VO 182.022
Prüfung Betriebssysteme

27. April 2006

KNr. | MNr. | Zuname, Vorname

Ges.)(100)	1.)(30)	2.)(25)	3.)(25)	4.)(20)	Zusatzblätter:

Bitte verwenden Sie nur dokumentenechtes Schreibmaterial!

1 Synchronisation (30)

Beim Kartenspiel "Jag the Gitter" erhält jeder *Mitspieler* vom Kartengeber *5 Karten* verdeckt vor sich auf den Tisch gelegt (Funktion `karte_geben(`*Spieler*`)`). Nachdem der Kartengeber (welcher selbst **nicht** am Spiel teilnimmt) die Karten **reihum einzeln** ausgeteilt hat, nehmen alle Spieler **gleichzeitig** ihre Karten auf (Funktion `karten_nehmen()`).

Nachdem alle Mitspieler ihre Karten aufgenommen haben, wird rundenweise jeweils eine Karte pro Mitspieler (Funktion `karte_spielen()`) gespielt. Dabei beginnt jeweils Spieler 1 mit dem Ausspielen, dann kommt Spieler 2 dran usw. Nachdem alle Karten ausgespielt wurden ist das Spiel beendet und der Kartengeber teilt erneut Karten für die nächste Runde aus.

Zu verwendende Funktionen (alle Funktionen blockieren so lange, bis der angegebene Task erledigt ist):

`karte_geben(`*Spieler*`)` teilt eine Karte an den angegebenen *Spieler* ($\in \{1, 2, 3, ...\}$) aus. Diese Funktion wird nur vom Kartengeber verwendet.

`karten_nehmen()` lässt den Spieler alle seine Karten aufnehmen. Diese Funktion darf erst aufgerufen werden, wenn alle Karten dieses Spielers ausgeteilt sind.

`karte_spielen()` lässt den Spieler eine seiner Karten ausspielen.

`initS(`*semname*`,`*n*`)`, `initE(`*evcname*`,`*n*`)` zum Anlegen und Initialisieren von Semaphoren und Eventcountern

`P(),V(),advance(),wait()` mit der üblichen Semantik für Semaphore und Eventcounter.

Weiters zu beachten:

- Die Verwendung von globalen Variablen zur Synchronisation ist verboten!
- Achten Sie auf maximale Parallelität!
- Verwenden Sie (unter Berücksichtigung der oberen beiden Punkte) nur so viele Ressourcen wie notwendig!

a) (14)

Zuerst soll das Kartenspiel für *2 Spieler* implementiert werden. Synchronisieren Sie die folgenden Prozesse (Kartengeber, Spieler 1 und Spieler 2) mit **Semaphoren**, sodass Sie das Spiel "Jag the Gitter" wie oben beschrieben spielen.

Initialisierungen		
Kartengeber	**Spieler 1**	**Spieler 2**
`FOREVER()`	`FOREVER()`	`FOREVER()`
`{`	`{`	`{`
`}`	`}`	`}`

c) (16)

Nun soll das Kartenspiel auf eine beliebige Anzahl N von Spielern erweitert werden (N ist konstant). Ergänzen Sie im folgenden Gerüst die Aufrufe von `await()`, um die Prozesse (Kartengeber, Spieler i) mit **einem Eventcounter** derartig zu synchronisieren, sodaß Sie das Spiel "Jag the Gitter" wie oben beschrieben spielen. Im Spielerprozess steht ihnen die entsprechende Nummer des Spielers (zwischen 1 und N) in der Variablen i zur Verfügung.

Tipp: Sie brauchen zur Lösung dieses Beispiel *keine* Sequencer!

Initialisierungen

Kartengeber	**Spieler** i
`FOREVER()`	`FOREVER()`
`{`	`{`
`}`	`}`

2 Scheduling (25)

Rate Monotonic Scheduling (9)

Schedulen Sie das nebenstehende Taskset nach dem Rate-Monotonic Verfahren. Alle Tasks sind periodisch, wobei die Deadlines mit dem Ende der jeweiligen Periode gleichzusetzen sind. Der Overhead für den Taskwechsel ist vernachlässigbar. Gehen Sie davon aus, dass alle Tasks die erste Arrival Time 0 haben.

Task	Ausführungszeit	Periodendauer
A	1	4
B	2	15
C	1	9
D	3	11
E	2	13

Tragen Sie in der Zeile **P** jenen Task ein, der für die entsprechende Zeiteinheit dem Prozessor zugeteilt wird.

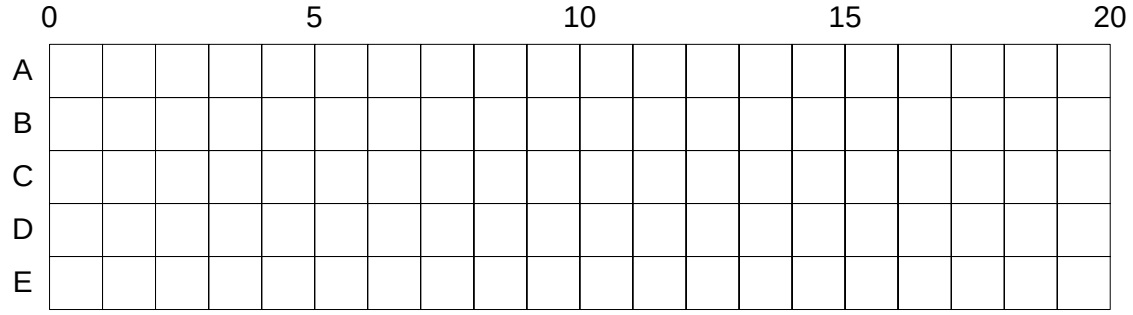

Abbildung 1: Round Robin

Kann es bei diesem Taskset zu einem Deadline-Miss kommen? Begründen Sie Ihre Antwort!

Round-Robin (8)

Schedulen Sie das nebenstehende Taskset. Beachten Sie dabei folgendes: Zu den angegebenen arrival times wird ein Task in die Ready Queue gestellt. Der Task kann frühestens beim nächsten diskreten Zeitpunkt dem Prozessor zugeteilt werden (Beispiel siehe Task A: arrival time= 0; Zuteilung = 1). Ein eintreffender Task wird immer ans Ende der Ready Queue gestellt.

Task	Arrival Time	Service Time
A	0	5
B	1	3
C	4	2
D	5	7
E	7	2

Tragen Sie in der Zeile **Exec.** jenen Task ein der für die entsprechende Zeiteinheit dem Prozessor zugeteilt wird. Der restliche Raster stellt die Ready Queue dar. Tragen Sie hier jene Tasks ein die in der/den Ready Queue(s) stehen (beginnend in der obersten Zeile mit

dem Task der als nächstes den Prozessor zugeteilt bekommt. Schedulen Sie das Task Set nach der Round-Robin Methode (time-slice = 1). Der Overhead für den Taskwechsel ist vernachlässigbar.

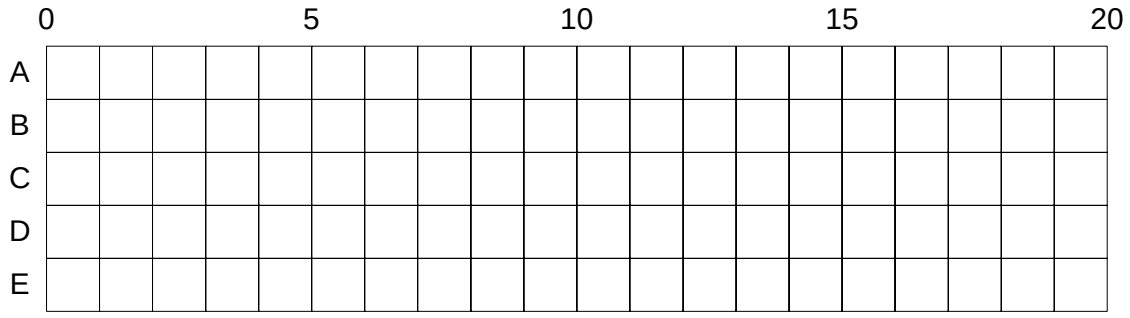

Abbildung 2: Round Robin Scheduling

Verständnisfragen (8)

Was bedeutet Starvation?

Geben Sie 3 Scheduling Methoden an bei denen es zu Starvation kommen kann:

Erklären Sie das **gemeinsame** Problem dieser 3 Methoden, wieso es zu Starvation kommen kann:

3 Deadlock (25)

Ein Computer-Server besitzt 5 Festplatten, 2 Speichermodule mit je 10 GByte und 3 Gigabit Ethernet Netzwerkkarten. Die Ressourcen, die das installierte Betriebssystem benötigt, brauchen Sie zur Lösung der folgenden Aufgaben nicht berücksichtigen.

Um die auf dem Server befindlichen Programme schedulen zu können, müssen die Ressourcen wie Festplatten (F), Speicher (S) und Netzwerkkarten (N) koordiniert werden.

Zurzeit befinden sich drei Programme am Server:

- **Programm 1** benötigt 4 Festplatten, 20 GByte Speicher und 2 Netzwerkkarten. Für dieses Programm ist bereits 1 Festplatte reserviert.
- **Programm 2** benötigt 4 Festplatten, 20 GByte Speicher und keine Netzwerkkarte. Für dieses Programm sind bereits 1 Festplatte und 10 GByte Speicher reserviert.
- **Programm 3** benötigt 2 Festplatten, 10 GByte Speicher und 3 Netzwerkkarten. Für dieses Programm sind bereits 1 Festplatte, 10 GByte Speicher und 1 Netzwerkkarte reserviert.

Process Initiation Denial (10)

Beschreiben Sie *Resource-* und *Available-Vektor* sowie *Claim-* und *Allocation-Matrix*. Die Matrizen sind so auszufüllen, dass die Ressourcen zeilenweise und die Programme (Prozesse) spaltenweise aufgezählt werden.

$$Resource = \begin{pmatrix} \mathtt{F} & \\ \mathtt{S} & \\ \mathtt{N} & \end{pmatrix} \qquad Available = \begin{pmatrix} & \\ & \\ & \end{pmatrix}$$

$$Claim = \begin{pmatrix} \mathtt{F} & & & \\ \mathtt{S} & & & \\ \mathtt{N} & & & \end{pmatrix} \qquad Allocation = \begin{pmatrix} & & \\ & & \\ & & \end{pmatrix}$$

Darf mit Programm 2 gemäß *Process Initiation Denial* begonnen werden?

◯ Ja ◯ Nein

Begründen Sie Ihre Antwort (Antworten ohne Begründung werden nicht gewertet!):

Bedingungen für Deadlock (5)

Erklären Sie die für das Auftreten eines Deadlocks notwendigen und hinreichenden Bedingungen.

Deadlock Vermeidung (10)

Verwenden Sie nun von obigem Initialzustand ausgehend den *Bankier-Algorithmus* (Banker's Algorithm) zum Schedulen der Prozesse. Geben Sie für *jeden* Schritt die Claim- und Allocation-Matrix, sowie den Availability Vector und das als nächstes durchzuführende Programm an. Wenn kein Programm mehr auszuführen ist, dann schreiben sie 'fertig', falls ein Deadlock auftritt, schreiben Sie 'Deadlock' in den nächsten Schritt.

Nächstes Programm: . Alle für das Programm benötigten Ressourcen sind in Verwendung:

$$Claim = \begin{pmatrix} \text{F} & & & \\ \text{S} & & & \\ \text{N} & & & \end{pmatrix} \quad Alloc = \begin{pmatrix} & & \\ & & \\ & & \end{pmatrix} \quad Avail = \begin{pmatrix} \\ \\ \end{pmatrix}$$

Nach der Ausführung des zuletzt markierten Programms:

$$Claim = \begin{pmatrix} \text{F} & & & \\ \text{S} & & & \\ \text{N} & & & \end{pmatrix} \quad Alloc = \begin{pmatrix} & & \\ & & \\ & & \end{pmatrix} \quad Avail = \begin{pmatrix} \\ \\ \end{pmatrix}$$

Nächstes Programm: . Alle für das Programm benötigten Ressourcen sind in Verwendung:

$$Claim = \begin{pmatrix} \text{F} & & & \\ \text{S} & & & \\ \text{N} & & & \end{pmatrix} \quad Alloc = \begin{pmatrix} & & \\ & & \\ & & \end{pmatrix} \quad Avail = \begin{pmatrix} \\ \\ \end{pmatrix}$$

Nach der Ausführung des zuletzt markierten Programms:

$$Claim = \begin{pmatrix} \text{F} & & & \\ \text{S} & & & \\ \text{N} & & & \end{pmatrix} \quad Alloc = \begin{pmatrix} & & \\ & & \\ & & \end{pmatrix} \quad Avail = \begin{pmatrix} \\ \\ \end{pmatrix}$$

Nächstes Programm: . Alle für das Programm benötigten Ressourcen sind in Verwendung:

$$Claim = \begin{pmatrix} \texttt{F} & & & \\ \texttt{S} & & & \\ \texttt{N} & & & \end{pmatrix} \quad Alloc = \begin{pmatrix} & & \\ & & \\ & & \end{pmatrix} \quad Avail = \begin{pmatrix} \\ \\ \end{pmatrix}$$

Nach der Ausführung des zuletzt markierten Programms:

$$Claim = \begin{pmatrix} \texttt{F} & & & \\ \texttt{S} & & & \\ \texttt{N} & & & \end{pmatrix} \quad Alloc = \begin{pmatrix} & & \\ & & \\ & & \end{pmatrix} \quad Avail = \begin{pmatrix} \\ \\ \end{pmatrix}$$

Nächstes Programm: . Alle für das Programm benötigten Ressourcen sind in Verwendung:

$$Claim = \begin{pmatrix} \texttt{F} & & & \\ \texttt{S} & & & \\ \texttt{N} & & & \end{pmatrix} \quad Alloc = \begin{pmatrix} & & \\ & & \\ & & \end{pmatrix} \quad Avail = \begin{pmatrix} \\ \\ \end{pmatrix}$$

Nach der Ausführung des zuletzt markierten Programms:

$$Claim = \begin{pmatrix} \texttt{F} & & & \\ \texttt{S} & & & \\ \texttt{N} & & & \end{pmatrix} \quad Alloc = \begin{pmatrix} & & \\ & & \\ & & \end{pmatrix} \quad Avail = \begin{pmatrix} \\ \\ \end{pmatrix}$$

4 Security (20)

4.1 Begriffe (8)

Was versteht man unter *Confidentiality (Secrecy)*, *Integrity* und *Availability*? Erklären Sie die drei Begriffe.

- Confidentiality:

- Integrity:

- Availability:

Arten der Bedrohung (Types of Threats)

Füllen Sie folgende Tabelle derart aus, dass Sie jede angegebene Art der Bedrohung einem der Begriffe *Confidentiality (Secrecy)*, *Integrity* und *Availability* zuordnen (Fehlende Antworten werden negativ, falsche Antworten werden doppelt negativ gewertet!):

Art der Bedrohung	**bedroht**
Interruption	
Interception	
Modification	
Fabrication	

4.2 Bedrohungen durch Programme oder Programmfragmente (8)

Erläutern Sie die folgenden Bedrohungen: *Logic Bomb*, *Trojan Horse*, *Virus* und *Worm*.

- Logic Bomb:

- Trojan Horse:

- Virus:

- Worm:

4.3 Security by Obscurity vs. Open Design (4)

Beschreiben Sie den Unterschied zwischen *Security by Obscurity* und *Open Design*. Geben Sie die Vor- und Nachteile an.