Klausur WS 2009/2010

02.02.2010

Name:	_____	Matr.Nr.:	_____
Vorname:	_____	Studiengang:	_____

Hinweise: (Bitte sorgfältig durchlesen!)

- Schreiben Sie auf **jedes Blatt** Ihren Namen und Ihre Matrikelnummer.
- Die Aufgaben können in der Regel auf den Aufgabenblättern beantwortet werden. Tragen Sie Ihre Lösungen in die dafür vorgesehenen Felder ein. Reicht der Platz nicht aus, **ist für jede Aufgabe ein neues Blatt zu verwenden**.
- Am Ende der Klausur ist das Deckblatt zusammen mit den Aufgabenblättern und evtl. zusätzlich verwendeten Blättern wieder abzugeben.
- Die Bearbeitungszeit beträgt **120 Minuten**.
- Es sind **keine Hilfsmittel** erlaubt.
- Die Klausur umfasst insgesamt **120** Punkte. Zum Bestehen genügen **60** Punkte.

Mit meiner Unterschrift bestätige ich, dass ich die Hinweise zur Kenntnis genommen habe.

_____
Unterschrift

Punktespiegel:

Aufgabe	1	2	3	4	5	6	7	8	9	Σ
Erreichbare Punkte	10	10	12	11	14	15	15	19	14	120
Erreichte Punkte										

Lösung 1 (Multiple Choice) **(1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1) = 10 Punkte**

Beantworten Sie folgende Fragen durch Ankreuzen der richtigen Antwort. Für jede falsche Antwort wird ein Punkt abgezogen (es werden minimal 0 Punkte vergeben). Welche der folgenden Aussagen gelten?

a) Nach der Ausführung einer I/O-Operation informiert der Hardware-Controller über Interrupts *direkt* den Benutzerprozess über die abgeschlossene Operation. Die CPU wird hierbei nicht benötigt.

ja	nein
☐	■

Nein -> es wird die CPU informiert, die den zugehörigen Benutzerprozess fortsetzt.

b) Der Zugriff auf den Hauptspeicher eines Rechners ist etwa 5 Mal langsamer als der Zugriff auf den CPU-Cache.

ja	nein
■	☐

Ja -> vergleiche Vorlesungsfolien

c) Alle Threads innerhalb eines Prozesses teilen sich dessen Adressraum, lediglich die Deskriptoren für die geöffneten I/O-Geräte werden über Shared-Memory-Bereiche geschützt.

ja	nein
☐	■

Nein -> Alle Threads teilen sich auch die geöffneten Dateien/Geräte.

d) Ein MMU ("Memory Management Unit") setzt physikalische in logische Adressen um (z.B. durch Aufaddieren der physikalischen Adresse auf das Base Register des Prozesses).

ja	nein
☐	■

Nein -> logisch in physikalisch.

e) Beim Demand-Paging wird der Seitenaustausch nur im Falle eines Seitenfehlers durchgeführt.

ja	nein
■	☐

Ja -> so ist die Definition.

f) Der Peterson-Algorithmus zum wechselseitigen Ausschluss erlaubt nur dann zwei Prozesse im kritischen Bereich, wenn einer der Prozesse erheblich langsamer ist.

ja	nein
☐	■

Nein -> nur einer Prozess darf sich im kritischen Bereich aufhalten.

g) Sowohl symbolische Verweise als auch benannte Pipes werden in UNIX in einer Inode-Struktur verwaltet.

ja	nein
■	☐

Ja -> Beides sind Typen eines Inodes.

h) Das Hauptproblem bei der CPU-Scheduling-Strategie LIFO ist die Benachteiligung kurzlaufender Jobs durch Langläufer.

ja	nein
■	☐

Ja -> LIFO is nicht-preemptiv, daher müssen während der Bedienung eines Langläufers alle kurzlaufenden Jobs lange warten.

i) Wie beim Paging werden auch beim Swapping einzelne Seiten ausgelagert, jedoch in viel größeren und zusammenhängenden Blöcken.

ja	nein
☐	■

Nein -> Beim Swapping werden ganze Prozesse ausgelagert. Eine Aufteilung in Seiten findet nicht statt.

j) Neue Kindprozesse in UNIX müssen sich die CPU-Zeit ihres Vaterprozesses teilen und werden somit benachteiligt. Es empfiehlt sich daher eher, mehrere Instanzen des gleichen Prozesses parallel zu starten.

ja	nein
☐	■

Nein -> Neue Kindprozesse in UNIX werden nicht benachteiligt und haben die gleichen Berechtigungen wie ihre Erzeuger.

Bitte beachten: aufgrund der ganzen Änderungen rund um Multiple-Choice-Aufgaben wird es ab diesem Semester keinen solchen Aufgabenteil mehr geben.

Lösung 2 (Prozesse) **(3 + 6 + 1) = 10 Punkte**

a) Was versteht man unter einem *Prozesskontrollblock* und welche Informationen enthält er? Nennen Sie 4 Beispiele.

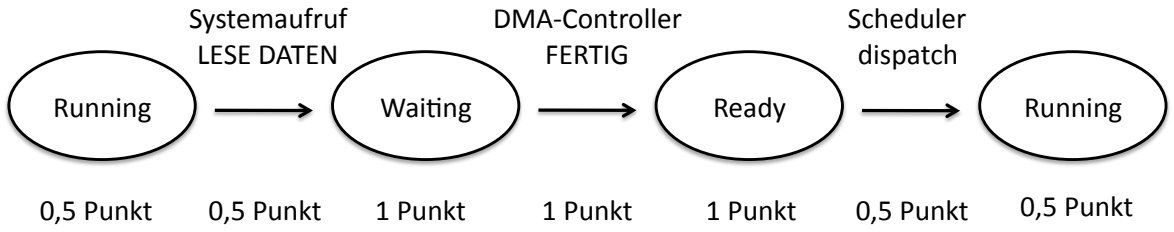
PCBs sind die Repräsentation von Prozessen innerhalb des Betriebssystems.
Sie enthalten u.a. folgende Informationen: Status, Programmzähler, CPU-Register, CPU-Scheduling, Speichermanagement, I/O-Status, Accounting.

Bewertung: Ein Punkt für die Beschreibung des Prozesskontrollblocks, je ein halber Punkt die Beispiele (max 2 Punkte).

b) Ein bereits laufender Benutzerprozess möchte 200 MB Daten von einer DMA-fähigen Festplatte auslesen. Was wird seitens des Benutzerprozesses für die Kommunikation mit dem Betriebssystem benötigt? *Skizzieren* Sie den Auslesevorgang in einem Zeit-Diagramm. *Beschriften* Sie die Prozesszustände und die zugehörigen Übergänge.

Die Kommunikation erfolgt über Systemaufrufe (1 Punkt).

Diagramm:



c) Welchen *Vorteil* bringt die DMA-Technologie im Beispielszenario b)?

Während die Daten direkt von der Festplatte in den Hauptspeicher kopiert werden, kann die CPU andere Prozesse bedienen. Oder auch den Benutzerprozess, falls er thread-basiert ist.

Bewertung: Ein Punkt für die richtige Antwort.

Lösung 3 (Prozesse und Prozesszustände) **(2 + 4 + 4 + 2) = 12 Punkte**

a) Worin liegen die Gemeinsamkeiten und Unterschiede von Prozessen und Threads? Nennen Sie *zwei Gemeinsamkeiten und zwei Unterschiede*.

Verschiedene Lösungen möglich, z.B.:

Gemeinsamkeiten:
- eigene Laufzeitumgebung wie Programmzähler, Stack, ...

Unterschiede:
- Threads nutzen Adressraum des Prozesses mit
- Threads teilen sich die CPU-Zeit eines Prozesses
- einfachere Erstellung, einfacherer CPU-Wechsel, schnellere Kommunikation
- Zugriff auf gemeinsamen Speicher koordinieren

Bewertung: je ein halber Punkt für eine Gemeinsamkeit bzw. einen Unterschied.

b) Was geschieht bei einem Prozesswechsel auf der CPU? *Skizzieren Sie knapp die Aktionen*, die das Betriebssystem unternimmt, um zwischen zwei Prozessen umzuschalten. Begründen Sie auf dieser Basis: sollten *Prozesswechsel bei Verwendung eines Round-Robin-Schedulings sehr häufig oder eher selten* stattfinden?

Siehe Folie 2.8:
- wird ein Prozesswechsel z.B. durch einen Interrupt ausgelöst, wird der Systemzustand gespeichert (als Prozesskontrollblock), d.h. wir geben die Register der CPU frei, merken uns den Programmzähler usw.
- der entsprechende Block des zweiten Prozesses wird geladen, um den zweiten Prozess ausführen zu können.

Daher: Umschalten kostet Zeit - man sollte nicht zu schnell umschalten, da der Overhead sonst irgendwann enorm wird.

Zwei Punkte für die Beschreibung des Prozesswechsels, zwei Punkte für die Begründung.
Auch eine sinnige Begründung in die andere Richtung kann einen Punkt geben.
Ohne Begründung gibt es allerdings nix.

c) Geben Sie ein Beispiel mit zwei Prozessen und einem Betriebsmittel an, das zeigt, wie *ein Prozess den Zustand **blocked** erreicht*. Geben Sie dazu in der ersten Tabelle die Ankunftszeitpunkte und Dauern der Prozesse P_1 und P_2 an, sowie nach wie vielen Zeiteinheiten im Zustand **running** auf das Betriebsmittel zugegriffen werden soll.

Tragen Sie in die zweite Tabelle die Zustände der Prozesse für Ihr Beispiel ein. Die **ready**-Warteschlange des Systems wird nach dem FIFO-Verfahren abgearbeitet.

Prozesse:

Prozess	P_1	P_2
Startzeitpunkt	**0**	**1**
Benötigte Zeiteinheiten im Zustand **running**	**2**	**2**
Nutzung des BM nach Zeiteinheit	**1**	**1**
Benötigte Zeiteinheiten der BM-Nutzung	**3**	**1**

Bewertung: 2 Punkte insgesamt

Prozesszustände:

	0	1	2	3	4	5	6	7	8	9
P_1	**ne**	**ru**	**wa**	**wa**	**wa**	**ru**	**te**			
P_2	**-**	**ne**	**ru**	**bl**	**bl**	**wa**	**ru**	**te**		

Bewertung: 2 Punkte insgesamt

d) Sie verwenden ein Betriebssystem, das über keinen gemeinsamen Speicher (Shared Memory) verfügt. Allerdings ist es für Ihre Prozessen nötig, miteinander zu kommunizieren. *Wie können Sie eine Interprozesskomunnikation erreichen?* Skizzieren Sie das vorgeschlagene Verfahren.

Durch Message Passing. (1 Punkt)

Zwei Befehle: send(A, m) und receive(A, m) - zum senden (bzw. empfangen) einer Nachricht m an einen (von einem) Prozess A. (1 Punkt)

Lösung 4 (Prozessverwaltung / C-Programmierung) **(6 + 4 + 1) = 11 Punkte**

a) *Schreiben* Sie ein C-Programm, das in einer Linux/Unix-Umgebung einen Kindprozess erzeugt. Der Vaterprozess soll in einer Endlosschleife "Sys" ausgeben. Der Kindprozess soll ebenfalls in einer Endlosschleife "Pro" ausgeben. Verwenden Sie dazu das folgende Rahmenprogramm:

Lösung:

```
int main() {
  pid_t pid = fork();
  if (pid == 0) { /* Sohn */
    while (1) {
      printf("Pro");
    }
  } else { /* Vater */
    while (1) {
      printf("Sys");
    }
  }
  return -1;
}
```

Bewertung: 2 Punkte für korrekte fork() Verwendung, 2 Punkte für korrekte Kontrollflussbehandlung (Vater/-Sohn Unterscheidung), 2 für Ausgabe.

b) Ändern Sie den Code des Vaterprozesses: Sobald er 10 Mal "Sys" ausgegeben hat, soll er seinen Kindprozess beenden. Verwenden Sie dazu die Funktion *kill(pid_t pid, int sig)*.

```
...
  } else { /* Vater */
    for (int i = 0; i < 10; i++) {
      printf("Sys");
    }
    /* beende den Kindprozess */
    kill(pid, SIGTERM); /* SIGKILL kann man auch als richtig ansehen  */
  }
...
```

Bewertung: 2 Punkte für die Schleife, 2 für die richtige kill-Anwendung.

c) *Beschreiben* Sie informell in 1-2 Sätzen, wie man den obigen Code erweitern muss, damit der Kindprozess die Signale seines Vaterprozesses (gleichzeitig auch Signale anderer Prozesse) ignoriert und seine Ausgabe fortsetzt.

Signal handler Funktion implementieren, die nichts macht. Bei SIGKILL wird es nicht funktionieren.

Bewertung: 1 Punkt für richtige Antwort.

Lösung 5 (Scheduling) **(4 + 4 + 4 + 2) = 14 Punkte**

Gegeben sei ein Rechnersystem mit einer CPU und 5 Prozessen $P_0, \ldots, P_4$. In der folgenden Tabelle ist für diese Prozesse angegeben, zu welchen Zeitpunkten sie das System betreten sowie für wie viele Zeiteinheiten sie die CPU benötigen:

Prozess	Ankunftszeitpunkt	Dauer
P_0	0	5
P_1	2	2
P_2	3	6
P_3	6	7
P_4	9	2

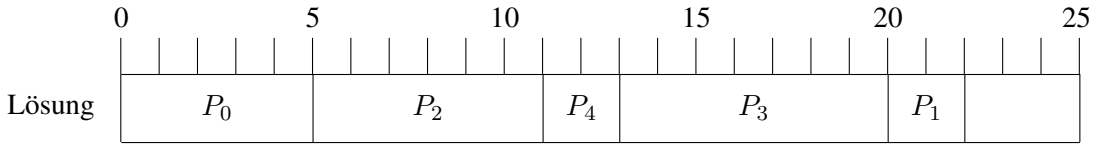
Ein Ankunftszeitpunkt von t bedeutet, dass der Prozess zu diesem Zeitpunkt im Zustand **ready** auf Zuteilung der CPU wartet. Zwischen zwei aufeinanderfolgenden Zeitpunkten t und $t + 1$ vergeht eine Zeiteinheit. Das Umschalten zwischen zwei Prozessen nehme keine Zeit in Anspruch. Hat ein Prozess für die unter **Dauer** angegebenen Zeiteinheiten die CPU belegt, verlässt er sofort das System, ohne weitere CPU-Zeit zu beanspruchen.

Im Folgenden sind 3 Scheduling-Strategien angegeben. *Illustrieren Sie die Belegung der CPU anhand der vorgegebenen Gantt-Charts*. Die Zahlen über den jeweiligen Charts bezeichnen Zeitpunkte und sind lediglich vorgegeben, um eine bessere Orientierung bei der Erstellung der Lösung zu geben. Zu jeder Strategie sind zwei Vorlagen vorhanden; die mit "Konzept" beschriftete Vorlage können Sie zur Erarbeitung der Lösung verwenden. **Wenn sowohl die mit "Konzept" als auch die mit "Lösung" beschriftete Vorlage Einträge enthalten, wird nur die mit "Lösung" beschriftete Vorlage gewertet!**

Berechnen Sie außerdem für jede Strategie die *mittlere Wartezeit* für Ihren Schedule. Führen Sie Ihre Berechnungen in dem mit „Rechnung" bezeichneten Feld durch und tragen Sie das Endergebnis in das mit „Ergebnis" bezeichnete Feld ein.

Bewertung für a - c: jeweils 2 für den korrekten Schedule und 2 für die Berechnung der Wartezeit.

a) Strategie: **LIFO**



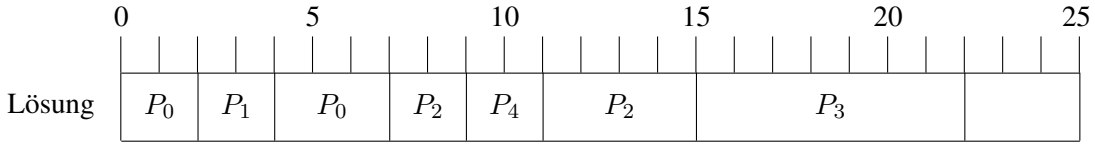
Mittlere Wartezeit:

Rechnung:

$\frac{0+18+2+7+2}{5} = \frac{29}{5} = 5,8$

Ergebnis: $\frac{29}{5} = 5,8$

b) Strategie: **SRPT**



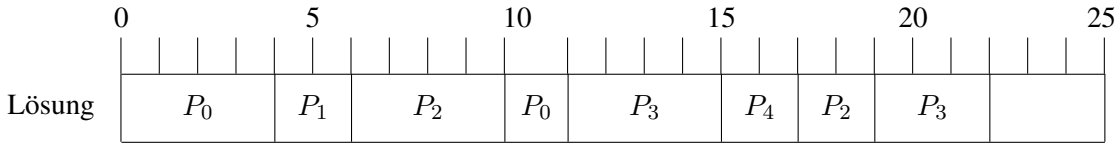
Mittlere Wartezeit:

Rechnung:

$\frac{2+0+6+9+0}{5} = \frac{17}{5} = 3,4$

Ergebnis: $\frac{17}{5} = 3,4$

c) Strategie: **Round Robin mit Quantum** $Q = 4$



Mittlere Wartezeit:

Rechnung:

$\frac{6+2+10+9+6}{5} = \frac{33}{5} = 6,6$

Ergebnis: $\frac{33}{5} = 6,6$

d) Sind die Strategien wie hier verwendet *work conserving*? Begründen Sie Ihre Antwort!

Ja - die Umschaltzeiten werden vernachlässigt. (Korrekte Antwort 2 Punkte, nur ein ja ohne Begründung 0 Punkte.

Lösung 6 (Hauptspeicherverwaltung) **(1 + 9 + 2 + 3) = 15 Punkte**

a) Beschreiben Sie in einem Satz, wofür in Betriebssystemen die *Lifetime-Funktion* verwendet wird.

Die Lifetimefunktion $L(m)$ gibt die mittlere Zeit zwischen aufeinanderfolgenden Seitenfehlern in Abhängigkeit von der zugeordneten Rahmenzahl m an, um die optimale Speicherzuteilung zu bestimmen. (1 Punkt).

b) Zeichnen Sie in die Vorlage auf der nächsten Seite den *Graphen der Lifetime-Funktion* $L(m)$ für $m = 1 \ldots 10$ und geben Sie zusätzlich in der Tabelle die *Zahl der Seitenfehler* für $m = 1 \ldots 10$ an zu einem Programm an, das durch den Referenzstring ω mit

$$\omega = 0\,5\,1\,4\,2\,5\,3\,0\,1\,4\,1\,2\,5\,4\,2\,5\,4\,2\,5\,1\,0\,2\,1\,5$$

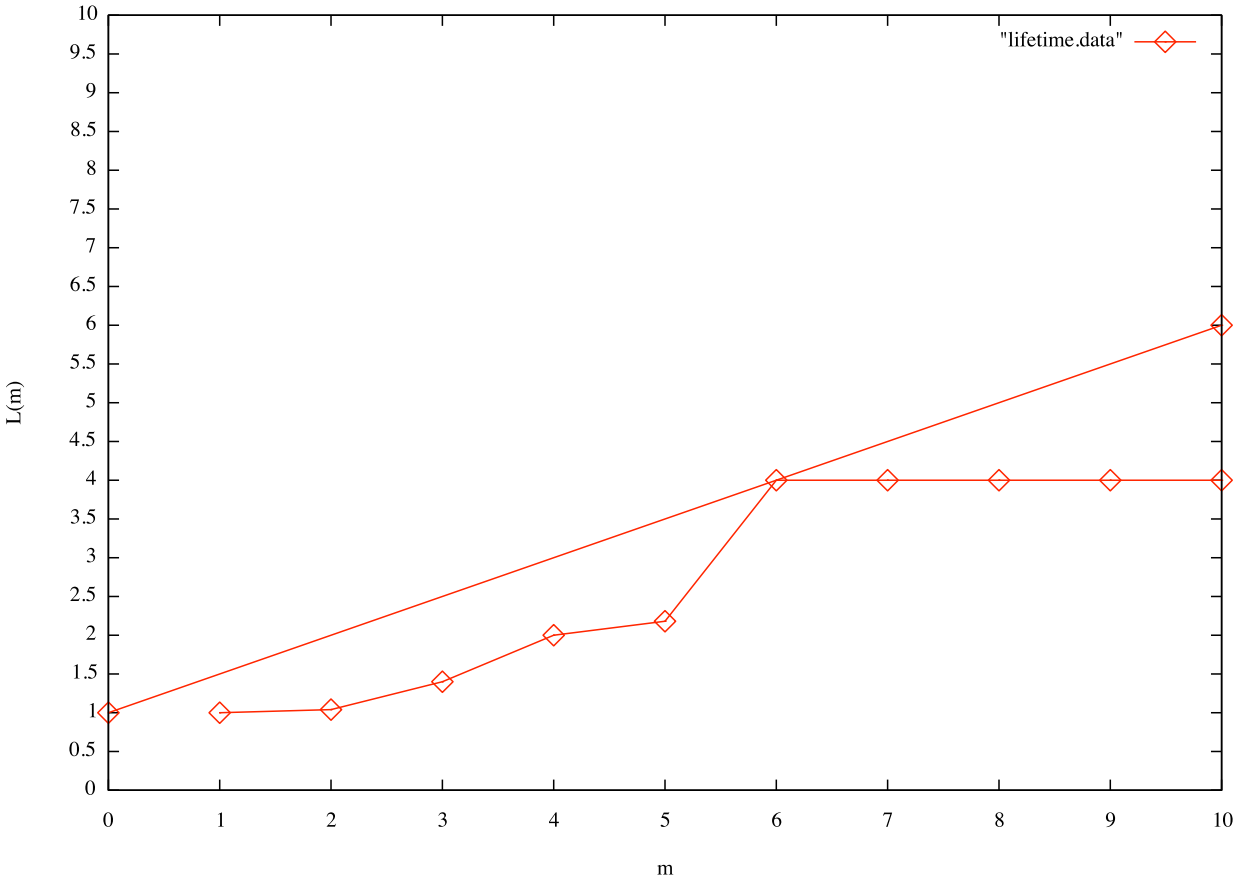
gekennzeichnet ist. Die Zeit, die zwischen zwei Seitenanfragen vergeht, soll jeweils eine Zeiteinheit betragen. Die Seitenersetzung erfolge mittels der folgenden Strategie: muss eine neue Seite eingefügt werden, so wird diejenige verdrängt, auf die am längsten nicht mehr zugegriffen wurde. Gehen Sie davon aus, dass das Working Set zu Beginn mit anderen als den angegebenen Seiten gefüllt ist.

$\omega =$	0	5	1	4	2	5	3	0	1	4	1	2	5	4	2	5	4	2	5	1	0	2	1	5
m = 2																								
Seitenfehler	*	*	*	*	*	*	*	*	*	*		*	*	*	*	*	*	*	*	*	*	*	*	*
Seitenrahmen	0	5	1	4	2	5	3	0	1	4	1	2	5	4	2	5	4	2	5	1	0	2	1	5
Seitenrahmen		0	5	1	4	2	5	3	0	1	4	1	2	5	4	2	5	4	2	5	1	0	2	1
m = 3																								
Seitenfehler	*	*	*	*	*	*	*	*	*	*		*	*	*						*	*	*		*
Seitenrahmen 1	0	5	1	4	2	5	3	0	1	4	1	2	5	4	2	5	4	2	5	1	0	2	1	5
Seitenrahmen 2		0	5	1	4	2	5	3	0	1	4	1	2	5	4	2	5	4	2	5	1	0	2	1
Seitenrahmen 3			0	5	1	4	2	5	3	0	0	4	1	2	5	4	2	5	4	2	5	1	0	2
m = 4																								
Seitenfehler	*	*	*	*	*		*	*	*	*		*	*								*			
Seitenrahmen 1	0	5	1	4	2	5	3	0	1	4	1	2	5	4	2	5	4	2	5	1	0	2	1	5
Seitenrahmen 2		0	5	1	4	2	5	3	0	1	4	1	2	5	4	2	5	4	2	5	1	0	2	1
Seitenrahmen 3			0	5	1	4	2	5	3	0	0	4	1	2	5	4	2	5	4	2	5	1	0	2
Seitenrahmen 4				0	5	1	4	2	5	3	3	0	4	1	1	1	1	1	1	4	2	5	5	0
m = 5																								
Seitenfehler	*	*	*	*	*		*	*	*	*		*	*											
Seitenrahmen 1	0	5	1	4	2	5	3	0	1	4	1	2	5	4	2	5	4	2	5	1	0	2	1	5
Seitenrahmen 2		0	5	1	4	2	5	3	0	1	4	1	2	5	4	2	5	4	2	5	1	0	2	1
Seitenrahmen 3			0	5	1	4	2	5	3	0	0	4	1	2	5	4	2	5	4	2	5	1	0	2
Seitenrahmen 4				0	5	1	4	2	5	3	3	0	4	1	1	1	1	1	1	4	2	5	5	0
Seitenrahmen 5					0	0	1	4	2	5	5	3	0	0	0	0	0	0	0	0	4	4	4	4
m = 6...10																								
Seitenfehler	*	*	*	*	*		*																	
Seitenrahmen 1	0	5	1	4	2	5	3	0	1	4	1	2	5	4	2	5	4	2	5	1	0	2	1	5
Seitenrahmen 2		0	5	1	4	2	5	3	0	1	4	1	2	5	4	2	5	4	2	5	1	0	2	1
Seitenrahmen 3			0	5	1	4	2	5	3	0	0	4	1	2	5	4	2	5	4	2	5	1	0	2
Seitenrahmen 4				0	5	1	4	2	5	3	3	0	4	1	1	1	1	1	1	4	2	5	5	0
Seitenrahmen 5					0	0	1	4	2	5	5	3	0	0	0	0	0	0	0	0	4	4	4	4
Seitenrahmen 6							0	1	4	2	2	5	3	3	3	3	3	3	3	3	3	3	3	3

Lifetime-Funktion:

m	1	2	3	4	5	6 . . . 10
Seitenfehler	24	23	17	12	11	6
$L(m)$	1 (24/24)	1.04 (24/23)	1.4 (24/17)	2 (24/12)	2.18 (24/11)	4 (24/6)
Bewertung	0,5 Punkt	0,5 Punkt	0,5 Punkt	0,5 Punkt	0,5 Punkt	0,5 Punkt

c) Welche *optimale Einstellung der Speichergröße* ergibt sich bei Anwendung des *primären Knie-Kriteriums*? Zeichnen Sie zusätzlich die *zugehörige Gerade* in Ihren Graphen ein.



$m_{opt} = 6$. Die zugehörige Gerade bei Anwendung des Knie-Kriteriums (s.o).
Achtung, bitte beachten: die Tangente wird immer im Punkt (0,1) begonnen, wie oben eingezeichnet.

Bewertung: optimale Speichergrösse: 1 Punkt. Gerade richtig eingezeichnet: 1 Punkt.

d) Geben Sie *alle Working-Sets* zum Zeitpunkt $t = 20$ an.

Working-Sets zum Zeitpunkt 20: {1}, {1,5}, {1,2,5}, {1,2,4,5}, {0,1,2,4,5}, {0,1,2,3,4,5}

Bewertung: jeweils ein halber Punkt.

Lösung 7 (Wechselseitiger Ausschluss) **(5 + 5 + 5) = 15 Punkte**

Zur korrekten Lösung des Problems des wechselseitgen Ausschlusses müssen drei Bedingungen erfüllt werden. Ein angehender Programmierer hat eine Lösung für zwei Prozesse P_i und P_j entworfen. Im Folgenden ist die Lösung für Prozess P_i gegeben (die Lösung für Prozess P_j sieht analog aus, nur mit vertauschten Indizes i und j):

```
repeat
    flag[i] := TRUE;
    turn := j;
    while (flag[i] and turn = j) do noop;
    kritischer Bereich;
    flag[i] := FALSE;
    unkritischer Bereich;
until FALSE;
```

Nennen Sie im Folgenden die drei Bedingungen für den wechselseitigen Ausschluss, geben Sie jeweils kurz die *Bedeutung* der Bedingung an und *begründen* Sie für jede Bedingung, warum die gegebene Lösung sie erfüllt bzw. nicht erfüllt.

Pro Teilaufgabe: 0,5 Punkte für die Bedingung, 1,5 Punkte für die Beschreibung, 0.5 Punkte für das richtige Kreuz und 2,5 Punkte für eine Begründung.

a) Bedingung 1: Mutual Exclusion

Bedeutung:

Nur ein Prozess auf einmal darf im KB sein.

Ist die Bedingung erfüllt? ja ■ nein □

Begründung:

Die Bedingung *flag[i]* kann ignoriert werden - sie ist hirnrissig, da das Flag direkt vorher erst auf true gesetzt wird und somit imemr wahr ist. Also prüft man durch *turn = j* lediglich, ob einem der andere Prozess Eintritt gewährt. Man selber hängt solange in *noop* fest, bis ein anderer Prozess einem das *turn* setzt. Dann darf man weiter machen, während der andere Prozess hängt - solange, bis der erste Prozess wieder von vorne beginnt und den Wert von *turn* wieder modifiziert.

b) Bedingung 2: **Bounded Waiting**

Bedeutung:

Fairnesskriterium: ein Prozess kann beim Versuch, in den kritischen Bereich einzutreten, nur endlich oft überholt werden, d.h. es gibt eine obere Schranke dafür, wie oft er überholt werden kann.
In der Vorlesung auch gesagt: es gibt eine obere Grenze für die Wartezeit eines Prozesses

Ist die Bedingung erfüllt? ja ■ nein □

Begründung:

Fairness ist gegeben, aus dem gleichen Grund wie oben: die Prozesse schalten sich gegenseitig frei, also kommen sie immer nur abwechselnd in den kritischen Bereich. Überholen ist nicht möglich.

Wer allerdings das in der Vorlesung erwähnte Zeitkriterium zugrunde legt, kann auch für ein Nein argumentieren (Begründung ähnlich wie bei Progress, siehe unten).

c) Bedingung 3: **Progress**

Bedeutung:

In endlicher Zeit wird die Entscheidung getroffen, welcher Prozess weitermacht, und dies nur abhängig von den wartenden Prozessen (also sozusagen Deadlockvermeidung).

Ist die Bedingung erfüllt? ja □ nein ■

Begründung:

Angenommen, ein Prozess stirbt im unkritischen Bereich - dann wird er nie mehr von vorne starten und *turn* so setzen, daß der andere Prozess weiter kommt.

Lösung 8 (Semaphore) **(2 + 3 + 4 + 10) = 19 Punkte**

Sie sollen eine Zufahrtsregelung für eine altersschwache Brücke entwerfen. Aus Sicherheitsgründen darf sich immer nur eine bestimmte Anzahl von Fahrzeugen gleichzeitig auf der Brücke befinden. Sie wollen Semaphore mit assoziierten Warteschlangen verwenden, um die Zufahrt zur Brücke zu regulieren und haben bereits die folgenden Vorgaben:

```
Fahrzeug(boolean Richtung)
{
    fahre_auf_Brücke_zu();
    enterBridge();
    überquereBrücke(); // kritischer Bereich
    exitBridge();
    fahre_weiter();
}
```

Die Variable *Richtung* wird pro Fahrzeug gesetzt und gibt an, in welche Richtung ein Fahrzeug die Brücke überqueren will.

Geben Sie die Implementierung von *enterBridge* und *exitBridge* sowie die *globalen Initalisierungsbedingungen der verwendeten Semaphore* (und, falls verwendet, globalen Variablen) für die folgenden vier Fälle an.

a) Zu jedem Zeitpunkt darf sich nur *ein einziges Fahrzeug auf der gesamten Brücke* befinden.

```
// Zum warm werden: simple Semphore reichen, da ja immer nur genau
// einer im kritischen Bereich sein darf.

init(s,1); // Sempahor initialisieren: nur einer auf der Brücke

enterBridge()
    wait(s); // auf Freigabe warten

exitBridge()
    signal(s); // Brücke freigeben
```

b) Zu jedem Zeitpunkt darf sich *pro Fahrtrichtung ein Fahrzeug auf der Brücke* befinden.

```
// Wie oben - aber für jede Richtung einzeln. also einfach mit zwei
// Semaphoren unter Abfrage der Fahrtrichtung.

init(s, 1); // Fahrtrichtung Nord

init(t, 1); //Fahrtrichtung Süd

enterBridge()
    if (Richtung == TRUE) // Fahrtrichtung abfragen
        wait(s);          // Fahrtrecht nach Norden bekommen
    else
        wait(t);          // Fahrtrecht nach Süden bekommen

exitBridge()
    if (Richtung == TRUE) // Fahrtrichtung abfragen
        signal(s);        // Fahrtrecht nach Norden freigeben
    else
        signal(t);        // Fahrtrecht nach Süden freigeben
```

c) Fahrzeuge der beiden Fahrtrichtungen dürfen nur *abwechselnd* die Brücke überqueren. Sie können frei entscheiden, welche der Fahrtrichtungen beginnen darf.

```
// Am einfachsten: wie oben, aber wechselseitiges Freischalten der
// beiden Richtungen.
// Oder: mit globaler Variable zur Verwaltung der Fahrtrichtung,
// ist ein wenig haariger.

init(s, 1); // Fahrtrichtung Nord fängt an

init(t, 0); //Fahrtrichtung Süd muss warten

enterBridge()                // wie oben
    if (Richtung == TRUE)
        wait(s);
    else
        wait(t);

exitBridge()                // wie oben, aber Freischalten der Gegenrichtung
    if (Richtung == TRUE)
        signal(t);
    else
        signal(s);
```

d) Zu jedem Zeitpunkt darf nur *eine Fahrtrichtung* genutzt werden, aber es dürfen *beliebig viele Fahrzeuge* in diese Richtung fahren.

```
// Jetzt wird's spannend: sozusagen ein Reader/Writer-Problem
// mit zwei Gruppen von Readern.
// Hier nur ein Beispiel - es gibt massenhaft andere Lösungsansätze.

init(s, 1); // Warteschlange für eine der Richtungen

init(mutex, 1); // Zugriff auf globale Variablen

int n = 0; // anfangs ist niemand auf der Brücke

int m = 0; // wartende Autos in anderer Richtung

boolean dir = TRUE;  // gloable Variable für aktuelle Fahrtrichtung

enterbridge()
    wait(mutex);     // auf Zähler exklusiv zugreifen
        if (n == 0) // niemand auf der Brücke
            wait(s); // Brücke befahren
            dir = Richtung; // erlaubte Fahrtrichtung setzen
            n = 1;  // bekannt geben, wieveile autos jetzt auf der Brücke sind
            signal(mutex); // andere Fahrzeuge an die Brücke anfahren lassen
        else if (Richtung == dir) // meine Richtung fährt gerade
            n = n + 1; // Auf Brücke auffahren
            signal(mutex); // andere Fahrzeuge an die Brücke anfahren lassen
        else
            m = m + 1; // in Warteschlange einreihen
            signal(mutex); // Freigeben der Brücke wegen Sinnlosigkeit
            wait(s); // warten, bis andere Richtung fertig ist
            wait(mutex);
                m = m - 1;
                n = n + 1;  // warteschlange verlassen, auf Brücke fahren
            signal(mutex);


exitBridge()
    wait(mutex);
        if (n > 1)
            n = n - 1;  // Verlassen der Brücke bekannt geben
        else
            n = 0; // Brücke leer
            if (m = 0)
                signal(s);    // niemand da - Ausgansgszustand herstellen
                signal(mutex);
            else
                dir = !dir; // andere Richtung dranlassen
                for (int i = 1 to m)
                    signal(s);    // alle aufwecken
                signal(mutex);
```

Lösung 9 (Deadlocks) **(8 + 5 + 1) = 14 Punkte**

Gegeben seien vier Prozesse P_1, P_2, P_3 und P_4, die zur Lösung Ihrer Aufgaben eine bestimmte Anzahl der Betriebsmittel R_1 und R_2 benötigen. Von beiden Betriebsmitteln gibt es jeweils 7 Exemplare.

Der Prozess P_i benötigt während seiner Laufzeit zu jedem Zeitpunkt maximal $Max(P_i)$ Einheiten der Betriebsmittel R_1, R_2:

$$Max(P_1) = (7,4), \quad Max(P_2) = (4,4), \quad Max(P_3) = (1,3), \quad Max(P_4) = (1,3)$$

Die aktuelle Betriebsmittelzuteilung sei wie folgt gegeben:

$$H_1 = (3,2), \quad H_2 = (2,2), \quad H_3 = (1,0), \quad H_4 = (0,1)$$

a) Überprüfen Sie mit Hilfe des *Banker-Algorithmus'*, ob sich das System in einem sicheren Zustand befindet.

$Q_1^{\max} = (4,2), \quad Q_2^{\max} = (2,2), \quad Q_3^{\max} = (0,3), \quad Q_4^{\max} = (1,2)$
$\Rightarrow V = (1,2)$

Durchlauf	Test	Ergebnis	Konsequenz
1	$Q_1^{\max} \leq V = (1,2)?$	Nein	
	$Q_2^{\max} \leq V?$	Nein	
	$Q_3^{\max} \leq V?$	Nein	
	$Q_4^{\max} \leq V?$	Ja	$V = V + H_4 = (1,3)$. Markiere P_4.
2	$Q_1^{\max} \leq V = (1,3)?$	Nein	
	$Q_2^{\max} \leq V?$	Nein	
	$Q_3^{\max} \leq V?$	Ja	$V = V + H_3 = (2,3)$. Markiere P_3.
3	$Q_1^{\max} \leq V = (2,3)?$	Nein	
	$Q_2^{\max} \leq V?$	Ja	$V = V + H_2 = (4,5)$. Markiere P_2.
4	$Q_1^{\max} \leq V = (4,5)?$	Ja	$V = V + H_1 = (7,7)$. Markiere P_1.

Alle Prozesse markiert, d.h. Zustand ist sicher!

Bewertung: korrekte Q je 0.5 Punkte (also 2 insgesamt), je 1,5 Punkte (im Schnitt) für jeden der vier Berechnungsschritte.

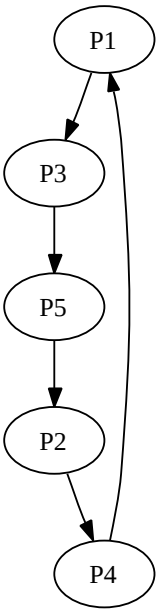
Betrachten Sie nun ein System mit fünf Prozessen $P_1, \ldots, P_5$ und fünf Betriebsmitteln $R_A, \ldots, R_E$, von denen jeweils nur ein Exemplar zur Verfügung steht und die von nur einem Prozess gleichzeitig verwendet werden können. Die Anforderungen von Betriebsmitteln durch die Prozesse ist in der folgenden Tabelle gegeben.

Zeit	P_1	P_2	P_3	P_4	P_5
1			(A;2)		
2	(C;5)			(D;7)	
3		(E;4)		(C;3)	
4		(D;4)			(A;3)
5			(B;4)		(E;4)
6	(B;3)		(A;3)		
7					

Ein Eintrag der Form $(X; a)$ in der Spalte P_i bedeutet, dass das Betriebsmittel R_X für a Zeiteinheiten vom Prozess P_i angefordert wird. Ein Prozess kann seine Arbeit erst dann fortsetzen, wenn ihm alle angeforderten Betriebsmittel zugewiesen wurden.

Sind einem Prozess die angeforderten Betriebsmittel zum Zeitpunkt t zugewiesen worden, werden sie in der Zeit von $t + 1$ bis $t + a$ verwendet und stehen zum Zeitpunkt $t + a + 1$ wieder zur Verfügung.

b) Verdeutlichen Sie die Abhängigkeiten der Prozesse zum Zeitpunkt 7 anhand eines *Wait-For-Graphen*.



Je korrekte Abhängigkeit 1 Punkt.

c) Liegt in dieser Situation ein *Deadlock* vor? . . . . . ja ■ nein □