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

mobile and distributed systems group

Betriebssysteme im Wintersemester 2018/2019
Übungsblatt 12

Abgabetermin: 28.01.2019, 18:00 Uhr

Besprechung: Besprechung der T-Aufgaben in den Tutorien vom 21. – 25. Januar 2019
Besprechung der H-Aufgaben in den Tutorien vom 28. Januar – 01. Februar 2019

Ankündigungen: Die **Klausur** findet am **7. Februar 2019 von 18.30 - 20.30 Uhr** statt. Bitte melden Sie sich **bis spätestens 28. Januar 2019, 12:00 Uhr** über UniWorX zur Klausur **an** bzw. **ab**.

Aufgabe 52: (T) Wechselseitiger Ausschluss

(– Pkt.)

In dieser Aufgabe soll das Szenario **zweier** Prozesse P_1 und P_2, die auf einen kritischen Abschnitt zugreifen, zu einem Erzeuger/Verbraucher-Szenario erweitert werden.
Der Prozess P_1 erzeugt Daten und schreibt diese in einen gemeinsamen Speicher mit 5 Plätzen. Der Prozess P_2 liest diese Daten. Dazu werden die beiden Zählsemaphoren `platz` und `bestand` eingeführt und wie folgt initialisiert:

```
Semaphore platz, bestand;
init(platz, 5);
init(bestand, 0);
```

Ergänzen Sie die folgenden Prozessdefinitionen unter Verwendung dieser Semaphoren so, dass P_1 nicht in den vollen Speicher schreiben und P_2 nicht aus einem leeren Speicher lesen kann.

```
P1:                                   P2:
<rechne im unkritischen Bereich>      <rechne im unkritischen Bereich>


wait(mutex);                          wait(mutex);


<erzeuge Element>                     <lies Element>


signal(mutex);                        signal(mutex);


<rechne im unkritischen Bereich>      <rechne im unkritischen Bereich>
```

Aufgabe 53: (T) Nachbildung von Zählsemaphoren mittels Monitoren in Java

(– Pkt.)

Schreiben Sie eine Java Klasse `SimpleCountingSemaphore`, die es ermöglicht, Instanzen zu erzeugen, welche in Analogie zu der von Dijkstra vorgeschlagenen Datenstruktur eines Zählsemaphor verwendet werden können. Verwenden Sie dazu den Java `synchronized`-Mechanismus. Verwenden Sie außerdem eine *minimale* Anzahl an `wait()`- und `notify()`-Aufrufen!

Hinweis: Für die Lösung dieser Aufgabe reicht es aus, wenn Sie das auftreten einer etwaigen `InterruptedException` ohne weitere Fehlerbehandlung abfangen.

Des Weiteren sollen Sie sich in dieser Aufgabe die Unterschiede und Gemeinsamkeiten zwischen Java-Synchronisation und Monitoren klar machen.

a. Beschreiben Sie das grundlegende Konzept der Synchronisation in Java. Gehen Sie dabei auf die Verwendung von Objekt-Locks, Threads und Warteschlangen ein.

b. Wie werden `wait()` und `signal()` in Java umgesetzt?

c. Erläutern Sie den grundlegenden Unterschied des Java Synchronisationsmechnismus im Gegensatz zu „echten“ Monitoren (ohne Beachtung der Signalisierungsmechanismen).

d. Beschreiben Sie die Einschränkungen bei der Verwendung von `wait()` und `notify()` gegenüber „echten“ Monitoren.

   Hinweis: Überlegen Sie sich dazu, wie im Falle echter Monitore bzw. im Falle des Java Synchronisierungsmechanismus der nächste aktive Thread selektiert wird.

e. Überlegen Sie sich, wie diese Einschränkungen umgangen werden können.

f. Es gibt zwei Modelle, wie die Ausführung in einem Monitor nach dem Aufruf von `signal()` fortfährt (A: signalisierender Prozess, B: aufgeweckter Prozeß):

   – Signal-and-wait: A muß nach seiner Signalisierung den Monitor sofort freigeben und warten, bis B den Monitor verlassen hat oder auf eine andere Bedingung wartet.

   – Signal-and-continue: B muß warten, bis A den Monitor verlassen hat oder auf eine andere Bedingung wartet.

   Welchem Modell folgt Java?

Aufgabe 54: (T) Leser-/Schreiberproblem in Java

(– Pkt.)

In dieser Aufgabe soll die schreiberfreundliche Variante des Leser-/Schreiberproblems in Java umgesetzt werden. Laden Sie sich bitte dazu zunächst von der Betriebssysteme-Homepage die Dateien `Vaterprozess.java`, `Prozess.java` und `Speicher.java` herunter.
Das Problem ist folgendermaßen definiert:

– Es gibt einen gemeinsamen Speicher *S*.

– Es gibt n Prozesse, die jeweils entweder *lesend* oder *schreibend* auf den gemeinsamen Speicher *S* zugreifen dürfen.

– Prozesse sind entweder Leserprozesse oder Schreiberprozesse.

– Schreiberprozesse müssen vor dem Schreiben vom Speicher *S* ein Schreibrecht erhalten. Leserprozesse benötigen ebenfalls ein entsprechendes Leserecht bevor sie aus dem Speicher *S* lesen dürfen.

– Sowohl Leser- als auch Schreiberprozesse geben das jeweilige Lese- bzw. Schreiberecht nach Abschluss des Lesens- bzw. Schreibens wieder frei.

– Besitzt ein Schreiberprozess das Schreiberecht, so darf währenddessen kein Leserprozess das Leserecht besitzen.

- Zu jedem Zeitpunkt darf nur maximal ein Schreiberprozess das Schreiberecht besitzen.
- Fragt ein Schreiberprozess das Schreiberecht an, so dürfen ab diesem Zeitpunkt keine Leserprozesse mehr mit dem Lesen beginnen.
- Die Prozesse werden von einem Vaterprozess erzeugt.

Im Folgenden soll die noch unvollständige Klasse `Speicher` aus der Datei `Speicher.java` fertig implementiert werden. Die Prozesse und der erzeugende Vaterprozess werden ebenfalls durch Java–Klassen und Threads simuliert. Die Beispielimplementierungen der Klassen `Vaterprozess` und `Prozess` soll verdeutlichen, wie die Klasse `Speicher` verwendet werden kann.
Bearbeiten Sie nun die folgenden Teilaufgaben unter der Berücksichtigung des oben definierten schreiberfreundlichen Leser-/Schreiberproblems:

a. Was versteht man allgemein unter einem kritischen Bereich?

b. Implementieren Sie den Konstruktor der Klasse `Speicher`. Ergänzen Sie dabei den Coderahmen aus der Datei `Speicher.java` und *kommentieren Sie Ihre Lösung ausführlich*! Achten Sie insbesondere auf die korrekte Intialisierung der bereits deklarierten Klassenattribute.

c. Implementieren Sie die Methode `leserechte_holen(int prozess_id)`, welche von Leserprozessen aufgerufen wird, um Leserechte zu erhalten. Implementieren Sie außerdem die Methode `leserechte_freigeben(int prozess_id)`, welche nach abgeschlossenem Lesevorgang aufgerufen wird, um die Leserechte wieder freizugeben. Ergänzen Sie dabei den Coderahmen aus der Datei `Speicher.java` und *kommentieren Sie Ihre Lösung ausführlich!*
Hinweis: Sie können davon ausgehen, dass die Methoden `leserechte_holen(int prozess_id)` bzw. `leserechte_freigeben(int prozess_id)` immer in einer sinnvollen Reihenfolge aufgerufen werden (siehe Beispielimplementierung der Klasse `Prozess`).

d. Implementieren Sie die Methode `schreibrecht_holen(int prozess_id)`, welche von Schreiberprozessen aufgerufen wird, um Schreibrechte zu erhalten. Implementieren Sie außerdem die Methode `schreibrecht_freigeben(int prozess_id)`, welche nach abgeschlossenem Schreibvorgang aufgerufen wird, um das Schreibrecht wieder freizugeben. Ergänzen Sie dabei den Coderahmen aus der Datei `Speicher.java` und *kommentieren Sie Ihre Lösung ausführlich!*
Hinweis: Sie können davon ausgehen, dass die Methoden `schreibrecht_holen(int prozess_id)` bzw. `schreibrecht_freigeben(int prozess_id)` immer in einer sinnvollen Reihenfolge aufgerufen werden (siehe Beispielimplementierung der Klasse `Prozess`).

e. Zeigen Sie zwei kritische Bereiche in ihrem Programm auf. Wie wird hier sichergestellt, dass die Bedingung der Mutual Exclusion erfüllt ist?

Aufgabe 55: (H) Buddy-Systeme

(14 Pkt.)

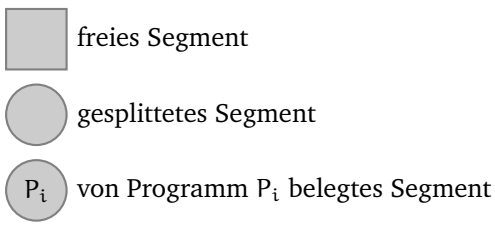
Ein mobiles Gerät verfüge über einen 256 MByte großen Speicher, der nach dem Buddy-Verfahren verwaltet wird. Die minimale Buddygröße soll 8 MByte betragen.

a. Wie viele Bits benötigt man mindestens, um diesen Speicher byteweise zu adressieren?

b. Nacheinander sollen nun die folgenden vier Programme in den Speicher geladen werden:

- P_1: 10 MByte
- P_2: 50 MByte
- P_3: 60 MByte
- P_4: 10 MByte

Zeichnen Sie den Buddy-Baum jedesmal, nachdem eines der Programme in den Speicher geladen wurde. Verwenden Sie dabei die Notation auf der rechten Seite:



Tragen Sie neben jedem freien Segment die Größe des freien Speicherbereichs an. Tragen Sie neben jedem belegten Segment die Größe des allokierten Speicherbereichs sowie die Speicheradressen des Segments an.

Achtung: Es wird immer versucht, das am weitesten links stehende Segment zu splitten und den am weitesten links stehenden Buddy zu belegen.

c. Die Programme aus Teilaufgabe b) benötigen insgesamt 130 MByte Speicherplatz. Damit müssten noch 256 - 130 = 126 MByte Speicher nutzbar sein. Warum ist das im Beispiel nicht der Fall? Welcher Effekt kommt hier zum Tragen? Wie viel nutzbarer Speicherplatz steht für weitere Programme insgesamt noch zur Verfügung?

d. Gegeben ist eine weitere Anfrage:

- P_5: 95 MByte

Kann P_5 noch zusätzlich in den Speicher geladen werden? Falls ja, zeichnen Sie den Buddy-Baum nach der Belege-Operation. Falls nein, begründen Sie Ihre Antwort.

e. Zunächst terminieren Prozess P_4 und dann Prozess P_1. Geben Sie den aktualisierten Buddy-Baum nach jeder der zwei Prozessterminierungen an. Achten Sie hierbei insbesondere wieder auf eine deutliche Unterscheidung von freien, gesplitteten und belegten Segmente.

Aufgabe 56: (H) Einfachauswahlaufgabe: Prozesskoordination und Speicher

(5 Pkt.)

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

a) Wie betritt ein Prozess einen Monitor?			
(i) Durch den Zugriff auf **public** Variablen.	(ii) Durch den Zugriff auf bestimmte Monitorprozeduren.	(iii) Durch den Aufruf von *notify()*.	(iv) Durch den Aufruf von *notifyAll()*.
b) Welches Kommunikationsschema liegt vor, wenn der Sender nach dem Absenden mit seiner Ausführung fortfahren kann aber der Empfänger ist bis zum Erhalt einer Nachricht wartet.			
(i) Blocking Send, Blocking Receive.	(ii) Blocking Send, Nonblocking Receive.	(iii) Nonblocking Send, Blocking Receive.	(iv) Nonblocking Send, Nonblocking Receive.
c) Welcher der folgenden Betriebssystemmechanismen dient vorrangig der Prozesssynchronisation und weniger der Prozesskommunikation?			
(i) Message Queues	(ii) Shared Memory	(iii) Semaphore	(iv) Pipes
d) Wie bezeichnet man die Thread-Implementierung, bei der das Threadmanagement Aufgabe der jeweiligen Anwendung ist und der Betriebssystemkern keine Informationen über die Existenz solcher Threads hat bzw. haben muss?			
(i) User-Level-Threads	(ii) Kernel-Level-Threads	(iii) Hardware-Level-Threads	(iv) Low-Level-Threads
e) Was ist keine Aufgabe der Speicherverwaltung?			
(i) Relocation	(ii) Destruction	(iii) Sharing	(iv) Protection