Tutorübung Grundlagen: Betriebssysteme und Systemsoftware

Moritz Beckel

München, 25. November 2022

Freitag 10:15-12:00 Uhr Raum (00.11.038)

Zulip-Stream https://zulip.in.tum.de/#narrow/stream/1295-GBS-Fr-1000-A

Unterrichtsmaterialien findet ihr hier:

https://home.in.tum.de/~beckel/gbs

Lösungen wurden von mir selbst erstellt. Es besteht keine Garantie auf Korrektheit.

Zusammenfassung

1. Parallele Systeme
2. Prozesssynchronisation
3. Verklemmungen (Deadlocks, …)

Zusammenfassung

1. Parallele Systeme

- Nebenläufige (deterministisch) vs. echt parallele Ausführung (nicht deterministisch)
- Kausale Beziehungen: Abhängigkeiten zwischen nebenläufig ausgeführten Aktivitäten
- Kommunikation: Austausch zwischen 2 Prozessen
- Koordinierung: Beziehung Auftragnehmer, Auftraggeber
- Konkurrenz: mehrere Prozesse greifen gleichzeitig auf begrenzte Ressourcen zu

Zusammenfassung

1. Gewünschte Eigenschaften paralleler Systeme

- Determiniertheit: gleiche Eingaben == gleiches Ergebnis
- Störungsfreiheit: Ergebnis wird nicht beeinflusst, sofern festgelegte Ausführungsreihenfolge eingehalten
- Wechselseitiger Ausschluss: maximal ein Prozess greift auf gemeinsame Ressource zu
- Verklemmungsfreiheit: Verhindern, dass mehrere Prozesse nicht mehr weiterrechnen können
- Kein Verhungern: Prozess darf nicht rechnen bspw. unfaires Scheduling

Zusammenfassung

1. Synchronisation von parallelen Systemen

- Race Condition: Ergebnis ist von der Reihenfolge der Prozessausführung abhängig
- Kritischer Abschnitt: Abschnitt in den Race Conditions auftreten können
- Sequentialisierung

Zusammenfassung

1. Anforderung für eine Synchronisation

- Kritische Abschnitte sind wechselseitig Ausgeschlossen
- Keine Annahmen über die Reihenfolge der Zugriffe
- Keine Annahmen über die Ausführungszeit
- Kein Prozess darf unendlich lange im kritischen Abschnitt verweilen

Zusammenfassung

2. Prozesssynchronisation

Hardware-Ebene:

- Unterbrechungssperre
- Test-and-Set-Lock (TSL)

Betriebssystem-Ebene:

- Aktives Warten (spinlock)
- Passives Warten (thread_yield, sleep)

Höhere Abstraktion:

- Semaphoren (Kontrollvariable)
- Mutexe (Binäre Semaphore)

Programmiersprachen-Ebene:

- Monitor-Konzept

Zusammenfassung

2. Producer-Consumer-Problem

```
void producer (void) {
    int item = 0;

    while (true) {
        item = produce_item ();
        if (count == N)
            sleep ();
        insert_item (item);
        count = count + 1;
        if (count == 1)
           wakeup (consumer);
    }
}
```

```
void consumer (void) {
    int item = 0;

    while (true) {
        if (count == 0)
            sleep ();
        item = remove_item ();
        count = count - 1;
        if (count == N-1)
            wakeup (producer);
        consume_item (item);
    }
}
```

Zusammenfassung

2. Producer-Consumer-Problem

```
/* Producer */
while (true) {
    element = produce();
    down (&leer);
    down (&wa);
    write_to_buf (W, element);
    // Would wait if full…
    up (&wa);
    up (&voll);
}
```

```
/* Consumer */
while (true) {
    down (&voll);
    down (&wa);
    element = read_from_buf (W);
    // Would wait if empty…
    up (&wa);
    up (&leer);
    consume (element);
}
```

Zusammenfassung

3. Verklemmungen (Deadlocks, …)

- Livelock: zwei Prozesse besitzen jeweils eine Ressource und wollen auf die des jeweils anderen zugreifen; wird bemerkt und beide geben Ressourcen frei. Sie nehmen wieder eine der beiden Ressourcen und wiederholen sich.
- Starvation: Prozess erhält Ressourcen vom Betriebssystem nicht, etwa durch unfaire Verteil-Strategie der Ressourcen

Zusammenfassung

3. Verklemmungen (Deadlocks, …)

Deadlocks: Zustand in dem das System verklemmt, jeder Prozess wartet auf ein Ergebnis eines anderen Prozesses

Bedingungen:

- Mutual exclusion: Exklusiv nutzbare Ressourcen
- Hold-and-wait: Prozesse geben Ressourcen nicht frei, während sie auf weitere Ressourcen warten
- No-preemption: Ressourcen können nicht entzogen werden
- Circular-Wait: Es kann eine zyklische Wartekette von Prozessen geben, in der die Prozesse jeweils auf die, vom nachfolgenden Prozess, belegten Ressourcen warten.

Zusammenfassung

Ansatz	Verfahren	Vorteile	Nachteile
Ignorieren	-	Schnell	Systemabstürze
Detection	Periodischer Aufruf	Interaktive Reaktionen	Verlust durch Abbruch
Prevention	Feste Reihenfolge bei Zuteilung; Alle Ressourcen auf einmal zuteilen	Keine Laufzeitprüfungen, Kein Ressourcenentzug notwendig	Statisch, inflexibel, ineffizient
Avoidance	Bankier-Algorithmus	Kein Ressourcenentzug	Zukünftiger Bedarf muss bekannt sein

Aufgabe 2

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. Um wechselseitigen Ausschluss zu erreichen, sei folgender Lösungsversuch mit dem Mutex wa als Pseudocode gegeben:

Aufgabe 2

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

```
1   Deklaration:
2   wa(1);
3
4   Netzwerkkarte N:
5   while(true) {
6     <empfange Datenblock>;
7     down(wa);
8     <schreibe Datenblock in B>;
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

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

- Verklemmung bei leerem Buffer (F beansprucht wa und wartet auf nächsten Datenblock, N wartet auf wa)
- Verklemmung bei vollem Buffer

```
1   Deklaration:
2   wa(1);
3
4   Netzwerkkarte N:
5   while(true) {
6     <empfange Datenblock>;
7     down(wa);
8     <schreibe Datenblock in B>;
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

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.

```
1   Deklaration:
2   wa(1);
3
4   Netzwerkkarte N:
5   while(true) {
6     <empfange Datenblock>;
7     down(wa);
8     <schreibe Datenblock in B>;
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

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.

```
1   Deklaration:          voll(0);
2   wa(1);                leer(n);
3                         ←
4   Netzwerkkarte N:
5   while(true) {                          down(leer);
6     <empfange Datenblock>;               ←
7     down(wa);
8     <schreibe Datenblock in B>;
9     up(wa); up(voll);
10  }         ←
11
12  Festplatte F:
13  while(true) {  down(voll);
14    down(wa);    ←
15    <entnimm Datenblock aus B, falls vorhanden, sonst warte>;
16    up(wa);
17    <schreibe Datenblock auf Festplatte>;  ←
18  }                                        up(leer);
```

Aufgabe 2

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

```
Deklaration:
wa(1);
voll(0);
leer(n);

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

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

Aufgabe 2

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

- Ein Deadlock wird möglich. (Buffer voll, Verbraucher blockiert innerhalb des kritischen Abschnitts, Erzeuger führt down(wa) aus und blockiert auch)

```
Deklaration:
wa(1);
voll(0);
leer(n);

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

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

Aufgabe 3

Wir betrachten die Strecke der U6 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 des Streckenabschnitts GF $\Leftarrow\Rightarrow$ F. Gegeben ist: im Bahnhof GF haben nur zwei Züge Platz, die Kapazität des Bahnhofs F ist unbegrenzt.

Aufgabe 3 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.

```
// Prozess
Fahre_in_richtung_F
{

  <Fahre aus GF aus>

  <Fahre in GH ein>

  <Fahre aus GH aus>

  <Fahre durch eingleisigen Abschnitt>

  <Fahre in F ein>

}
```

```
// Prozess
Fahre_in_richtung_GF
{

  <Fahre aus F aus>

  <Fahre durch eingleisigen Abschnitt>

  <Fahre in GH ein>

  <Fahre aus GH aus>

  <Fahre in GF ein>

}
```

Aufgabe 3 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.

Deklarationen:

Baustelle(1);

```
// Prozess
Fahre_in_richtung_F
{

  <Fahre aus GF aus>

  <Fahre in GH ein>

   down(Baustelle);
  <Fahre aus GH aus>

  <Fahre durch eingleisigen Abschnitt>

  <Fahre in F ein>
   up(Baustelle);

}
```

```
// Prozess
Fahre_in_richtung_GF
{

  down(Baustelle);
 <Fahre aus F aus>

 <Fahre durch eingleisigen Abschnitt>

 <Fahre in GH ein>
 up(Baustelle);

 <Fahre aus GH aus>

 <Fahre in GF ein>

}
```

Aufgabe 3 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.

Deklarationen:
Baustelle(1);

```
// Prozess
Fahre_in_richtung_F
{

  <Fahre aus GF aus>

  <Fahre in GH ein>

   down(Baustelle);
  <Fahre aus GH aus>

  <Fahre durch eingleisigen Abschnitt>

  <Fahre in F ein>
   up(Baustelle);

}
```

```
// Prozess
Fahre_in_richtung_GF
{

   down(Baustelle);
  <Fahre aus F aus>

  <Fahre durch eingleisigen Abschnitt>

  <Fahre in GH ein>
   up(Baustelle);

  <Fahre aus GH aus>

  <Fahre in GF ein>

}
```

Aufgabe 3 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.

Deklarationen:
Baustelle(1);
GF_voll(1);
GF_frei(1);
F_voll(3);

```
// Prozess
Fahre_in_richtung_F
{

  down(GF_voll);
 <Fahre aus GF aus>
  up(GF_frei);

 <Fahre in GH ein>

  down(Baustelle);
 <Fahre aus GH aus>

 <Fahre durch eingleisigen Abschnitt>

 <Fahre in F ein>
  up(Baustelle);
  up(F_voll);
}
```

```
// Prozess
Fahre_in_richtung_GF
{
  down(GF_frei);
  down(F_voll);
  down(Baustelle);
 <Fahre aus F aus>

 <Fahre durch eingleisigen Abschnitt>

 <Fahre in GH ein>
  up(Baustelle);

 <Fahre aus GH aus>

 <Fahre in GF ein>
  up(GF_voll);
}
```

Aufgabe 3 c)

Verhindern Sie, dass auf dem Streckenabschnitt GF ⇐⇒ GH in beiden Richtungen zusammen mehr als zwei Züge unterwegs sind.

Deklarationen:
Baustelle(1);
GF_voll(1);
GF_frei(1);
F_voll(3);

```
// Prozess
Fahre_in_richtung_F
{

  down(GF_voll);
 <Fahre aus GF aus>
  up(GF_frei);

 <Fahre in GH ein>

  down(Baustelle);
 <Fahre aus GH aus>

 <Fahre durch eingleisigen Abschnitt>

 <Fahre in F ein>
  up(Baustelle);
  up(F_voll);
}
```

```
// Prozess
Fahre_in_richtung_GF
{
  down(GF_frei);
  down(F_voll);
  down(Baustelle);
 <Fahre aus F aus>

 <Fahre durch eingleisigen Abschnitt>

 <Fahre in GH ein>
  up(Baustelle);

 <Fahre aus GH aus>

 <Fahre in GF ein>
  up(GF_voll);

}
```

Aufgabe 3 c)

Verhindern Sie, dass auf dem Streckenabschnitt GF ⇐⇒ GH in beiden Richtungen zusammen mehr als zwei Züge unterwegs sind.

Deklarationen:
Baustelle(1);
GF_voll(1);
GF_frei(1);
F_voll(3);
Abschnitt_GF_GH(2);

```
// Prozess
Fahre_in_richtung_F
{

   down(GF_voll);
   down(Abschnitt_GF_GH);
  <Fahre aus GF aus>

   up(GF_frei);


  <Fahre in GH ein>

   up(Abschnitt_GF_GH);

   down(Baustelle);
  <Fahre aus GH aus>




  <Fahre durch eingleisigen Abschnitt>




  <Fahre in F ein>
   up(Baustelle);
   up(F_voll);

}
```

```
// Prozess
Fahre_in_richtung_GF
{
   down(GF_frei);
   down(F_voll);
   down(Baustelle);
  <Fahre aus F aus>




  <Fahre durch eingleisigen Abschnitt>




  <Fahre in GH ein>
   up(Baustelle);
   down(Abschnitt_GF_GH);

  <Fahre aus GH aus>




  <Fahre in GF ein>
   up(GF_voll);
   up(Abschnitt_GF_GH);

}
```