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 2 (100 Punkte)

Scheduling

Abgabe bis Di, 6. 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 können in Gruppen von bis zu 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.

Vorbereitung - Rechenintensive Prozesse und Scheduling

Allgemeine Begriffe

- Ein Prozess wird **aktiviert**, wenn er Prozessorzeit zugeteilt bekommt, d.h. der Zustand des Prozesses wechselt von bereit zu rechnend.
- Die **Ankunftszeit** eines Prozesses ist der Zeitpunkt, ab dem der Prozess vom Scheduling berücksichtigt wird. Der Prozess wird erzeugt, ist rechenbereit und wird der Menge der Prozesse mit Zustand bereit hinzugefügt. Ist dieser Prozess der einzige in der Menge der Prozesse mit Zustand bereit, wird der Prozess sofort aktiviert.

  Beispiel. Kommt der Prozess P zum Zeitpunkt t an, ist die Bereitliste leer und kein Prozess belegt den Prozessor, dann wird der Prozess P zum Zeitpunkt t aktiviert.
- Die **Endzeit** ist der Zeitpunkt an dem ein Prozess vollständig abgelaufen ist.
- Die **Wartezeit** eines Prozesses ist die Zeit zwischen Ankunfts- und Endzeitpunkt, in der der Prozess nicht rechnet.

Nicht-unterbrechendes Scheduling

- Ein Prozess läuft nach der Aktivierung vollständig ab ohne zu blockieren.
- Kommt eine neuer Prozess an, wird er in die Menge der bereiten Prozesse eingeordnet.
- Beendet sich der aktuell rechnende Prozesse, muss ein Prozess, aus der Menge der bereiten Prozesse, ausgesucht werden, der aktiviert wird. Ist die Menge leer, läuft der Prozessor (quasi im Leerlauf) weiter, bis wieder ein Prozess in der Menge der bereiten Prozesse verfügbar ist.

Unterbrechendes Scheduling

- Der aktuell rechnende Prozess läuft ohne zu blockieren solange bis er unterbrochen wird oder vollständig abgelaufen ist, d.h. sich beendet.
- Kommt eine neuer Prozess an, wird er in die Menge der bereiten Prozesse eingeordnet. Weiterhin wird der aktuell rechnende Prozess unterbrochen und ebenfalls in die Menge der bereiten Prozesse eingeordnet, dabei wird die noch verbleibende Rechenzeit dem Prozess als Rechenzeit zugeordnet.
- Kommt ein neuer Prozess an oder beendet sich der aktuell rechnende Prozess, muss ein Prozess, aus der Menge der bereiten Prozesse, ausgesucht werden, der aktiviert wird. Ist die Menge leer, läuft der Prozessor (quasi im Leerlauf) weiter, bis wieder ein Prozess in der Menge der bereiten Prozesse verfügbar ist.

Aufgabe 1 (25 Punkte)

Hinweis: Lesen Sie zuerst die Informationen zu Scheduling auf der **Vorseite** durch, bevor Sie die Aufgabe bearbeiten.

Auf der Suche nach einem guten Scheduling-Verfahren für Großrechner, auf den hauptsächlich rechenintensive Prozesse ablaufen, wird folgendes Beispiel betrachtet.

Prozesse	P_1	P_2	P_3	P_4	P_5	P_6
Ankunftszeit	0	13000	14000	39000	41000	47000
Rechenzeit	15000	20000	5000	50000	2000	10000

1. Lassen Sie die Prozesse $P_1, \ldots, P_6$ mit nicht-unterbrechenden und unterbrechenden Scheduling ablaufen. Stellen Sie jeweils, wenn ein Prozess aktiviert wird, den rechnenden Prozess und die Menge der bereiten Prozesse dar.

   Wählen Sie jeweils den zu aktivierenden Prozess, sodass die Summe der Wartezeit für die Prozesse in der Menge der bereiten Prozesse möglichst klein wird.

   Bestimmen Sie die durchschnittliche Wartezeit pro Prozess.
   (10 Punkte)
2. Leiten Sie aus dem Beispiel für nicht-unterbrechendes und unterbrechendes Scheduling ein allgemeines Kriterium für die Wahl des jeweils zu aktivierende Prozesses ab, sodass die durchschnittliche Wartezeit möglichst klein wird.

   - Welche Strategie sollte beim nicht-unterbrechenden Scheduling angewendet werden? (5 Punkte)
   - Welche Strategie sollte beim unterbrechenden Scheduling angewendet werden? (5 Punkte)
3. Weist man den einzelnen Prozessen Prioritäten zu, kann beim Scheduling anhand dieser Prioritäten entschieden werden, welcher Prozess als nächstes gewählt wird. D.h., ein Prozess mit höhrer Priorität wird bevorzugt. Dies kann sowhol in einem nicht-unterbrechenden als auch in einem unterbrechenden Szenario betrachtet werden.

   Geben Sie Vor- und Nachteile von Scheduling mit Prioritäten an. Gibt es unterschiede zwischen einem nicht-unterbrechenden und unterbrechenden Szenario? (5 Punkte)

Aufgabe 2 (25 Punkte)

Betrachten Sie folgendes Beispiel, welches mit Round-Robin mit Zeitscheibenlänge 10 ausgeführt wird.

Prozesse	P_1	P_2	P_3	P_4	P_5	P_6
Ankunftszeit	0	5	7	10	15	20
Rechenzeit	6	5	10	2	1	8

1. Was beobachten Sie bei der gegeben Zeitscheibenlänge von 10? (5 Punkte)
2. Beschreiben Sie was passiert, wenn beim Round-Robin Verfahren die Zeitscheibenlänge zu klein oder zu groß gewählt wird. (10 Punkte)
3. Welches Ihnen aus der Vorlesung bekannte Scheduling Verfahren würden in dem oben dargestellten Beispiel eine bessere durchschnittliche Wartezeit erreichen? (5 Punkte)
4. Beschreiben die Funktionsweise eines Scheduling-Algorithmus der die Prozesse in einzelne Gruppen aufteilt. Gehen Sie dabei auf folgende Punkte ein.
   (a) In welche Gruppen könnten Prozesse aufgeteilt werden? (1 Punkt)
   (b) Welchen Gruppen sollte eine höhere Priorität zugewiesen werden? (2 Punkte)
   (c) Wie werden die einzelnen Gruppen verwaltet? (2 Punkte)

Praktischer Teil

1. Geben Sie die Prüfsumme mit dem Test *GdPI 02 - Testat* ab, der in Stud.IP *Grundlagen der Praktischen Informatik (Informatik II)* → *Lernmodule* hinterlegt ist.
2. Es ist **wichtig**, dass Sie den Test vollständig durchlaufen, damit die Daten ins System übertragen werden. D.h., Sie müssen den Test, nach dem Eintragen der Prüfsumme, mit der Schaltfläche ***Test beenden*** beenden.
3. Wenn Sie den Test einmal vollständig durchlaufen haben, kommen Sie auf die Seite *Testergebnisse*. Starten Sie den Test erneut aus Stud.IP, ist jetzt auch eine Schaltfläche *Testergebnisse anzeigen* vorhanden, die auf diese Seite führt.
4. Auf der Seite *Testergebnisse* können Sie sich unter *Übersicht der Testdurchläufe* zu jedem Testdurchlauf *Details anzeigen* lassen.
5. Um die praktische Übung testieren zu lassen, **müssen** Sie einen Termin über die Stud.IP-Terminvergabe für ein Testat reservieren. Ein Testat ohne Termin ist nicht möglich.
6. Die Testate können in Gruppen von bis zu drei Personen absolviert werden. Dabei sind die Gruppen identisch zu denen, die auch die theoretischen Aufgaben zusammen bearbeiten. In diesem Fall reserviert nur ein Gruppenmitglied einen Termin.
7. Bringen Sie bzw. jedes Mitglied Ihrer Gruppe zu jedem Testat den unten angehängten Kontrollzettel **Praktische Übung - Testate** mit. Ein Einspruch zu nicht oder falsch eingetragenen Punkten erfordert die Vorlage des Kontrollzettels.

Aufgabe 1 (50 Punkte)

1. Programmieren Sie eine Funktion

```
foldList :: (Double -> Double -> Double) -> [Double] -> Double

```

die alle Elemente einer Liste mit Hilfe der/des übergebenen Funktion/Operators `(Double -> Double -> Double)` zusammenfasst, entsprechend nachfolgender rekursiver Definition mit beliebiger Funktion $f : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ und Folge $x_n, \ldots, x_0$ wobei $x_i \in \mathbb{R}$.

$$\text{foldList}_f(x_n, \ldots, x_0) = \begin{cases} x_0 & \text{für } n = 0 \\ f\Big(x_n, \text{foldList}_f(x_{n-1}, \ldots, x_0)\Big) & \text{sonst} \end{cases}$$

Benutzen Sie nicht die Funktion `fold` aus `prelude`. Die Funktion sollte wie im folgenden Beispiel funktionieren.

```
> foldList (+) [1.0..100.0]
5050.0

```

(10 Punkte)

Hinweis.

- `[1.0..10.0]` ist die Liste der ganzahligen Gleitkommazahlen von 1.0 bis 10.0.
- Sie können davon ausgehen, dass die Funktion nur mit einer nicht leeren Liste aufgerufen wird.

2. Programmieren Sie eine Funktion

```
mapList :: (Int -> Int) -> [Int] -> [Int]

```

die als Funktionswert eine Liste von ganzen Zahlen `[Int]` liefert, die entsteht durch Anwendung einer übergebene Funktion `(Int -> Int)` auf jedes Element einer übergebenen Liste von ganzen Zahlen `[Int]`.

Benutzen Sie nicht die Funktion `map` aus `prelude`. Die Funktion sollte wie im folgenden Beispiel funktionieren.

```
> mapList square [1..10]
[1,4,9,16,25,36,49,64,81,100]
> mapList  (quadratic (1,2,4)) [1..10]
[7,12,19,28,39,52,67,84,103,124]

```

`[1..10]` ist die Liste der ganzen Zahlen von 1 bis 10. (10 Punkte)

3. Programmieren Sie eine Funktion

```
containsList :: [Int] -> Int -> Bool
```

die überprüft, ob ein Wert `Int` in einer Liste von Werten `[Int]` enthalten ist und einen entsprechenden Wahrheitswert `Bool` zurückliefert.
(15 Punkte)

Bemerkungen

- Benutzen Sie nicht die Funktion `elem` aus `prelude`, sondern **verwenden Sie Rekursion**.
- Die Funktion sollte wie im folgenden Beispiel funktionieren.

```
> containsList [-100..100] (-12)
True
```

4. Mit dem Kommando

```
> import Data.Char (toLower)
```

können Sie aus dem Modul `Data.Char` die Funktion `toLower :: Char -> Char` importieren, die sich auf alle `Char` anwenden lässt und für Großbuchstaben den zugehörigen Kleinbuchstaben und sonst den `Char` unverändert zurückliefert.

```
> import Data.Char (toLower)
> toLower 'S'
s
> toLower 's'
s
> toLower '#'
#
```

Programmieren Sie eine Funktion

```
countList :: [Char] -> Char -> Int
```

die zählt wie oft in der übergebenen Liste `[Char]` der übergebene `Char` enthalten ist und das Ergebnis zurückliefert. Dabei soll nicht zwischen Groß- und Kleinschreibung unterschieden werden (siehe `toLower`).
(15 Punkte)

Hinweis. `[Char]` und `String` haben dieselbe Bedeutung.