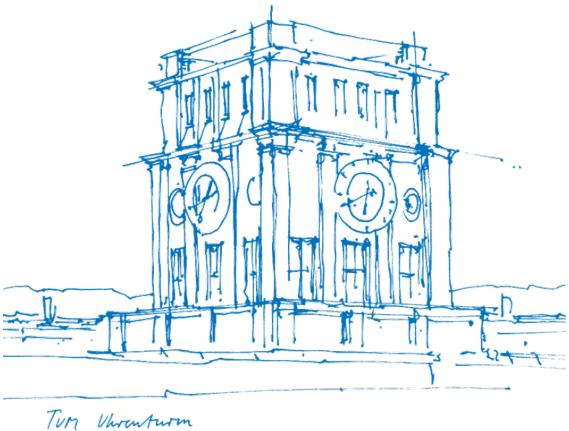
Grundlagen: Betriebssysteme und Systemsoftware

Tutorübung

Mario Delic

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

Übungswoche 5



Synchronisation
Begrifflichkeiten

Determinismus
Ein Programm liefert für die gleichen Eingaben immer die gleichen Ausgaben

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

Mögl. Lösung: **Mutual Exclusion**
Nur ein Prozess kann zu einer Zeit auf die Variable zugreifen

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

4 Deadlock-Bedingungen

- Mutual exclusion: Die Ressourcen kann nur 1 Prozess gleichzeitig halten
- Hold-and-wait: Das besetzen von Ressourcen, auch wenn man mit Ihnen allein nicht weiterkommt und erst auf andere warten muss
- No preemption: Beim halten der Ressource kann sie dem Prozess nicht entzogen werden
- Circular wait: Die Prozesse warten in einem Kreis aufeinander, sodass keiner weiterkommt

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

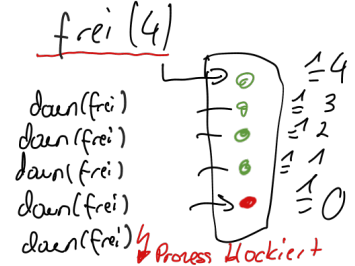
Verhungern
Prozesse, die nicht im Deadlock sind, aber deren Ausführung (aus anderen Gründen) unendlich lange hinausgezögert/vermieden wird
↪ Ein System sollte verklemmungsfrei (keine Deadlocks) und fair (kein Vehungern) sein.

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

("Zeichenträger"; in manchen Sprachen literally Ampel)

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

Um sich nicht zu verwirren: das Semaphor so benennen, dass bei $x \geq 1$ „ja" gilt, und bei $x = 0$ „nein" gilt.

Mutex

- Unterstützt Operationen unlock (≈up) und lock (≈down)
- lock kann nicht ausgeführt werden wenn Mutex==locked
- unterschied zu binärem Semaphor: Mutex wird acquired, d.h. nur der lockende kann unlocken

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

Szenario: B ist voll:

```
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  }
```

#2 hold-and-wait

N blockiert

#3: kein Mechanismus zum Ressourcenentzug vorhanden

#1: Mutually exclusive

#4: warten auf F, damit sie B leert

#4: circular wait

F blockiert → #4: warten auf up von N

Analog dazu Deadlock wenn B leer und F kritischen Bereich betritt (down(wa)).

Aufgabe 2
Netzwerkkarte

```
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);
}
```

sonst warte

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

Kritischen Bereich immer „am engsten" Synchronisieren (hier: wa).

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.

```
// Prozess
Fahre_in_richtung_F
{
  down ( Züge GF )
  down ( Frei GFGH)
<Fahre aus GF aus>
  up ( Freie Gleise GF )

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



<Fahre durch eingleisigen Abschnitt>



<Fahre in F ein>
  up ( Bvoll )
  up ( Züge F )
}
```

```
// Prozess
Fahre_in_richtung_GF
{ down (Freie Gleise GF)
  down ( Züge F )
  down (Bvoll)
<Fahre aus F aus>



<Fahre durch eingleisigen Abschnitt>



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

  up ( Züge GF)
<Fahre in GF ein>
  up( Frei GFGH)

}
```

Bavoll (1)

Züge F (3)

Züge GF (1)

Freie Gleise GF (1)

Frei GFGH (2)