Grundlagen der praktischen Informatik
Dr. Stefan Siemer
M.Sc. Timo Specht

Sommersemester 2025
Georg-August-Universität Göttingen
Institut für Informatik
Lehrstuhl für Theoretische Informatik

Übungsblatt 3 (100 Punkte)

Prozess-Synchronisation & Scheduling

Abgabe bis Di, 13. Mai, 14 Uhr.

Für das erfolgreiche absolvieren der Übungen, müssen Sie sich einmalig in FlexNow zum Übungsmodul **B.Inf.1102.Ue: Grundlagen der Praktischen Informatik - Übung** anmelden. Melden Sie sich **rechtzeitig** in **FlexNow** an.

Rechnerübung: Hilfe zu den praktischen Übungen können Sie in den Rechnerübungen bekommen. Details dazu werden noch bekannt gegeben.

Für die Prüfungszulassung benötigen Sie auf 8 verschiedenen Übungsblättern im theoretischen und praktischen Teil jeweils 40%, also jeweils 20 Punkte.

Theoretischer Teil

Hinweise zur Abgabe der Lösungen:

Die Lösungen der Aufgaben werden in geeigneter Form in der Stud.IP-Veranstaltung der Vorlesung über das Vips-Modul hochgeladen. Sie können Ihre Bearbeitungen gerne mit LaTeX formatieren, es ist aber auch der Upload von Text und Bilddateien in gängigen Formaten möglich.

Sollten Sie LaTeX nutzen wollen, gibt es hierfür Hinweise in Form eines Videos in Stud.IP und Sie können unser LaTeX-Template für Ihre Abgaben nutzen.

Die Aufgaben müssen in Gruppen von 2 bis 3 Personen bearbeitet und abgegeben werden. Wenn Sie in einer Gruppe abgeben möchten, achten Sie darauf, dass Sie sich vor der Abgabe im Vips-Modul der Vorlesung in einer Gruppe zusammenschließen.

Aufgabe 1 (15 Punkte)

1. Was versteht man bei der Prozess-Synchronisation unter dem Begriff der Race Condition? (5 Punkte)
2. Beschreiben Sie kurz wozu ein Mutex verwendet wird und welche Datenstrukturen ein Mutex verwendet. (5 Punkte)
3. In der Vorlesung wurden die Methoden `up` und `down` für einen Mutex definiert. Beschreiben Sie jeweils, wie diese Methoden funktionieren. (5 Punkte)

Aufgabe 2 (10 Punkte)

Betrachten Sie den Pseudo-Code der Prozesse I (Init), A (Alice), B (Bob).

I

```
1  global int i = 1;
2  global mutex a = false;
3  global mutex b = true;
```

A

```
4  while(i < 4) {
5    down(a)
6    i = i + 1
7    up(b)       }
```

B

```
8  while(i < 4) {
9    down(b)
10   i = i / 2 + 2
11   up(a)       }
```

Hinweis. Die Variablen `i`, `a` und `b` sind global, d.h. in jedem Prozess verfügbar.

Das System verwaltet die Prozesse, die darauf warten Rechnenzeit zu bekommen, nach dem Prinzip *First Come First Served* (FCFS). Dabei kann `down` den Prozess aktiv schlafen legen. Das heißt, wird der Prozess geblockt, ist er nicht mehr rechnend und ein neuer Prozess kann aus der Warteschlange genommen werden. Wird ein Prozess von `down` schlafen gelegt, wird er erstmal der Schlange von dem entsprechenden Mutex hinzugefügt. Erst, wenn der Prozess aufwacht, wird er der FCFS Warteschlange hinzugefügt.

Die Prozesse werden in der Reihenfolge I, A, B gestartet.

Vervollständigen Sie die folgende Tabelle

Zeile	i	Queue A	Queue B	aktiver Prozess
1	1	leer	leer	I

(10 Punkte)

Aufgabe 3 (15 Punkte)

Für diese Aufgabe wird folgendes Szenario betrachtet. Es gibt einen Init Prozess I, der alle globalen Variablen initiiert. Anschließend werden zwei Prozesse A und B gleichzeitig gestartet. Dabei sollen die Prozesse die Summe der ersten 10 natürlichen Zahlen berechnen, indem sie abwechselnd die Zahlen von 1 bis 10 auf eine Summe addieren. Nachdem ein Prozess eine Zahl addiert hat, berechnet er die als nächstes aufzuaddierende Zahl.

Im Folgenden sind zwei fehlerhafte Implementierungen dieses Szenarios dargestellt.

Hinweis. Die Variablen `sum`, `next_add` und `a` sind global, d.h. in jedem Prozess verfügbar.

I

```
1  global int sum = 0;
2  global int next_add = 1;
3  global mutex a = true;
```

1. A

```
4  while(next_add < 11) {
5      sum += next_add;
6      next_add += 1;
7  }
```

B

```
8  while(next_add < 11) {
9      sum += next_add;
10     next_add += 1;
11 }
```

2. A

```
4  while(next_add < 11) {
5      down(a);
6      sum += next_add;
7      up(a);
8      next_add += 1;
9  }
```

B

```
10 while(next_add < 11) {
11     down(a);
12     sum += next_add;
13     up(a);
14     next_add += 1;
15 }
```

1. Was ist der Fehler bei der ersten Implementierung? Beschreiben Sie einen möglichen fehlerhaften Durchlauf. (5 Punkte)
2. Was ist der Fehler bei der zweiten Implementierung? Beschreiben Sie einen möglichen fehlerhaften Durchlauf. (5 Punkte)
3. Geben Sie eine korrigierte Version an, sodass die Anforderungen erfüllt werden. (5 Punkte)

Aufgabe 4 (10 Punkte)

In einem Computerspiel sollen abwechselnd zwei computer-gesteuerte Spieler (P1 und P2) Züge aus einem Liste `actions` nehmen und diese anschließend ausführen. Nachdem ein Zug ausgeführt wurde, wird dieser mit der Methode `eval` ausgewertet und der Spieler schreibt das Ergebnis in eine gemeinsame Liste `results`. Dabei soll `results[i]` die Bewertung des Zuges `actions[i]` beinhalten.

P1 und P2 werden gleichzeitig gestartet. Die Liste `actions` und die Liste `results` sind globale Ressourcen. D.h., sie sind in beiden Prozessen (P1 und P2) vorhanden.

1. Welche Probleme können bei dem oben beschriebenen Szenario auftreten? (5 Punkte)

   Hinweis: Untersuchen Sie, was bei dem Zugriff auf die geteilten Ressourcen schief laufen kann.

2. Wie könnte man diese Probleme lösen? Geben Sie eine Beschreibung oder Pseduocode an. (5 Punkte)

   Hinweis: Sie können annehmen, dass die Methode `eval` implementiert ist und als einziges Argument den zu bewertenden Zug übergeben bekommt.

Praktischer Teil

Hinweise zur Abgabe der Lösungen:

Die Prüfsumme wird in geeigneter Form in der Stud.IP-Veranstaltung der Vorlesung über das Vips-Modul hochgeladen. Jedes Aufgabenblatt hat die Aufgabe *Abgabe-Prüfsumme*. Dort hinterlegen Sie Ihre generierte Prüfsumme. Sie können zudem auch Ihren Quellcode dort hinterlegen.

Die Aufgaben müssen in Gruppen von 2 bis 3 Personen bearbeitet und abgegeben werden. Wenn Sie in einer Gruppe abgeben möchten, achten Sie darauf, dass Sie sich vor der Abgabe im Vips-Modul der Vorlesung in einer Gruppe zusammenschließen. Dabei ist zu beachten, dass die Gruppen für den theoretischen und praktischen Teil identisch sein müssen.

Aufgabe 5 (50 Punkte)

Bilden Sie das Verhalten verschiedener Scheduling-Verfahren nach.

In der Vorlesung wurde der Typ `Prozess` mit Prozessbezeichner `pid`, Ankunftzeit `arrival` und Rechenzeit `computing` eingeführt. Prozesse sind über Ihre Rechenzeiten vergleichbar und können nach Rechenzeiten geordnet werden.

```haskell
data Prozess = Prozess { pid       :: String
                       ,arrival   :: Int
                       ,computing :: Int } deriving (Show)

instance Eq Prozess where
    Prozess { computing = a } == Prozess { computing = b } = a == b

instance Ord Prozess where
    compare x y
        | computing x < computing y   = LT
        | computing x > computing y   = GT
        | otherwise                   = EQ
```

Verwenden Sie zum Signalsieren, dass der Prozessor mit keinem der Prozesse, denen Rechenzeit zugeteil werden soll, belegt ist, den Idle-Prozess.

```haskell
let idle = Prozess{pid = "IDLE", arrival = -1, computing = -1}
```

Verwenden Sie den Typ `State`, mit den noch nicht angekommenen Prozessen `new`, dem rechnenden Prozess `run`, den wartenden Prozessen `ready`, der aktuellen Zeit `time` und einer Repräsentation des Schedules der Prozesse bis zur aktuellen Zeit `chart`, um den fortlaufenden Status des betrachteten Verfahrens zu beschreiben.

```haskell
data State = State { new      :: [Prozess]
                   ,run      :: Prozess
                   ,ready    :: [Prozess]
                   ,time     :: Int
                   ,chart    :: String }
```

1. Die Standarddarstellung von `State`, die man mit `show` bekommen würde, wenn man `deriving (Show)` der Deklaration von `State` hinzufügt, ist sehr unübersichtlich.

   Sorgen Sie selbst für eine übersichtlichere Darstellung mit `show`.

```haskell
instance Show State where
    show s = ...
```

   (5 Punkte)

   Beispiel

   Scheduling-Beispiel aus der Vorlesung.

Prozesse	P_1	P_2	P_3	P_4	P_5	P_6
Ankunftszeit a_i	0	2	4	12	16	19
Rechenzeit t_i	6	6	5	4	3	6

   Umgesetzt in den initialen Status für das Scheduling.

```haskell
let ps = [Prozess{pid="P1", arrival=0, computing=6},...]
let lectureExample = State { new=ps
                           ,run=idle
                           ,ready=[]
                           ,time=0
                           ,chart="" }
```

   Dann könnte eine übersichtlichere Ausgabe wie folgt aussehen.

```haskell
print lectureExample
```

```
-- new
Prozess {pid = "P1", arrival = 0, computing = 6}
Prozess {pid = "P2", arrival = 2, computing = 6}
Prozess {pid = "P3", arrival = 4, computing = 5}
Prozess {pid = "P4", arrival = 12, computing = 4}
Prozess {pid = "P5", arrival = 16, computing = 3}
Prozess {pid = "P6", arrival = 19, computing = 6}
-- run
Prozess {pid = "IDLE", arrival = -1, computing = -1}
-- ready
-- time: 0
-- chart:
```

   Hinweis

   Die Ausgabe ist vor allem für die Fehlersuche hilfreich.

2. Programmieren Sie zum Aktualisieren der wartenden Prozesse die folgende Funktion.

```haskell
update_ready :: State -> State
```

Alle noch nicht angekommene Prozesse, deren Ankunftszeit mit der aktuellen Zeit übereinstimmt, werden in die Liste der wartenden Prozesse verschoben.
(5 Punkte)

Hinweis

Es ist sinnvoll ein Hilfsfunktion zu verwenden, die abhängig von einer übergebenen Zeit eine übergebene Prozessliste in ein Tupel von Prozesslisten aufteilt.

3. Programmieren Sie eine Funktion

```
update_run :: State -> State
```

die, falls aktuell der Idle-Prozess gerechnet wird, den vordersten Prozess in der Liste der wartenden Prozesse zum rechnenden Prozess macht.
(5 Punkte)

4. Programmieren Sie zum Aktualisieren des Status für den nächsten Zeitabschnitt die folgende Funktion.

```
update_time :: State -> State
```

Die aktuelle Zeit wird um Eins erhöht, die Rechenzeit des rechnenden Prozesses um Eins verringert. In der Repräsentation des Schedules der Prozesse bis zur aktuellen Zeit (`chart`) wird die Prozess-ID des rechnenden Prozesses eingetragen. Ist der rechnende Prozess abgearbeitet, d.h. keine Rechenzeit mehr benötigt, wird der Idle-Prozess zum rechnenden Prozess.
(5 Punkte)

5. Erzeugen Sie einen Status für das Beispiel aus Aufgabeteil 1. und wenden Sie die Funktionen `update_ready`, `update_run` und `update_time`, in dieser Reihenfolge, solange auf diesen Status an, bis alle Prozesse abgelaufen sind. Geben Sie die Repräsentation des Schedules der Prozesse für den resultierenden Status aus.
(10 Punkte)

Hinweis

Diese Funktion bildet das Scheduling-Verfahren `First Come First Serve` (FCFS) nach.

6. Passen Sie die Funktion `update_ready` an, sodass die Scheduling-Verfahren *Shortest Job First (SJF)* und *Shortest Remaining Time First (SRTF)* simuliert werden (siehe nachstehend beide Verfahren).

Wenden Sie analog zu Aufgabeteil 5 diese Scheduling-Verfahren auf das Beispiel aus Aufgabeteil 1 an.
(20 Punkte)

Hinweis

Mit der Funktion `sort` können Listen sortiert werden, die Elemente eines Typs enthalten, der eine Instanz von `Ord` ist. Diese Funktion steht nach dem Import des Moduls `Data.List` zu Verfügung (`import Data.List`).

Shortest Job First (SJF)

- Nicht-unterbrechendes Scheduling.
- Es wird jeweils der Prozess mit der kürzesten Rechenzeit als nächstes auf dem Prozessor ausgeführt.

Shortest Remaining Time First (SRTF)

- Unterbrechendes Scheduling.
- Es wird jederzeit der Prozess auf dem Prozessor ausgeführt, der die geringste verbleibende Rechenzeit hat.
- Neu eintreffende Prozesse, die eine kürzere Rechenzeit haben, als der gerade bearbeitete Prozess, lösen den aktuell rechnenden Prozess ab.