Matrikel-Nr.: Name: Studiengang:

Prüfungsklausur Betriebssysteme und Sicherheit am 24. Juli 2012

Alle Aussagen sind so ausführlich wie nötig, aber so knapp wie möglich zu begründen.

Bewertung 1: 2: 3: 4: 5: 6: Σ:

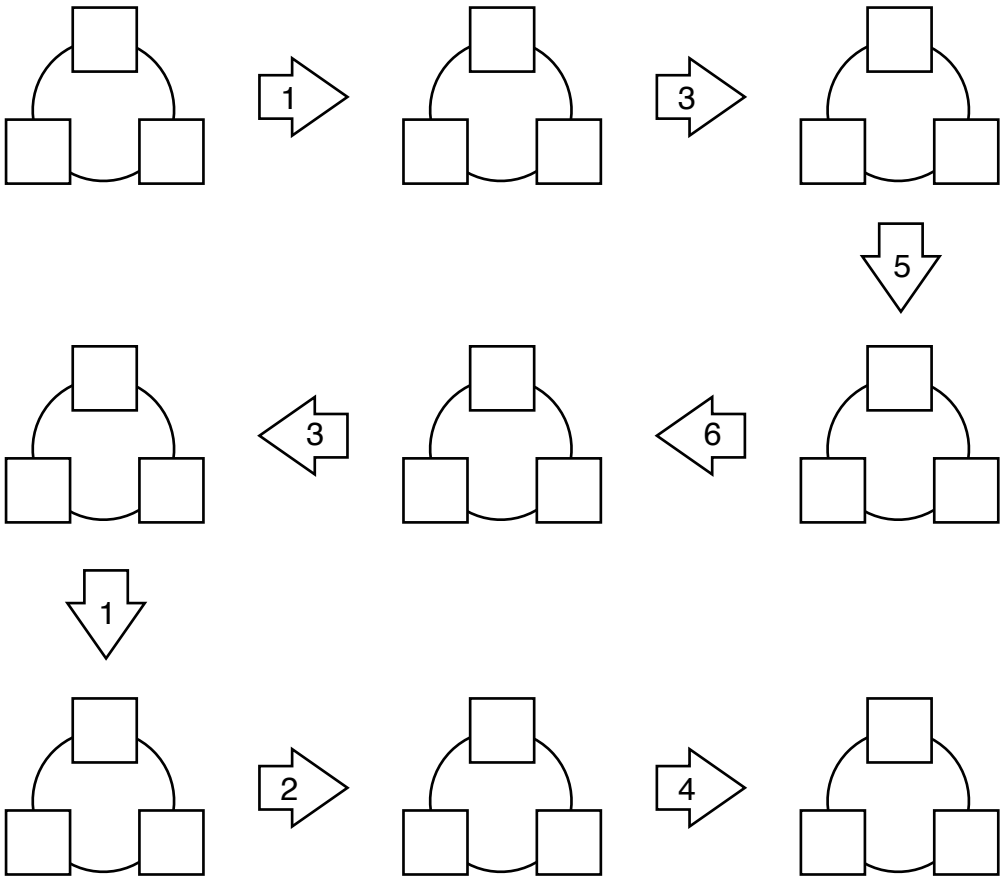
1. Aufgabe **9 Punkte**

Ein System hat 3 Kacheln und zwei Prozesse *A* und *B*. Geben Sie das Verhalten der Verdrängungsstrategien *Clock* und *Arbeitsmenge* für folgende Sequenz von Seitenzugriffen an:

Prozess:	A	A	B	B	A	A	A	A
Seite:	1	3	5	6	3	1	2	4

Der Arbeitsmengenparameter (Größe der Arbeitsmenge) betrage 2. Sobald der Hauptspeicher für die Arbeitsmengen beider Prozesse nicht ausreicht, wird derjenige Prozess komplett ausgelagert, der den zur Speicherknappheit führenden Seitenfehler auslöst.

Clock:



Arbeitsmenge:

	A	A	B	B	A	A	A	A
Kachel	1	3	5	6	3	1	2	4
1								
2								
3								

2. Aufgabe **4 Punkte**

Betrachtet wird ein System, dessen virtueller und physischer Adressraum jeweils mit 32 Bit breiten Adressen angesprochen wird. Die Seitengröße beträgt 4 KiB, die Seitentabelle ist zweistufig. Die einzelnen Tabellen benötigen jeweils 4 KiB Speicherplatz und enthalten jeweils 1024 Einträge.

Hinweis: Die Einheit KiB bezeichnet 1024 Byte.

a) Wie viele Einträge werden in beiden Stufen benötigt um den gesamten virtuellen Adressraum abzubilden?

b) Ein neuer Anwendungsprozess soll gestartet werden. Das Code-Segment der Anwendung ist 13 KiB groß, für das Datensegment sind 27 KiB erforderlich. Innerhalb des Datensegments soll die Anwendung sowohl lesen als auch schreiben können, im Code-Segment sollen keine Änderungen möglich sein. Das Betriebssystem verwaltet Speicherbelegungen nur in ganzen Seiten. Bei der Erzeugung des Anwendungsprozesses werden die Segmente zufällig im virtuellen Adressraum platziert.

   Zum Zeitpunkt des Starts der Anwendung verfügt das System über 14 Kacheln ungenutzten physischen Speicher. Kann die Anwendung in jedem Fall ausgeführt werden, ohne dass Seiten ausgelagert werden müssen? Berücksichtigen Sie den Speicherbedarf sowohl für Code und Daten als auch die zur Abbildung des Adressraums benötigten Seitentabellen.

3. Aufgabe **5 Punkte**

In einem System seien A, B und C Subjekte; x, y und z Objekte und G eine Gruppe bestehend aus den Subjekten A und B. Drücken Sie die folgenden durch Zugriffskontrolllisten (ACLs) gegebenen Berechtigungen mit Capabilities aus.

x: (A, {r,w}), (C, {r})
y: (A, {r,w}), (G, {r}), (C, {w})
z: (B, {r}), (C, {r})

Beschreiben Sie einen Nachteil der Abbildung von Gruppenrechten mit Capabilities.

4. Aufgabe **9 Punkte**

Betrachtet wird eine Listen-Datenstruktur und zwei darauf definierte Operationen. Der Listen-Typ `List` verwaltet Elemente vom Typ `Element`. Jede Instanz von `List` enthält außerdem ein internes Lock, welches genutzt wird um wechselseitigen Ausschluss bei parallel ausgeführten Listen-Operationen zu gewährleisten.

Es sind zwei Operationen definiert:

```
append(List l, Element e)
{
    lock(l.mutex);
    // Hängt Element e ans Ende
    // der Liste l an.
    unlock(l.mutex);
}
```

```
move(List a, List b)
{
    lock(a.mutex);
    lock(b.mutex);
    // Entfernt alle Elemente
    // aus Liste a und hängt
    // sie in derselben Reihenfolge
    // an Liste b an.
    unlock(b.mutex);
    unlock(a.mutex);
}
```

In einem Prozess existieren zwei Instanzen `L1` und `L2` vom Typ `List`, welche beide zu Beginn leer sind. Zwei Threads T_1 und T_2 führen jeweils genau einmal folgenden Code aus:

T₁
```
append(L1, 1);  // (1)
move(L1, L2);   // (2)
```

T₂
```
append(L2, 2);  // (3)
move(L2, L1);   // (4)
```

a) Wie kann es zu einer Verklemmung zwischen T_1 und T_2 kommen? Begründen Sie Ihre Antwort unter Verwendung der aus der Vorlesung bekannten Bedingungen zum Auftreten von Verklemmungen.

b) Welche Folgen von Elementen sind in den Listen `L1` und `L2` nach verklemmungsfreiem Ablauf von T_1 und T_2 möglich?

c) Verklemmungen sollen durch ein zusätzliches globales Lock `g` verhindert werden. Passen Sie die Funktionen `append()` und `move()` entsprechend an. Die parallele Abarbeitung soll dabei nicht unnötig eingeschränkt werden.

5. Aufgabe **6 Punkte**

In einem Echtzeitsystem ist eine Menge von drei periodischen Tasks so einzuplanen, dass deren Jobs in jeder Periode erfolgreich beendet werden. Das Periodenende entspricht der Deadline. Die Parameter der Tasks beschreiben jeweils die Periode p und die konstante Bearbeitungszeit e der Jobs:

T_1: $p_1 = 12$, $e_1 = 6$; T_2: $p_2 = 6$, $e_2 = 2$; T_3: $p_3 = 4$, $e_3 = 1$

Die Tasks sind unabhängig voneinander und an beliebiger Stelle unterbrechbar. Der Scheduling-Overhead werde vernachlässigt. Es stehe genau ein Prozessor zur Verfügung.

a) Gibt es ein Scheduling-Verfahren, welches die gegebene Taskmenge unter den genannten Bedingungen einplanen kann?

b) Während des Entwurfs des Echtzeitsystems wird festgestellt, dass die Periode der Task T_3 verändert werden muss. Welche Periode p_3 kann minimal verwendet werden, so dass die Taskmenge mit dynamischen Prioritäten einplanbar ist?

c) Wie muss die Periode p_3 verändert werden, wenn auch eine Einplanung mit statischen Prioritäten sicher gestellt werden soll?

d) Beim Scheduling nach dem Verfahren EDF (Earliest Deadline First) belegt stets der bereite Job mit der nächstgelegenen Deadline den einzigen Prozessor. Dieses Verfahren kann auf n Prozessoren erweitert werden, wobei stets die bereiten Jobs mit den n am nächsten gelegenen Deadlines die Prozessoren belegen. Alle weiteren Randbedingungen (Periodenende gleich Deadline, beliebige Unterbrechbarkeit, kein Scheduling-Overhead) werden unverändert von EDF übernommen.
Kann mit diesem Verfahren die ursprüngliche Taskmenge (*nicht* die nach Teilaufgabe b oder c veränderten Taskmengen) auf zwei Prozessoren eingeplant werden?

Sie können diesen Bereich zum Skizzieren von Ablaufplänen nutzen:

6. Aufgabe **9 Punkte**

In einem UNIX-System soll eine vorher noch nicht existierende Datei `numbers` angelegt und mit den Zahlen 0, 1, 2, 3, 4 in dieser Reihenfolge beschrieben werden. Das Schreiben der Datei soll auf mehrere Prozesse verteilt werden, wobei jeder Prozess nur genau eine Zahl schreiben darf.

Geben Sie in C-ähnlichem Pseudocode ein Programm an, welches durch Starten mehrerer Prozesse das obige Verhalten implementiert. Dabei dürfen sie folgende Funktionen benutzen:

fd = open(name)	öffnet die angegebene Datei und liefert einen File Descriptor als Ergebnis, die Datei wird angelegt, wenn sie noch nicht existiert
write_value(zahl, fd)	schreibt die Zahl gemäß der UNIX-Write-Semantik in die durch den File Descriptor gegebene Datei; diese Funktion darf von jedem Prozess nur einmal ausgeführt werden
pid = fork()	erzeugt nach der bekannten UNIX-Fork-Semantik einen Kind-Prozess
pid = wait()	wartet auf das Beenden eines beliebige Kind-Prozesses