Technische Universität Berlin Fachgebiet Komplexe und Verteilte IT-Systeme Sommersemester 2017	**Aufgabenblatt 5** zu – Systemprogrammierung – Prof. Dr. Odej Kao, Florian Schmidt, Lauritz Thamsen, Tutoren
Abgabetermin:	[1] – 02.07.2017, 23:55 Uhr [2] – 09.07.2017, 23:55 Uhr

Aufgabe 5.1: (0,8 Punkte)
Betriebsmittelverwaltung mit Fremdbelegung (Theorie[1])

Ein teilbares Betriebsmittel soll mit Fremdbelegung verwaltet werden. Wir nehmen an, dass zum Zeitpunkt $t_0 = 0$ alle Einheiten belegt sind und die folgenden Anforderungen in der Warteschlange stehen:

$A_1 = 10$ Einheiten, $A_2 = 6$ Einheiten, $A_3 = 7$ Einheiten, $A_4 = 1$ Einheiten, $A_5 = 4$ Einheiten
(Anmerkung: A_1 steht ganz vorn und A_5 ganz hinten in der Warteschlange)

Die Anforderungen müssen dabei jeweils im Ganzen erfüllt werden, d.h. eine Anforderung kann nur erfüllt werden, wenn zum jeweiligen Zeitpunkt mindestens so viele Einheiten frei sind, wie angefordert werden. Angenommen zu den gegebenen Zeitpunkten werden nun folgende Belegungen freigegeben:

Zeitpunkt	$t_1 = 1$	$t_2 = 2$	$t_3 = 3$	$t_4 = 4$	$t_5 = 5$
Freigabe	$F_1 = 3$ Einh.	$F_2 = 13$ Einh.	$F_3 = 6$ Einh.	$F_4 = 2$ Einh.	$F_5 = 4$ Einh.

In welcher Reihenfolge werden die gegebenen Anforderungen bei Abarbeitung nach

a) FCFS/FIFO, (0,2 Punkte)

b) First-Fit-Request, (0,2 Punkte)

c) Best-Fit-Request und (0,2 Punkte)

d) Best-Fit-Request mit dynamischem Fenster von $L_{max} = 3$ (0,2 Punkte)

erfüllt? Geben Sie für jedes Verfahren außerdem die durchschnittliche Wartezeit an.

Reichen Sie für jedes Verfahren eine Lösung in Form der unten dargestellten Tabelle ein. Falls mehrere Aktionen zum selben Zeitpunkt stattfinden, *notieren Sie bitte jede Aktion in einer eigenen Spalte*.

Zeitpunkt	0	1	...
Freigaben	-	F_1	...
Anforderungen	-	-	...
Aktuell freie Einheiten	0	3	...

...

Geben Sie in Unteraufgabe d) außerdem in jeder Spalte zusätzlich die aktuelle Fenstergröße mit an.

Aufgabe 5.2: (1,2 Punkte)
Handsimulation des Bankieralgorithmus (Theorie[1])

Der Bankieralgorithmus kontrolliert Ressourcenallokationen, damit keine unsicheren Zustände auftreten.[1]

Gegeben ist die folgende verzahnte Ausführung der vier Prozesse P_1, P_2, P_3 und P_4:

	Zeit	Aktion
P_1	1	`allocate_r(A,3);`
P_1	2	`allocate_r(D,2);`
P_2	3	`allocate_r(B,3);`
P_2	4	`allocate_r(D,1);`
P_3	5	`allocate_r(C,3);`
P_3	6	`allocate_r(B,2);`
P_1	7	`release_r(A,3);`
P_1	8	`allocate_r(C,3);`
P_4	9	`allocate_r(B,1);`
P_4	10	`allocate_r(D,1);`
P_3	11	`allocate_r(C,1);`
P_3	12	`release_r(C,4);`
P_2	13	`allocate_r(A,1);`
P_2	14	`release_r(B,3);`

	Zeit	Aktion
P_1	15	`allocate_r(D,1);`
P_1	16	`release_r(C,3);`
P_4	17	`allocate_r(D,2);`
P_4	18	`release_r(B,1);`
P_3	19	`release_r(B,2);`
P_3	20	`exit();`
P_1	21	`release_r(D,3);`
P_1	22	`exit();`
P_2	23	`release_r(A,1);`
P_2	24	`release_r(D,1);`
P_2	25	`exit();`
P_4	26	`release_r(D,3);`
P_4	27	`exit();`

Die Funktionen `allocate_r` und `release_r` erhalten hierbei jeweils die Kennung für eines der Betriebsmittel A, B, C oder D und die angeforderte bzw. freizugebende Anzahl als Parameter.
Von jedem Betriebsmittel sind zu Beginn 4 Einheiten vorhanden und nicht belegt.

Führen Sie eine Handsimulation für den gegebenen Ablauf durch. Geben Sie dabei für jede Allokation die Belegungen, die Restanforderungen und die freien Betriebsmittel in der aus der Vorlesung bekannten Matrixschreibweise an, prüfen Sie mit dem Bankieralgorithmus, ob die Belegung zu einem unsicheren Zustand führt oder nicht und geben Sie ggf. eine mögliche sichere Ausführungsreihenfolge an. Tritt ein unsicherer Zustand auf oder sind zum Zeitpunkt der Anfrage nicht genug Betriebsmittel vorhanden, wird der dazugehörige Prozess bis zum Ende der Handsimulation blockiert. Blockierte Prozesse geben ihre Betriebsmittel *nicht* frei.

Aufgabe 5.3:
Betriebsmittelverwaltung (Tafelübung)

Untersuchen Sie die folgende Belegungssituation:

Belegung: $B = \begin{pmatrix} 5 & 1 \\ 1 & 1 \\ 2 & 2 \end{pmatrix}$ Gesamtanforderung: $G = \begin{pmatrix} 9 & 3 \\ 2 & 5 \\ 3 & 3 \end{pmatrix}$ Freie Ressourcen: $f = \begin{pmatrix} 1 & 1 \end{pmatrix}$

[1] Dieser Ansatz ist bei Banken inzwischen scheinbar in Vergessenheit geraten, kommt dort heute doch bevorzugt eine andere Strategie zum Einsatz: `https://de.wikipedia.org/wiki/Bail-out_(Wirtschaft)`

a) Wie sieht die Restanforderungs-Matrix aus?

b) Sind im System genügend Ressourcen vorhanden, um theoretisch alle Anforderungen zu erfüllen?

c) Erläutern Sie, was ein sicherer Zustand ist und prüfen Sie, ob es sich hier um einen solchen handelt!

d) Wie sieht es aus, wenn $f = \begin{pmatrix} 2 & 1 \end{pmatrix}$ ist?

e) Was kann man üblicherweise tun, wenn sich ein System in einem Deadlock befindet?

Aufgabe 5.4:
Scheduling mit Prioritäten (Tafelübung)

Recherchieren Sie, welches Problem beim Mars Rover Pathfinder wiederholt zu Systemresets geführt hat. Welches grundlegende Problem liegt dem zugrunde und wie lässt es sich beheben?

Aufgabe 5.5:
Verklemmung (Tafelübung)

An einem Frühstückstisch in einer WG (Uli, Shanti, Gwen und Lara) gibt es Käse, Toast, Marmelade, Brot und Peanutbutter. Jeden Morgen entsteht ein erbitterter Kampf um die Ressourcen. Die Auseinandersetzung am Frühstückstisch kann näherungsweise durch folgendes C-Programm beschrieben werden:

```
1  // Ressourcen
2  // von jeder Ressource gibt es genau 1 Instanz, die nicht verbraucht wird
3  single_unit_R peanutbutter, jelly, cheese, toast, bread;
4  // Uli - Prozess 1:
5  int main() {
6      allocate_r(&cheese);
7      allocate_r(&jelly);
8      allocate_r(&bread);
9      eat(); // Brot mit Jelly und Cheese
10     release_r(&bread);
11     release_r(&jelly);
12     release_r(&cheese);
13     return 0;
14 }
15
16 // Shanti - Prozess 2:
17 int main() {
18     allocate_r(&peanutbutter);
19     allocate_r(&bread);
20     eat(); // Peanutbutter & Brot
21     release_r(&bread);
22     release_r(&peanutbutter);
23     return 0;
24 }
25
26 //Lara - Prozess 3:
27 int main() {
28     allocate_r(&toast);
```

```
29      allocate_r(&cheese);
30      eat(); // Käse-Toast (Lara ist auf Diät)
31      release_r(&cheese);
32      release_r(&toast);
33      return 0;
34  }
35
36  // Gwendolin - Prozess 4:
37  int main() {
38      allocate_r(&jelly);
39      allocate_r(&peanutbutter);
40      allocate_r(&toast);
41      eat(); // it's Peanut Butter Jelly Time!!
42      release_r(&toast);
43      release_r(&peanutbutter);
44      release_r(&jelly);
45      return 0;
46  }
```

Es sei ein System gegeben, bei welchem jeder Prozess 2 Befehle ausführen kann, bevor zum nächsten Prozess gewechselt wird. Die Reihenfolge der Ausführung soll wie oben Uli→Shanti→Lara→Gwen→Uli. . . sein.

a) Simulieren Sie den Frühstückstisch. Geben Sie in jedem Schritt an, wer welche Ressource hat oder freigibt.

b) Wie sieht der Ablauf aus, wenn jeder Prozess nur einen Befehl ausführen kann, bevor zum nächsten Prozess gewechselt wird?

c) Zeichnen Sie den Betriebsmittelgraph wie er am Ende des Ablaufs aus Teil b) aussieht.

d) Welche Bedingungen sind notwendig und welche hinreichend, damit eine Verklemmung auftreten kann?

Aufgabe 5.6: (3 Punkte)
Implementierung der Verklemmungsbehandlung (Praxis^2)

In dieser Aufgabe sollen Sie schrittweise ein **C-Programm** entwickeln, das gesicherte Ressourcenbelegungen und -freigaben simuliert. Die Ausgangssituation Ihrer Simulationen soll in einer simplen Textdatei (aber natürlich mit festgelegtem Format) definiert und dann durch Ihr Programm ausgelesen werden.

a) (0,2 Punkte) Schreiben Sie ein Programm, das zwei Argumente auf der Kommandozeile erwartet: die Dateipfade zu einer Eingabe- sowie einer Ausgabedatei. Die Eingabedatei sollte bereits existieren und dem nachfolgend beschriebenen Format folgen (falls nicht, brechen Sie Ihr Programm ab!):

```
 2          // Anzahl der simulierten Prozesse
 3          // Anzahl der simulierten Betriebsmittel

// Gesamtanforderungen
 1  1  1    // jede Zeile entspricht den Anforderungen eines Prozesses
 3  3  0    // die Spalten entsprechen demnach den Betriebsmitteln

// aktuelle Belegungen
 1  0  0    // jede Zeile entspricht den Anforderungen eines Prozesses
 1  0  0    // die Spalten entsprechen demnach den Betriebsmitteln

 3  3  3    // insgesamt verfügbare Betriebsmittel
```

Listing 1: Beispiel einer valide formatierten Eingabedatei

Lesen Sie die Eingabedatei ein und schreiben Sie die ausgelesenen Werte (Prozesszahl, Betriebsmittelzahl, Gesamtanforderungen, Belegungsmatrix, verfügbare Betriebsmittel) in die Ausgabedatei. Achten Sie darauf, dass die Ausgabedatei *exakt* so formatiert ist, wie nachfolgend dargestellt:

```
Prozesse: 2 / Betriebsmittel: 3

Gesamtanforderungen:
 1  1  1
 3  3  0

Belegungsmatrix:
 1  0  0
 1  0  0

verfügbar:  3  3  3
```

Listing 2: Beispiel einer valide formatierten Ausgabedatei

Zum Testen Ihres Programms finden Sie in den ISIS-Vorgaben die beiden Dateien `a_b_in.txt` und `a_out.txt`. Wenn Sie `a_b_in.txt` als Eingabedatei für Ihr Programm verwenden, sollte es eine Ausgabedatei erstellen, deren Inhalt sich in absolut nichts von `a_out.txt` unterscheidet.

Überprüfen können Sie dies z.B. mit dem Kommandozeilen-Tool `diff` (vgl. `man diff`):

```
diff -q -s <Dateiname ihrer Ausgabedatei> a_out.txt
```

Wenn Sie das Flag `-q` entfernen, können Sie sehen, worin genau sich beide Dateien unterscheiden.

b) (0,3 Punkte) Berechnen Sie aus den eingelesenen Werten nun die Restanforderungsmatrix sowie den Vektor der freien Betriebsmittel und geben Sie diese zusätzlich am Ende der Ausgabedatei aus:

```
1 Restanforderungen:
2  0  1  1
3  2  3  0
4
5 frei:  1  3  3
```

Listing 3: Zusätzlich hinzugekommenes Ausgabeformat

Wie diese berechnet werden, können Sie den Vorlesungsfolien oder der Tafelübung entnehmen.

Um die Korrektheit Ihrer Ergebnisse und des Ausgabeformats zu überprüfen, können Sie Ihre Ausgabe für die Eingabedatei `a_b_in.txt` mit der Datei `b_out.txt` aus den Vorgaben vergleichen.

c) (1,3 Punkte) Erweitern Sie Ihr Programm nun um eine Implementierung des Bankieralgorithmus (engl.: banker algorithm) zur Verklemmungsentdeckung (engl.: deadlock detection). Ist die durch die verwendete Eingabedatei gegebene Situation sicher, soll ihr Programm den String `SICHER`, andernfalls den String `UNSICHER` in einer eigenen Zeile an das Ende der Ausgabedatei schreiben.

```
1 SICHER
```

```
1 UNSICHER
```

Listing 4: Zusätzlich hinzugekommenes Ausgabeformat

Machen Sie sich hierfür nochmals bewusst, was genau ein (un-)sicherer Zustand ist.

Auch für diese Aufgabe stellen wir Ihnen im ISIS-Kurs passende Eingabedateien zur Vorgabe bereit: `sicher_in.txt` und `unsicher_in.txt`, die ihrem Namen entsprechend eine sichere bzw. unsichere Situation beschreiben. Die Ausgabe Ihres Programms für diese beiden Dateien sollte inhaltlich folglich den Dateien `sicher_out.txt` bzw. `unsicher_out.txt` entsprechen.

d) (1,0 Punkte) Passen Sie Ihr Programm nun so an, dass es auch zur Verklemmungsvermeidung (engl.: deadlock avoidance) verwendet werden kann. Die für diese Aufgabe auf ISIS bereitgestellten Eingabedateien enthalten hierzu am Schluss einen zusätzlichen Abschnitt mit folgendem Format:

```
1 A 1 0 1   // <Operation> <Prozessnummer> <Betriebsmittel> <Anzahl>
2 R 0 0 1   // Mögliche Operationen: A = Allocate, R = Release
3 A 1 2 3   // Nummerierung startet bei 0 (bei Prozessen von oben,
4 ...       // bei Betriebsmitteln von links ausgehend)
```

Listing 5: Zusätzlich hinzugekommenes Eingabeformat

Ihr Programm soll jede dieser Belegungen und Freigaben (in der Reihenfolge ihrer Nennung von oben nach unten) simulieren. Schreiben Sie für jede valide Operation, für die garantiert werden kann, dass sie nicht zu einer Verklemmung (engl.: deadlock) führt, die danach vorliegende Situation, bestehend aus der Restanforderungsmatrix und dem Vektor der freien Ressourcen, ans Ende der Ausgabedatei:

```
1 Operation: A  1  0  1
2
3 Restanforderungen:
4  0  1  1
5  0  3  0
6
7 f: 0 3 3
8
9 ...
```

Listing 6: Zusätzlich hinzugekommenes Ausgabeformat

Falls die Ausführung einer Operation zu einer Verklemmung führen könnte, ignorieren Sie diese, schreiben Sie nichts in die Ausgabe und setzen Sie die Simulation mit der nächsten Operation fort.

Zum Testen können Sie die Vorgabe-Dateien `d_in.txt` und `d_out.txt` verwenden.

e) (0,2 Punkte) Schreiben Sie ein Makefile für ihr Programm und geben Sie dieses mit ihrem restlichen Code auf ISIS ab. Sie können sich hierbei an den Makefiles der vorherigen Aufgaben orientieren oder ein komplett neues Makefile schreiben. Ihr Makefile sollte mindetsens die Targets `all` und `clean` beinhalten, sodass ein wiederholtes Kompilieren für uns mit minimalem Aufwand möglich ist.

Weitere Hinweise:

- Zum Einlesen der Eingabedateien kann z.B. die Funktion `fscanf()` verwendet werden.
- Die Kommentare in den Eingabedateien auf diesem Blatt dienen lediglich der Einführung in die verwendete Formatierung. Die Vorgaben selbst (sowie unsere Tests) sind unkommentiert.
- Falls es Ihnen leichter fallen sollte, dürfen Sie weiterhin annehmen, dass in den Eingabedateien, die wir zum Testen verwenden, nur ein- oder zweistellige Zahlenwerte enthalten sein werden.
- Wir entfernen überflüßige Leerzeilen und -zeichen in Ihren Ausgabedateien automatisch. Falls Ihre Ausgabedateien also nur diesbezüglich von unseren Vorgaben abweichen, ist das in Ordnung.
- Die korrekte Einrückung aller Zahlenwerte, auch wenn sie nicht durchgängig zweistellig sind, können Sie durch die Verwendung des Formatstring `%2d` statt `%d` bei der Zahlenausgabe erreichen. Sie dient aber nur der besseren Lesbarkeit und ist zum Bestehen - siehe Hinweis darüber - nicht nötig.
- Wie Sie die eingelesenen und berechneten Daten innerhalb Ihres Programms speichern, wird *nicht* bewertet! Es empfiehlt sich aber eine gut skalierbare und leicht zu handhabende Lösung.
- Achten Sie außerdem auf wiederholt anfallende Aufgaben in Ihrem Programm und lagern Sie diese nach Möglichkeit in separate Funktionen aus, um Ihren Arbeitsaufwand zu reduzieren.
- Sie können selbstverständlich auch eigene Eingabedateien schreiben und diese mit ihren Kommilitonen (gern auch über das Diskussionsforum auf ISIS) austauschen, um Ihr Programm zu testen. In jedem Fall sollten Sie sicherstellen, dass Ihr Programm mit beliebigen Eingaben korrekt arbeitet.