Vorlesung Systemnahe Informatik
Sommersemester 2023

Prof. Dr. Peter Martini, Dr. Matthias Frank, Lennart Buhl M.Sc.

5. Übungszettel

Ausgabe:	Dienstag, 09. Mai 2023.
Abgabe:	Sonntag, 14. Mai 2023
Besprechung:	In den Übungen ab Montag, 15. Mai 2023.
Hinweis:	Abgabe erfolgt freiwillig per PDF über eCampus, siehe Hinweise in Aufgabe 1 auf dem 1. Übungszettel Sie können Kontakt zu Ihrer Tutor/in aufnehmen durch E-mail an cs4+ueb-si-XX@cs.uni-bonn.de mit XX als Gruppennummer.

Besonderheit: Am Donnerstag 18.05.2023 ist Feiertag. Teilnehmer/innen unserer Donnerstags-Übungen können in dieser Woche in eine der anderen Präsenz-Übungen gehen. Teilnehmer/innen der Übungsgruppen am Brücken-Freitag 19.5. sprechen bitte das Vorgehen mit der Tutorin ab.

Aufgabe 1: Alpha-Notation, Queue

Eine Queue ist eine Datenstruktur, die nach dem FIFO-Prinzip (First In First Out) arbeitet. Bei einer Queue gibt es folgende Operationen:

- enqueue(x) - Hängt x an das Ende der Queue.
- dequeue - Entfernt das vorderste Objekt aus der Schlage.
- front - Liefert das vorderste Objekt der Queue

Bei dieser Aufgabe stehen Ihnen ausschließlich die folgenden Befehle zur Verfügung:

$\rho(i) := \alpha$	$\rho(i) := \rho(j)$ op α	$\rho(i) := \rho(j)$ op k	if $\alpha = 0$ then goto *label*
$\alpha := \rho(i)$	$\rho(i) := \rho(j)$	$\rho(i) := \alpha$ op k	goto *label*
$\alpha := \alpha$ op $\rho(i)$	$\rho(i) := \rho(\rho(j))$	$\alpha := \alpha$ op k	call *label*
$\alpha := \rho(i)$ op $\rho(j)$	$\rho(\rho(i)) := \rho(j)$	$\alpha := k$	return

a) Realisieren Sie obige Funktionen einer Queue mit Unterprogrammen in α-Notation. Überlegen Sie sich, wie Sie die Queue mit "indirekter Addressierung" geschickt im Speicher realisieren. Benennen Sie bitte wieder, welche Art von Unterprogrammen Sie gewählt haben. Um Fehlerfälle brauchen Sie sich nicht zu kümmern.

b) Beschreiben Sie, wie sich die Lage der Queue im Speicher bei Ihrer Implementierung verhält, wenn wiederholt Elemente eingefügt und entfernt werden. Können Probleme auftreten?

   Kommentieren Sie Ihren Code.

Aufgabe 2: Multi-Tasking und Multi-Threading

Bei der Entwicklung des vollautomatischen Tischstaubsauger-Roboters HooBot $X12$ wurde die Tischkantenerkennung leider etwas vernachlässigt. Dies hat zur Folge, dass er momentan Tischkanten nicht mehr *rechtzeitig* erkennt. Es stellt sich nun die Frage, ob man die Laufzeit der Tischkantenerkennung mittels Nebenläufigkeit verbessern kann. Das aktuell sequentielle Programm **BorderWalk** arbeitet die folgenden Aufgaben ab:

- `Collect`: In diesem Programmteil werden die Sensordaten der Tischkantensensoren ausgelesen und in einen Puffer $B1$ geschrieben.
- `Log`: Sowohl während der Testphase als auch beim endgültigen Produkt sollen die Sensordaten in einem nicht volatilen Speicher festgeschrieben werden. Hierzu greift der Programmteil nur lesend auf den Puffer $B1$ zu.
- `Stat`: Diese Funktion greift ebenfalls nur lesend auf den Puffer $B1$ zu und wertet die Sensordaten aus. Die resultierenden Daten werden in den Puffer $B2$ geschrieben.
- `Report`: Dieser Teil bereitet die Informationen aus dem Teil Stat für die Steuerung der Motoren auf und greift dafür lesend auf Puffer $B2$ zu.

Der Ablauf der sequentiellen Version mit CPU-Zeiten (blaue, große Balken) und IO-Zeiten (grüne, kleine Balken) ist in Abbildung 1 schematisch dargestellt.

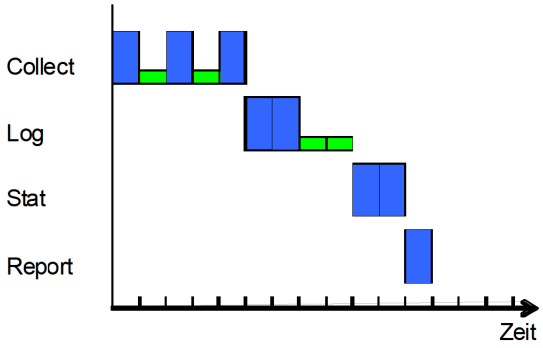

Abbildung 1: BorderWalk

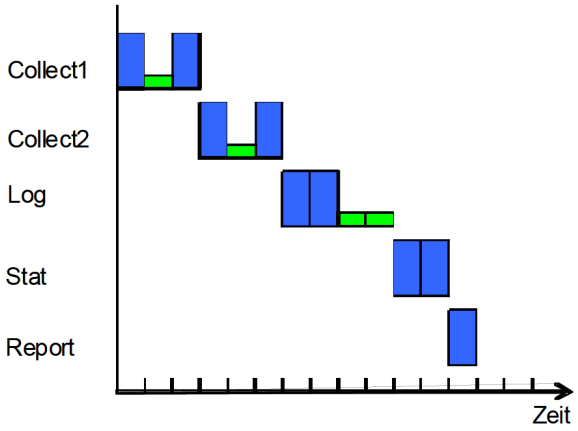

Abbildung 2: BorderCase

Neben der geplanten Implementierung wird auch noch der alternative Vorschlag **BorderCase** diskutiert. Einst wurde dieser Vorschlag in der sequentiellen Form verworfen, da er mehr CPU-Zeiten in Anspruch nimmt. Folgende Idee ist dabei eingebracht:

- `Collect 1`, `Collect 2`: Da mehrere Sensoren am HooBot angebracht sind, kümmern sich zwei getrennte Programmteile um die Abfrage entsprechender Sensoren. Die Sensordaten werden in den Puffer $B1.1$ bzw. $B1.2$ geschrieben.

a) Überlegen Sie, wie man die beiden sequentiellen Programme nebenläufig implementieren kann. Zeichnen Sie hierzu die Präzedenzgraphen der beiden Programme **BorderWalk** und **BorderCase**. Was beschreibt der Präzedenzgraph? (vgl. dazu die Vorlesungsfolien aus Kapitel 1 Teil 3; bei eigener Recherche ist u.U. ebenfalls zum Stichwort "Abhängigkeitsgraph" Hilfreiches zu finden).

b) Lesen Sie aus Abbildungen 1 und 2 entsprechend ab: Wie lange dauert die Bearbeitung eines sequentiellen Zyklus insgesamt und wie viele Zeiteinheiten davon sind reine CPU-Zeit?

c) Geben Sie die Ablaufdiagramme für die beiden nebenläufigen Programme an. Sollte man einem Programm den Vorzug geben? Begründen Sie ihre Entscheidung.

d) Geben Sie für die Lösungen Ihrer Ablaufdiagramme an: Wie lange dauert die Bearbeitung eines nebenläufigen Zyklus insgesamt und wie viele Zeiteinheiten davon sind reine CPU-Zeit?

Bezüglich der Prioritäten gilt:

- `Log` hat eine höhere Priorität als `Stat`.
- `Report` eine höhere als `Collect`.

Aufgabe 3: Java Bytecode

a) Betrachten Sie die unten angegebenen Funktionen f1, f2 im Java-Bytecode und kommentieren Sie, was in den einzelnen Zeilen passiert.

b) Formulieren Sie beide Methoden in Java und geben Sie an wofür diese Methoden eingesetzt werden können.

```
int f1(int,int,int);

 0:  iload_1
 1:  iload_2
 2:  ixor
 3:  iload_3
 4:  ixor
 5:  ireturn
```

```
int f2(int,int,int);

 0:  iload_1
 1:  iload_2
 2:  iand
 3:  iload_1
 4:  iload_3
 5:  iand
 6:  ior
 7:  iload_2
 8:  iload_3
 9:  iand
10:  ior
11:  ireturn
```