VO 182.022 18. Jänner 2005

Prüfung Betriebssysteme

KNr. MNr. Zuname, Vorname

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

Bitte verwenden Sie nur dokumentenechtes Schreibmaterial!

1 Scheduling (25)

1.1 Uniprocessor Scheduling (13)

Gegeben ist nebenstehendes Taskset. Alle Tasks sind periodisch, wobei die Deadlines mit dem Ende der jeweiligen Periode gleichzusetzen sind. Der Overhead für den Taskwechsel ist vernachlässigbar.

Task	Ausführungszeit	Periodendauer
A	1	5
B	2	10
C	2	7
D	1	6

Ermitteln Sie für dieses Taskset die *notwendige* und die *hinreichende* Bedingung für das *Rate Monotonic Scheduling* (RMS) Verfahren. Berechnen Sie die **Zahlenwerte** überschlagsmäßig (ohne Taschenrechner)!

Ist die notwendige Bedingung erfüllt? ◯ Ja ◯ Nein

Ist die hinreichende Bedingung erfüllt? ◯ Ja ◯ Nein

Versuchen Sie das Taskset einmal mit dem RMS und einmal mit dem *Earliest-Deadline-First* (EDF) Verfahren zu schedulen. Verwenden Sie dazu die nachstehenden Vorlagen. Tragen Sie bei jeder Vorlage die aktiven Taskzeiten ein und bezeichnen Sie deutlich eventuelle Deadlineverletzungen. Kreuzen Sie jeweils an ob das Scheduling erfolgreich war. Eine Vorlage dient als Ersatz, streichen Sie gegebenenfalls eine falsch ausgefüllte Vorlage deutlich durch.

Scheduling nach dem **RMS**-Verfahren: Erfolgreich: ◯ Ja ◯ Nein

A																		
B																		
C																		
D																		

0 5 10 15

Scheduling nach dem **EDF**-Verfahren: Erfolgreich: ◯ Ja ◯ Nein

A																		
B																		
C																		
D																		
	0					5					10					15		

Ersatzvorlage: Scheduling nach dem -Verfahren: Erfolgreich: ◯ Ja ◯ Nein

A																		
B																		
C																		
D																		
	0					5					10					15		

1.2 Verständnisfragen (12)

Beurteilen Sie die folgenden Aussagen für Single-Prozessor Scheduling!
Fehlende Antworten werden negativ, falsche Antworten werden doppelt negativ gewertet!

◯ Ja ◯ Nein Jedes Taskset, das mittels EDF gelöst werden kann, lässt auch mit RMS schedulen.

◯ Ja ◯ Nein Ein Taskset, das mittels EDF nicht gelöst werden kann, lässt eventuell mit RMS schedulen.

◯ Ja ◯ Nein Die Prioritäten beim EDF Scheduling ergeben sich durch die Periodendauer der Tasks.

◯ Ja ◯ Nein Bei Round Robin Scheduling kann das Problem der Starvation auftreten.

◯ Ja ◯ Nein Beim Scheduling nach dem Round-Robin-Verfahren werden I/O-intensive Prozesse benachteiligt.

◯ Ja ◯ Nein Wenn bei allen Tasks die Deadline gleich ihrer Periode ist, dann liefert RMS Scheduling dasselbe Ergebnis wie EDF Scheduling.

◯ Ja ◯ Nein Die First-Come-First-Serve-Strategie benachteiligt CPU-intensive Prozesse.

◯ Ja ◯ Nein Das Shortest-Remaining-Time-Verfahren begünstigt I/O-intensive Prozesse.

◯ Ja ◯ Nein Das Shortest-Process-Next-Verfahren ist für Echtzeitscheduling ungeeignet.

◯ Ja ◯ Nein Die Prioritäten beim RMS Scheduling ergeben sich durch die reziproke Periodendauer der Tasks.

◯ Ja ◯ Nein Das Shortest-Process-Next-Verfahren kommt im Vergleich zu Round Robin mit weniger Interruptaufrufen aus.

2 Synchronization (25)

Synchronisieren Sie die Tätigkeiten der Mechaniker während eines Boxenstopps in der Formel 1. Ein Boxenstopp läuft folgendermaßen ab:

- Der Pilot fährt in die Boxengasse (`pit_stop()`). Ein Teammitglied (Signalgeber) gibt via der Signaltafel welche auf 'Brake on' gedreht ist, die Anweisung das Fahrzeug in der Box des Teams zum Stillstand zu bringen (`brake()`). Der Fahrer darf erst wieder Gas geben (`accelerate()`), wenn die Mechaniker fertig sind und die Signaltafel auf 'GO' gestellt ist. Damit ist der Boxenstopp beendet.

- Ein Teammitglied (Signalgeber) signalisiert dem Fahrer mit der Tafel 'Brake on/GO' (`signal_brake()` und `signal_go()`), ab wann das Fahrzeug wieder bereit ist, die Box zu verlassen. Der Signalgeber dreht das Signal auf 'GO', sobald das alle vier Reifen wieder Bodenkontakt haben.

- Um einen Reifenwechsel zu ermöglichen, muss das Auto angehoben werden. Diese Aufgabe übernehmen zwei Teammitglieder, sobald das Auto still steht und der Signalgeber dies mittels 'Break on' anzeigt. Einer bockt das Auto vorne (`lift_front()`) und einer hinten (`lift_back()`) auf. Erst dann können die Reifenmechaniker, die Tankwarte und der Visierreiniger mit ihrer Arbeit beginnen. Sobald diese fertig sind und die Hand gehoben haben, kann das Fahrzeug wieder abgesenkt werden (`down_front()` und `down_back()`) damit alle vier Reifen wieder Bodenkontakt haben.

- Insgesamt 12 Teammitglieder stehen bereit zum Reifenwechsel, je drei davon für ein Rad. Ein Mechaniker löst (`unscrew_wheel_nut()`) und befestigt die Radmutter (`fasten_wheel_nut()`) und hebt den Arm sobald er das Rad fertig gewechselt ist (`ready()`). Ein andere Mechaniker zieht das abgefahrene Rad ab (`remove_old_wheel()`) und der dritte steckt das neue Rad auf (`put_new_wheel()`).

- Zwei Teammitglieder sind für das Betanken des Wagens zuständig (`fill_up()`). Sobald das Fahrzeug fertig betankt ist, wird dies durch das Heben einer Hand signalisiert (`ready()`). Damit kann das Fahrzeug aus Sicht der Tankwarte wieder abgesenkt werden.

- Ein Teammitglied (Visierreiniger) säubert das Helmvisier des Piloten (`clean_visor()`). Sobald das Helmvisier gereinigt ist, wird dies durch das Heben einer Hand signalisiert (`ready()`). Damit kann das Fahrzeug aus Sicht des Visierreinigers wieder abgesenkt werden.

Ergänzen Sie den folgenden Code entsprechend. Beachten Sie dabei folgende Punkte

- Die Synchronisation hat durch Semaphore zu erfolgen, welche in der Funktion `Init()` zu initialisieren sind. Dafür steht die Funktion `initsem(semaphor,value)` zur Verfügung. Zur Synchronisation sind `P(semaphor)` und `V(semaphor)` zu verwenden. Verwenden Sie eine minimale Anzahl von Ressourcen!
- Achten Sie auf maximale Parallelität, um den Boxenstopp so schnell wie möglich zu beenden!

Initialisierung:

```
Init() {

    initsem(signal, 0);
```



```
}
```

Der Fahrer:

```
Driver() {

    pit_stop();

    break();

    V(signal);
```



```
}
```

Mechaniker für den Reifenwechsel:

Obwohl diese Funktion viermal aufgerufen wird (für jedes Rad), braucht diese Funktion nur einmal programmiert werden. Diese Funktion implementiert die Tätigkeit von je drei Mechanikern pro Rad.

```
Mechanic_Wheel() {
```



```
}
```

Mechaniker zum Aufbocken des Fahrzeugs:

Diese Funktion implementiert die Tätigkeit der zwei Mechaniker, welche das Auto aufbocken (vorne und hinten).

```
Mechanic_Lifter() {
```



```
}
```

Der Visierreiniger:

```
Visor() {
```



```
}
```

Betanken des Wagens:

```
Fuel() {
```



```
}
```

Der Signalgeber:

```
Signal() {
    signal_break(); //wait for standing car
    P(signal);
```



```
}
```

3 Security (25)

Security Threats (6)

Beschreiben Sie die 3 grundsätzliche Arten von Security Threats? Geben Sie jeweils ein Beispiel an.

Verschlüsselung (4)

Beschreiben Sie das Prinzip des Public-Key Verschlüsselungsverfahrens, sowie die Vor- und Nachteile gegenüber symmetrischen Verschlüsselungsverfahren.

Prinzip:

Vorteile:

Nachteile:

Reference Monitor (2)

Was ist ein Reference Monitor?

Intrusion Detection (5)

Was ist *Intrusion Detection*, und wozu wird es verwendet?

Beschreiben Sie 3 *Intrusion Detection* Methoden, und geben Sie jeweils ein Beispiel an.

Authentification (2)

Was ist Authentification und wie wird das gemacht?

Verständnisfragen (6)

Beurteilen Sie die folgenden Aussagen! Fehlende Antworten werden nicht gewertet, falsche Antworten werden negativ gewertet!

◯ Ja ◯ Nein *Network security* umfasst Maßnahmen zum Schutz der Daten während der Übertragung.

◯ Ja ◯ Nein Bei Asymmetrischen Crypto-Systemen werden zum Ver- und Entschlüsseln dieselben Keys verwendet.

◯ Ja ◯ Nein *Threshold detection* ist ein statistisches Verfahren zur Intrusion Detection.

◯ Ja ◯ Nein Das *Least Privilege* Prinzip besagt, dass jeder Benutzer keine Rechte per default hat.

◯ Ja ◯ Nein Bei *Masquerading* wird der Inhalt einer Message verändert.

◯ Ja ◯ Nein Eine *denial-of-service* Attacke ist eine aktive Attacke.

4 Deadlock (25)

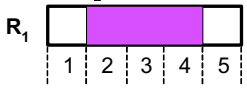
Gegeben sind zwei Prozesse, P_1 und P_2, die jeweils die Ressourcen R_1 und R_2 benötigen. Jede der zwei Ressourcen ist zweimal vorhanden. Benötigt ein Prozess eine vom anderen Prozess belegte Ressource, so wird er auf jeden Fall bis zum Freiwerden der Ressource verzögert.

Der Fortschritt von P_1 und P_2 bei der (quasi)parallelen Abarbeitung kann als Kantenzug zwischen den Punkten *start* und *end* in der Grafik eingetragen werden. Die Achsenbeschriftung entspricht dabei der Zeilennummer des gerade auszuführenden Befehles.

Unterhalb bzw. links der Diagrammachsen sind Balken vorgesehen, in denen die Anforderungen von Ressourcen für P_1 bzw. P_2 eingetragen werden.

1. Tragen Sie die Anforderungen von Ressourcen für P_1 bzw. P_2 unterhalb bzw. links der Diagrammachsen ein. Dabei ist anzunehmen, dass eine Roussource bereits ab Start der Anweisung `get()` als allokiert gilt und erst nach Beendigung der Anweisung `free()` als wieder freigegeben gilt, wie im folgenden Beispiel verdeutlicht ist:

   ```
   2: get(R1)
   3: ...
   4: free(R1)
   ```

   Sind mehrere Instanzen einer Ressource belegt, so geben Sie die zuletzt belegte Instanz frei.

2. Umranden und schraffieren Sie in der Grafik jene Bereiche, durch die der Kantenzug einer (quasi)parallelen Abarbeitung aufgrund von Ressourcenkonflikten nicht gehen kann.

3. Kennzeichnen Sie auf unterschiedliche Weise die Bereiche, die von einem Kantenzug nicht passiert werden dürfen, wenn eine Abarbeitung von P_1 und P_2 deadlockfrei erfolgen soll. Beschriften Sie diese Bereiche deutlich mit einem "D".

4. Zeichnen Sie einen Kantenzug für eine gültige, deadlockfreie Abarbeitung von P_1 und P_2 in der Grafik ein.

5. Beschriften Sie jeweils ein Kästchen im Diagramm

   mit 'A', von welchem aus der Punkt *end* erreichbar ist

   mit 'B', welches unweigerlich zu einen Deadlock führt

   mit 'C', welches ein nicht erreichbarer Zustand ist

Achten Sie bitte darauf, dass alle Lösungen gut erkennbar und die Lösungen zu den Teilaufgaben 2 und 3 *deutlich unterscheidbar* sind.

Program P_1:

```
 1: a=0;
 2: get(R2);
 3: a=b-6;
 4: get(R2);
 5: a=a*2;
 6: a=b*3;
 7: get(R1);
 8: free(R2);
 9: a=5;
10: free(R1);
11: c=c-2;
12: b=10;
13: free(R2);
14: return;
```

Program P_2:

```
 1: get(R1);
 2: a=10;
 3: b=a+12;
 4: get(R1);
 5: get(R2);
 6: get(R2);
 7: free(R1);
 8: c=b*2;
 9: free(R1);
10: free(R2);
11: d=c-2;
12: a=b;
13: free(R2);
14: return;
```