Ludwig-Maximilians-Universität München
Institut für Informatik
Lehrstuhl für Mobile und Verteilte Systeme
Prof. Dr. Claudia Linnhoff-Popien

mobile and distributed systems group

Übungsblatt 13
Betriebssysteme im WiSe 2020/2021

Besprechung: Das Übungsblatt dient der Vorbereitung auf die Klausur.

Ankündigungen:

- Die **Online-Hausarbeit** findet am **20. Februar 2021 im Zeitraum von 14 - 17 Uhr** statt. Bitte melden Sie sich **bis spätestens 15. Februar 2021, 10:00 Uhr** über Uni2work dazu **an** bzw. **ab**.
- Am Mittwoch, den **10. Februar 2021 findet von 14 - 16:15 Uhr s.t. ein Probelauf zur Durchführung der Online-Hausarbeit statt.** Der Probelauf wird nicht die komplette Zeit in Anspruch nehmen. Zudem wird in der Zeit im Rahmen eines Sondertutoriums den Studierenden die Gelegenheit gegeben noch einmal gezielt Fragen zum Stoff zu stellen. Zentraler Anlaufpunkt ist das Zoom-Meeting zur Vorlesung `https://lmu-munich.zoom.us/j/95188149894?pwd=aGw0WC96cm5BMXhYc1RKd2RaRWptZz09`.

Aufgabe K34: Einfachauswahlaufgabe

(– Pkt.)

Für jede der folgenden Fragen ist eine korrekte Antwort auszuwählen („1 aus n“). Kreuzen Sie dazu die jeweils ausgewählte Antwortnummer ((i), (ii), (iii) oder (iv)) an. Eine korrekte Antwort ergibt jeweils einen Punkt. Mehrfache Antworten oder eine falsche Antwort werden mit 0 Punkten bewertet.

a) Worüber können vom Hauptprogramm schnellstmöglich eine geringe Anzahl von Parametern an ein Unterprogramm übergeben werden?

(i) Stack	(ii) Spezielle CPU-Register	(iii) Speicherbereich auf der Festplatte	(iv) Speicherbereich auf einem Bandlaufwerk

b) Wie ist die mittlere Antwortzeit bei Prozessen mit folgender Ressourcennutzung unter Anwendung von Multiprogramming?

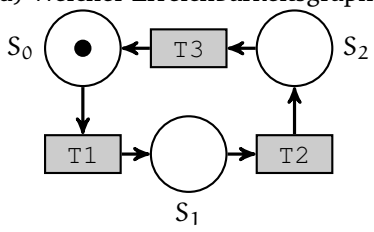
Job	durchschnittliche CPU-Auslastung	Dauer	benötigter Speicher	Platte	Terminal	Drucker
1	20%	10 min.	50 KBytes	-	-	-
2	45%	20 min.	100 KBytes	-	ja	-
3	25%	30 min.	80 KBytes	ja	-	ja

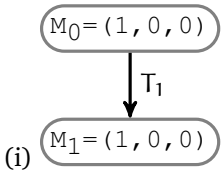
(i) 3,33 min.	(ii) 6,66 min.	(iii) 10 min.	(iv) 20 min.

c) Mit welchem Systemaufruf wird unter Unix/Linux Systemen eine identische Kopie des aufrufenden Prozesses erzeugt?

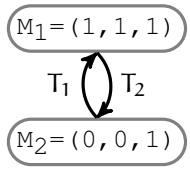
(i) `close`	(ii) `fork`	(iii) `creat`	(iv) `getpid`

d) Welcher Erreichbarkeitsgraph gehört zu folgendem Petrinetz?

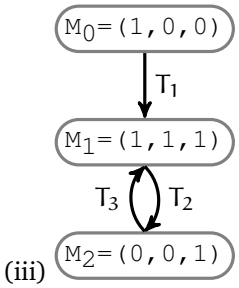




(i)

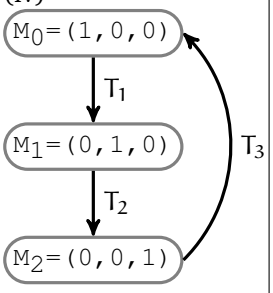


(ii)



(iii)

(iv)



e) Für eine korrekte Lösung des wechselseitigen Ausschlusses müssen drei Bedingungen erfüllt sein. Was ist keine davon?

(i) Mutual Exclusion	(ii) Correlated Blocking	(iii) Progress	(iv) Bounded Waiting

Aufgabe K35: Prozessfortschrittsdiagramm

(– Pkt.)

Bearbeiten Sie die folgenden Aufgaben.

a. Gegeben seien zwei Prozesse P und Q, die auf einem Uniprozessorsystem ausgeführt werden sollen. Der Prozess P benötigt 9 und der Prozess Q 10 Zeiteinheiten für seine Ausführung. Es stehen die Betriebsmittel BM 1–6 zur Verfügung, die von den Prozessen während ihrer Ausführung benötigt werden.

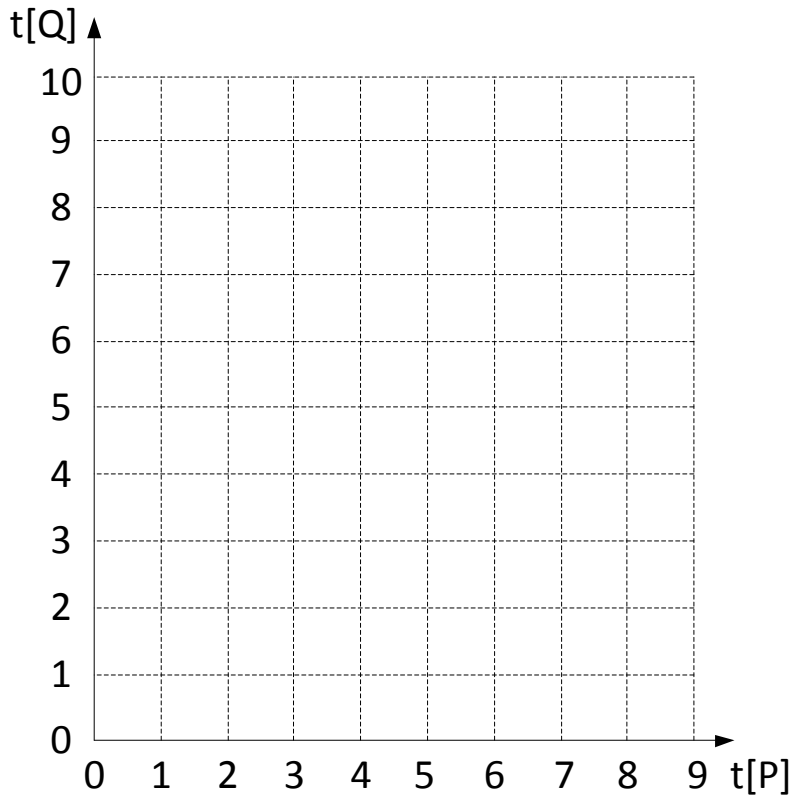
Q benötigt:

- BM1 im Zeitraum $]1;3[$,
- BM2 im Zeitraum $]1;2[$,
- BM3 im Zeitraum $]6;7[$,
- BM4 im Zeitraum $]1;2[$,
- BM5 im Zeitraum $]5;7[$ und
- BM6 im Zeitraum $]4;6[$.

P benötigt:

- BM1 im Zeitraum $]2;3[$,
- BM2 im Zeitraum $]2;4[$,
- BM3 im Zeitraum $]4;6[$,
- BM4 im Zeitraum $]5;7[$,
- BM5 im Zeitraum $]5;8[$ und
- BM6 im Zeitraum $]7;8[$.

Skizzieren Sie das Prozessfortschrittsdiagramm für die oben beschriebenen Anforderungen, indem Sie die benötigten Betriebsmittel entsprechend ihrer zeitlichen Verwendung durch die beiden Prozesse P und Q korrekt einzeichnen. Gehen Sie davon aus, dass der Scheduler die Prozesse P und Q zu beliebigen Zeitpunkten aktivieren bzw. suspendieren kann. Gehen Sie zudem davon aus, dass ein Kontextwechsel zwischen P und Q keinerlei Zeit in Anspruch nimmt. Tragen Sie Ihre Lösung in die folgende Vorlage ein:



b. Kennzeichnen Sie **deutlich** alle unmöglichen und unsicheren Bereiche im Diagramm aus Teilaufgabe a).

c. Zeichnen Sie alle *prinzipiell* unterschiedlichen Ausführungspfade (d.h., dass Prozess P und Q anders ge-scheduled werden und dementsprechend die Betriebsmittel in unterschiedlicher Reihenfolge nutzen) in das Diagramm aus Teilaufgabe a) ein, so dass die Prozesse P und Q terminieren.

d. Bezogen auf Teilaufgabe a): Wieviele *prinzipiell* unterschiedliche Ausführungspfade gibt es (d.h., dass Prozess P und Q anders ge-scheduled werden und dementsprechend die Betriebsmittel in unterschiedlicher Reihenfolge nutzen), die in einem Deadlock enden?
Geben Sie für jeden solchen Ablauf ein Beispiel an und beschreiben sie dabei, wann und wie lange Prozess P bzw. Q aktiviert bzw. suspendiert werden muss, um in eine Deadlock-Situation zu gelangen.

Aufgabe K36: Scheduling

(– Pkt.)

In dieser Aufgabe sollen zwei Scheduling-Strategien untersucht werden: die nicht-preemptive Strategie **SJF (Shortest Job First)** und die preemptive Strategie **RR (Round Robin)**. Dazu seien die folgenden Prozesse mit ihren Ankunftszeitpunkten und Bedienzeiten (in beliebigen Zeiteinheiten) gegeben.

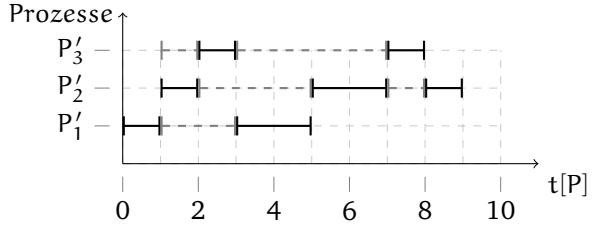
Prozess	Ankunftszeitpunkt	Bedienzeit
P_1	1	7
P_2	1	5
P_3	0	3
P_4	4	1
P_5	5	2

- Trifft ein Prozess zum Zeitpunkt t ein, so wird er direkt zum Zeitpunkt t berücksichtigt.
- Wird ein Prozess zum Zeitpunkt t' unterbrochen, so reiht er sich auch zum Zeitpunkt t' wieder in die Warteschlange ein.
- Sind zwei Prozesse absolut identisch bezüglich ihrer relevanten Werte, so werden die Prozesse nach aufsteigender Prozess-ID in der Warteschlange eingereiht (Prozess P_i vor Prozess P_{i+1}, usw.). Diese Annahme gilt sowohl für neu im System eintreffende Prozesse, als auch für den Prozess, dem der Prozessor u.U. gerade entzogen wird!
- Jeder Prozess nutzt sein Zeitquantum stets vollständig aus d.h. kein Prozess gibt den Prozessor freiwillig frei (Ausnahme: bei Prozessende).

Beispiel: Es seien folgende Ankunfts– und Bedienzeiten für die drei Beispielprozesse P'_1, P'_2 und P'_3 gegeben:

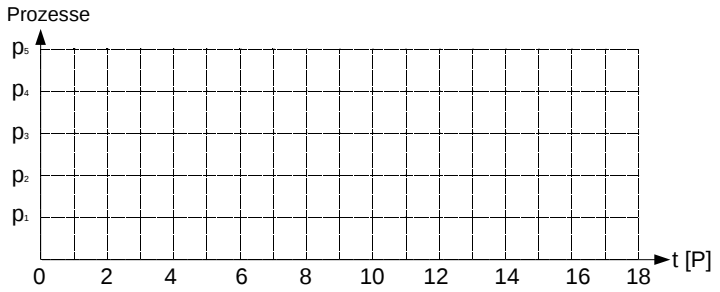
Prozess	Ankunftszeitpunkt	Bedienzeit
P'_1	0	3
P'_2	1	4
P'_3	1	2

Das folgende Diagramm veranschaulicht ein beliebiges Scheduling der drei Prozesse P'_1, P'_2 und P'_3:



Bearbeiten Sie unter den gegebenen Voraussetzungen nun die folgenden Aufgaben:

a. Verwenden Sie nun die **nicht-preemptive Strategie SJF** und erstellen Sie entsprechend dem vorherigen Beispiel ein Diagramm, das für die Prozesse P_1–P_5 angibt, wann welchem Prozess Rechenzeit zugeteilt wird und wann die Prozesse jeweils terminieren. Kennzeichnen Sie zudem für jeden Prozess seine Ankunftszeit. Erstellen Sie Ihre Lösung basierend auf folgender Vorlage:



b. Geben Sie für die **preemptive Strategie RR** an, wann welchem Prozess Rechenzeit zugeteilt wird und wann die Prozesse jeweils terminieren, indem Sie Ihre Lösung wie in der vorherigen Teilaufgabe a) darstellen. Die Dauer einer **Zeitscheibe betrage 2 Zeiteinheiten**. Gehen Sie davon aus, dass jeder Prozess die Dauer seiner Zeitscheibe stets vollständig ausnutzt, sofern er nicht terminiert. Terminiert ein Prozess vor Ablauf seiner Zeitscheibe, gibt er den Prozessor zum Zeitpunkt der Terminierung sofort frei. Trifft genau nach Ende einer Zeitscheibe ein neuer Prozess ein, so wird der neue Prozess **vor** dem gerade unterbrochenen Prozess in die Warteschlange eingereiht.

c. Berechnen Sie als Dezimalzahl mit einer Nachkommastelle die mittlere Verweil- und Wartezeit für die zwei Verfahren SJF und RR.

Aufgabe K37: Petrinetze

(– Pkt.)

In dieser Aufgabe soll eine Modifikation des Erzeuger-/Verbraucherproblems als Petrinetz modelliert werden. Beim klassischen Erzeuger-/Verbraucherproblem liegt folgende Situation vor:

– Es gibt zwei Prozesse. Einen Erzeuger (E) und einen Verbraucher (V).

– E produziert Resultate (R), die zunächst in einem gemeinsamen Speicher (S) der Kapazität max abgelegt werden.

– V entnimmt die Resultate dem gemeinsamen Speicher S und verbraucht diese.

– V darf nur mittels des gemeinsamen Speichers S auf ein Resultat R zugreifen.

– E darf nur bei freien Plätzen ein Resultat R im gemeinsamen Speicher S ablegen.

– Der Speicher darf nicht gleichzeitig von E und V verändert werden.

Dieses klassische Problem wird nun durch Einführen der folgenden zusätzlichen Bedingung modifiziert:

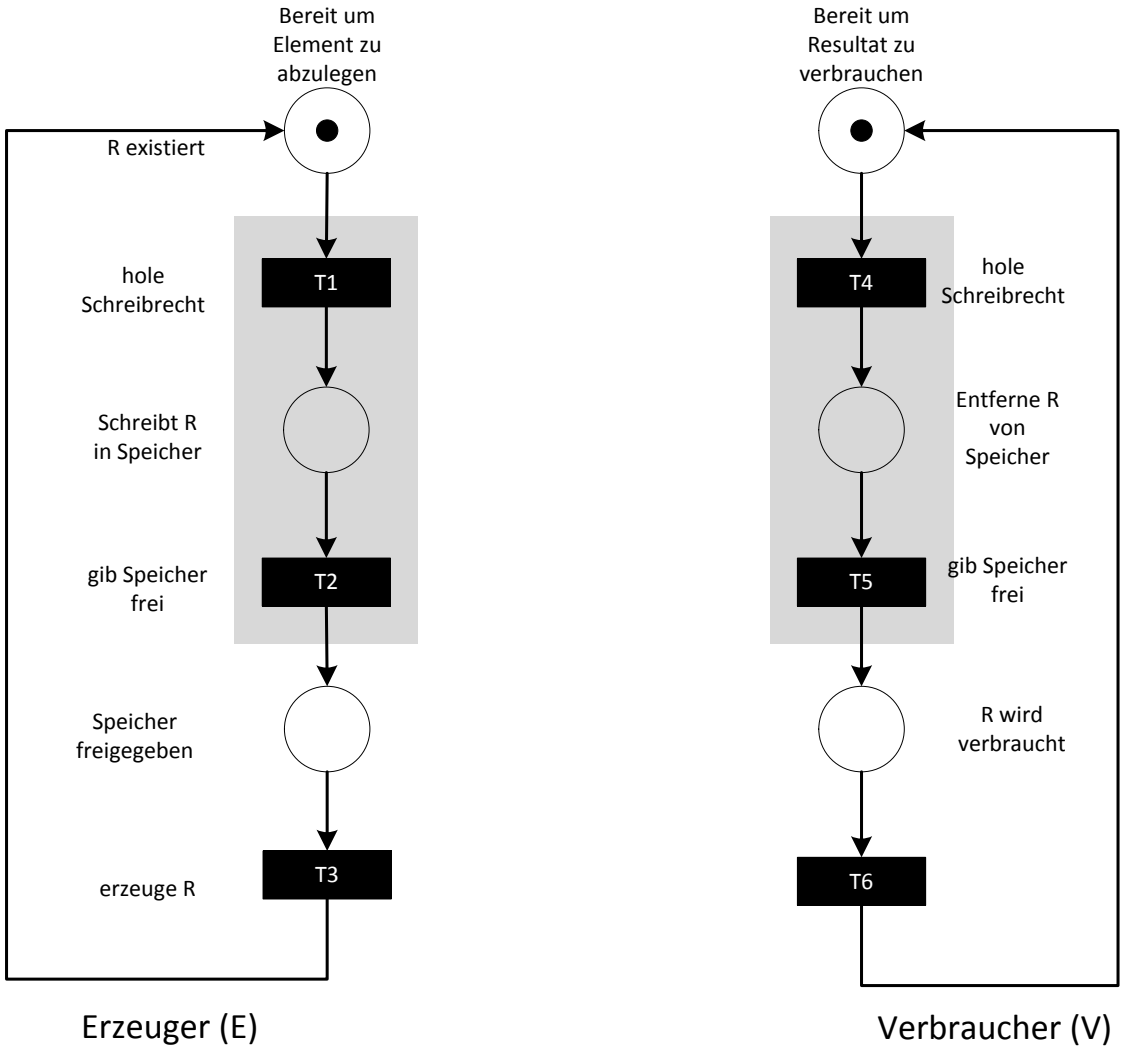
– E darf nur ein Resultat R in den Speicher schreiben, wenn danach noch mindestens ein freier Platz im gemeinsamen Speicher S vorhanden ist.

Bearbeiten Sie unter Berücksichtigung der gegebenen Anforderungen nun die folgenden Teilaufgaben:

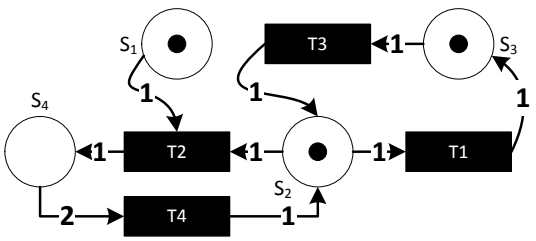
a. Modellieren Sie das beschriebene modifizierte Erzeuger-/Verbraucherproblem durch ein Petrinetz. Ergänzen Sie dazu den umseitig gegebenen Rahmen, indem Sie die minimal notwendigen Stellen, Marken, Kanten und Transitionen ergänzen. Versehen Sie alle Stellen und Transitionen mit aussagekräftigen Bezeichnern. Gehen Sie von folgenden Bedingungen aus:

- Die Kapazität max des gemeinsamen Speichers S beträgt 4.
- Zu Begin befindet sich bereits ein fertiges Resultat im gemeinsamen Speicher S.

Hinweis: Überlegen Sie sich zunächst die Funktionsweise des klassischen Erzeuger-/Verbraucherproblems und leiten Sie darauf basierend die nötigen Anpassungen her.



b. Gegeben sei nun folgendes Petrinetz:



Leiten Sie den Erreichbarkeitsgraphen für dieses Petrinetz her. *(Platz zur Fortsetzung der Lösung von Teilaufgabe b))*

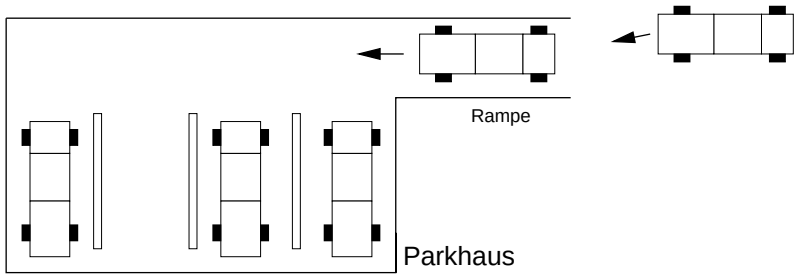
c. Begründen Sie jeweils separat, ob im Petrinetz aus Teilaufgabe b ein Deadlock bzw. ein partieller Deadlock entstehen kann.

Aufgabe K38: Parkhauskontrolle mit Semaphoren

(– Pkt.)

In dieser Aufgabe sollen Sie das Konzept der Semaphore am Beispiel eines Parkhauses umsetzen. Dazu soll die Ein- und Ausfahrt von Autos in ein Parkhaus mit `4` Stellplätzen simuliert werden. Autos (welche hier als Prozesse angesehen werden können) können ein- und ausfahren. Dabei dürfen sich stets maximal soviele Autos im Parkhaus befinden (inklusive eines Autos auf der Rampe), wie Stellplätze vorhanden sind. Zu einem bestimmten Zeitpunkt kann sich immer nur ein Auto auf der Rampe befinden.

Die nachfolgende Abbildung zeigt einen Beispielzustand des Parkhauses, wo drei von den vier Stellplätzen bereits belegt sind. Ein Auto befindet sich gerade auf der Einfahrt in das Parkhaus und ein Auto wartet auf das Freiwerden der Einfahrt.



a. Was sind die kritischen Bereiche bei diesem Problem?

b. Wieviele Semaphore benötigt man um ein- und ausfahrende Autos zu synchronisieren? Um welche Art von Semaphor handelt es sich jeweils?

c. Geben Sie in Pseudocode an, wie die benötigten Semaphore initialisiert werden müssen. Wählen Sie sinnvolle Bezeichner für Ihre Semaphore. Verwenden Sie folgende Notation für die Initialisierung:

Pseudocode	Beispiel	Bedeutung
`init(<semaphor>,<value>);`	`init(mutex,1);`	Initialisiert den Semaphor `mutex` mit dem Wert 1.

d. Vervollständigen Sie nun den folgenden Pseudocode, so dass dieser das Ein- und Ausfahren eines Autos simuliert und mehrere Autos stets synchronisiert werden.

```
1  auto() {
2     while(true) {
3        ...
4        <in das Parkhaus einfahren>;
5        ...
6        <parken>;
7        ...
8        <aus dem Parkhaus herausfahren>;
9        ...
10    }
11 }
```

Verwenden Sie für den Zugriff auf Ihre Semaphore folgende Notation:

Pseudocode	Beispiel	Bedeutung
`wait(<semaphor>);`	`wait(mutex);`	Erniedrigt den Wert des Semaphor `mutex` um eins
`signal(<semaphor>);`	`signal(mutex);`	Erhöhe den Wert des Semaphor `mutex` um eins

e. Das Parkhaus soll nun auch LKWs die Möglichkeit zum Parken geben. Dazu werden zwei der vier vorhandenen Stellplätze vergrößert. Es gibt fortan also vier Parkplätze wovon zwei als LKW Parkplatz genutzt werden können. LKWs können nur auf LKW Parkplätzen parken, Autos dürfen aber weiterhin auf allen Parkplätzen parken. Unabhängig von der Art kann auf einem Parkplatz immer nur ein Fahrzeug parken. Gehen Sie davon aus, dass ein Prozess stets über die boolesche Variable `is_lkw_parkplatz` testen kann, ob es sich bei dem freien Parkplatz um einen Parkplatz für LKWs handelt.

(i) Damit das Parken von LKWs und Autos weiterhin synchronisiert erfolgen kann, benötigt man weitere Semaphore. Wählen Sie geeignete Bezeichner für die neuen Semaphore und initialisieren Sie diese in Analogie zu Aufgabe c).

(ii) Geben Sie in Analogie zum Pseudocode für Autos aus Aufgabe d) den Pseudocode für LKWs an, die in dem Parkhaus parken wollen. Nennen Sie die entsprechende Funktion `lkw()`.

(iii) Verändern Sie nun den Pseudocode aus der Funktion `auto()` aus Aufgabe d) so, dass Autos bzw. LKWs (d.h. die ausführenden Prozesse der Funktionen `auto()` und `lkw()`) synchronisiert werden. Verwenden Sie dazu geeignete `if`-Konstrukte.

Aufgabe K39: Seitenersetzungsstrategien

(– Pkt.)

Bearbeiten Sie die folgenden Aufgaben zum Thema Seitenersetzung (Paging).

Die Menge der Seiten (Pages) sei gegeben durch $N = \{0, 1, 2, 3, 4\}$ und die Menge der Seitenrahmen (Frames), die für die Speicherung der Seiten im Arbeitsspeicher zur Verfügung steht, sei gegeben durch $Frame_3 = \{f_0, f_1, f_2\}$. Auf die fünf Seiten der Menge N werde in folgender Reihenfolge zugegriffen:

```
w = 1 3 1 0 1 0 2 4 4 1 3 1
```

Ein Seitenfehler liegt immer dann vor, wenn sich eine referenzierte Seite nicht im Arbeitsspeicher befindet. Dieser ist zu Beginn leer. Ermitteln Sie die Anzahl der Seitenfehler für die folgenden Seitemersetzungsstrategien, indem Sie alle Veränderungen im Speicher tabellarisch dokumentieren. Es sollen alle Seitenzugriffe seit dem Laden einer Seite berücksichtigt werden.

a. FIFO (First In, First Out)

b. OPT (Optimalstrategie). Was macht die Ausführung der OPT-Strategie überhaupt erst möglich?

Verwenden Sie dazu folgendes Schema:

Zeitpunkt	Referenzierte Seite	f_0, Messwert	f_1, Messwert	f_2, Messwert	Summe Seitenfehler
...	...	...	...	...	...

– Die Spalte *Referenzierte Seiten* gibt die Seite an, auf die gerade zugegriffen wird.

– Die Spalten f_0 – f_2 enthalten die Seitennummer der aktuell im entsprechenden Frame gespeicherten Seite.

– Dokumentieren Sie zusätzlich für jede Seitemersetzungsstrategie relevante Werte, die für deren Ausführung notwendig sind, z.B. Anzahl der Zugriffe oder Zeitspanne bis zum nächsten Zugriff.

– Die Spalte *Summe Seitenfehler* enthält die aktuelle Gesamtanzahl an Seitenfehlern.

– **Achtung:** Bereits in den Hauptspeicher geladene Seiten dürfen nicht von einem Seitenrahmen in einen anderen verschoben werden!

Aufgabe K40: Buddy-Systeme

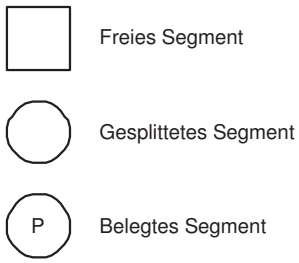
(– Pkt.)

Gegeben sei ein mobiles System mit einem Hauptspeicher von 1GB = 1024 MB, der byteweise adressiert wird. Zur Speicherverwaltung werden Buddy-Systeme eingesetzt. Dabei wird immer die am weitesten links stehende Speicherzelle geteilt, wenn ein neuer Prozess eingefügt wird. Die minimale Buddy-Größe soll 64MB betragen. Bearbeiten Sie nun folgende Aufgaben:

a. Es werden 4 Prozesse der Reihe nach in das Buddy-System eingefügt:

(i) p_1: 17 MB
(ii) p_2: 70 MB
(iii) p_3: 80 MB
(iv) p_4: 32 MB

Zeichnen Sie insgesamt 4 Buddy-Bäume, die jeweils den Zustand der Speicherbelegung darstellen, nachdem ein weiterer der 4 Prozesse eingefügt wurde. Es muss genau ersichtlich sein, ob es sich um ein freies, ein gesplittetes bzw. um ein belegtes Segmente des Hauptspeichers handelt. Kennzeichnen Sie die entsprechenden Segmente eines Buddy-Baums mit den folgenden Symbolen:



b. Wieviele Bits werden zur Adressierung eines Bytes im gegebenen Speicher benötigt? Ergänzen Sie im letzten Buddy-Baum der Teilaufgabe a (also dem Buddy-Baum nach dem Einfügen von p_4) für alle freien, belegten und gesplitteten Segmente die ersten 5 Bits ihrer Speicheradresse.

c. Geben Sie für jeden der vier Prozesse an, wie viel Speicher diesem entsprechend der Zuordnung nach dem Buddy-Verfahren tatsächlich zur Verfügung stünde.

d. Die eingefügten Prozesse p_1 bis p_4 benötigen insgesamt 199 MB. Durch die Verwendung des Buddy-Baums unterscheidet sich der tatsächlich zugeordnete Hauptspeicher jedoch von diesem Wert. Wieviel von den ursprünglichen 1024 MB stehen für weitere Prozesse somit nur noch zur Verfügung und wie viel Speicher wird letztendlich verschwendet? Wie nennt man den für diese Verschwendung verantwortlichen Effekt?

e. Ein neuer Prozess p_5 mit einem Speicherbedarf von 520 MB soll gestartet werden. Ist es möglich diesen Prozess einem Buddy zuzuweisen? Falls ja, zeichnen sie den aktualisierten Buddy-Baum. Falls nein, erläutern Sie den Grund.

f. Zunächst terminieren Prozess p_1 und dann Prozess p_2. Zeichen Sie den aktualisierten Buddy-Baum nach jeder der beiden Prozessterminierungen. Kennzeichnen Sie dabei freie, gesplitteten bzw. belegten Segmente wieder mit den zuvor verwendeten Symbolen.

Aufgabe K41: Koordination von Threads

(– Pkt.)

In dieser Aufgabe soll das bekannte Philosophenproblem (Dining Philosophers) mit Hilfe der Programmiersprache Java implementiert werden.
Beim Philosophenproblem sitzen fünf Philosophen an einem runden Tisch beim Essen. Vor jedem Philosophen befindet sich ein Teller voller Reis und zwischen je zwei Tellern befindet sich ein Stäbchen. Ein Philosoph, der hungrig ist, benötigt sowohl das linke als auch das rechte Stäbchen, um essen zu können. Hat ein Philosoph zwei Stäbchen, so isst dieser, bis er satt ist, erst dann legt er die Stäbchen an die ursprünglichen Plätze zurück, so dass seine Nachbarn davon Gebrauch machen können.

a. Beschreiben Sie, ab wann man sich beim Philosophenproblem im kritischen Bereiche befindet und ab wann man diesen wieder verlässt.

b. Wenn man beim Philosophenproblem keine Einschränkung beim Zugriff auf die Stäbchen formuliert, kann es zu einem Deadlock kommen.

   (i) Beschreiben Sie informell einen Ablauf für das Philosophenproblem, der zu einem Deadlock führt.

   (ii) Geben Sie zwei der *effizientesten* (!) Möglichkeit an, wie sich die Deadlocksituation im speziellen Fall des Fünf-Philosophenproblems vermeiden lässt.

   (iii) Das Auftreten welcher der für einen Deadlock notwendigen Bedingungen wird durch ihre Lösungen ausgeschlossen?

c. Im Folgenden sollen Sie das Philosophenproblem in Form der drei Java Klassen `Philosoph`, `Staebchen` und `Dinner` implementieren. Der Quelltext der Klasse `Dinner` ist bereits vorgegeben und sieht folgendermaßen aus:

```
1  public class Dinner {
2     public static void main(String[] args) {
3        Thread[] philosoph = new Thread[5];
4        Staebchen staebchen[] = new Staebchen[5];
5
6        for (int i = 0; i < 5; ++i) {
7           staebchen[i] = new Staebchen();
8        }
9        for (int i = 0; i < 5; ++i) {
10          philosoph[i] =
11             new Philosoph(i, staebchen[(i+4)%5], staebchen[i]);
12          philosoph[i].start();
13       }
14    }
15 }
```

(i) Implementieren Sie zunächst die Klasse `Staebchen`. Diese soll die beiden Methoden `get()` (Stäbchen vom Tisch nehmen) und `put()` (Stäbchen an den ursprünglichen Platz zurücklegen) zur Verfügung stellen. Verwenden Sie dazu den nachfolgenden Code-Rahmen. Achten Sie darauf, dass der Zugriff auf ein Stäbchen im wechselseitigen Ausschluss erfolgt und keine Race Conditions auftreten können! Verwenden Sie für Ihre Implementierung das vorgegebene Attribut `wirdBenutzt`.

Kommentieren Sie Ihre Lösung ausführlich!

Code-Rahmen der Klasse `Staebchen`

```
1  public class Staebchen {
2
3      // Attribut
4      private boolean wirdBenutzt = false;
5
6      // get()-Methode
7      // ...
8      // put()-Methode
9      // ...
10 }
```

(ii) Implementieren Sie nun die Klasse `Philosoph` Achten Sie darauf, dass Ihre Implementierung frei von Deadlocks ist. Verwenden Sie dazu den nachfolgenden Code-Rahmen (Fortsetzung auf der nächsten Seite!). Fügen Sie Ihrer Implementierung alle Attribute hinzu, die notwendig sind, damit Ihre Implementierung sich mit der vorgegebenen Klasse `Dinner` fehlerfrei übersetzen lässt.

Kommentieren Sie Ihre Lösung ausführlich!

Hinweis: Die Klasse `Random` dient dazu, Zufallszahlen zu erzeugen. Die Methode `nextInt(int positiveZahl)` liefert eine Zahl zwischen 0 (inklusive) und `positiveZahl` (exklusive). Mit Hilfe dieser Methode wird die Zeit, die ein Philosoph isst bzw. denkt simuliert.

Code-Rahmen der Klasse `Philosoph`

```
1  import java.util.Random;
2
3  public class Philosoph extends Thread {
4      private static Random rand = new Random();
5
6      // Attribute
7      // ...
8      // Konstruktor
9      // ...
10     // run()-Methode
11     public void run() {
12         try {
13             while(true) {
14                 // denken
15                 System.out.println("Philosoph "+ id + " denkt");
16                 Thread.sleep(rand.nextInt(500) + 500);
17                 // hungrig (versuche Staebchen zu erhalten)
18                 // ...
19                 // essen
20                 System.out.println("Philosoph "+ id + " isst");
```

```
21              Thread.sleep(rand.nextInt(500) + 500);
22              // satt (lege Staebchen zurück)
23              // ...
24            }
25        }
26        catch(InterruptedException ie) {
27        }
28    }
29 }
```