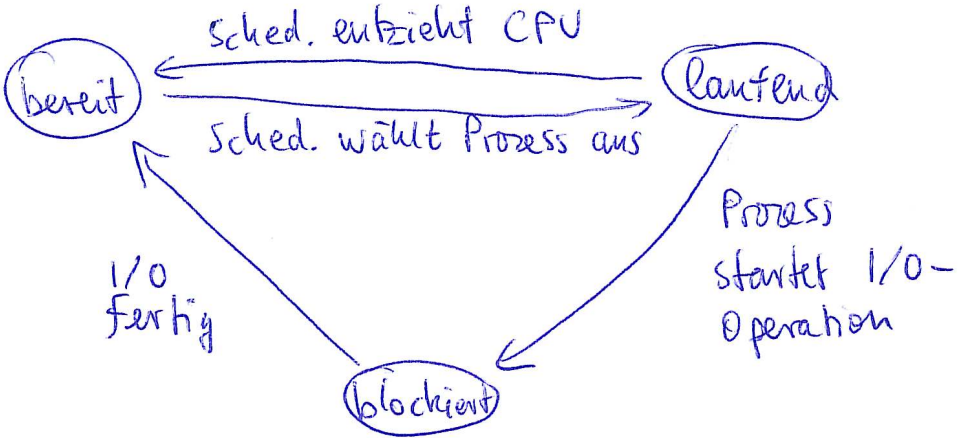
1. Prozess-Zustände (8/46 Punkte)

a) Die drei wichtigsten Prozesszustände sind **bereit**, **laufend** und **blockiert**. Beschreiben Sie, welche Übergänge zwischen diesen Prozessen möglich sind und warum es zu diesen Übergängen kommt. (Es reicht aus, die Zustände und Übergänge in einer Grafik zu zeichnen und die Übergangspfeile mit Stichworten zu erläutern.)

Intern: Speicherbereiche, die einem Prozess zugeordnet wurden, werden nicht vollständig genutzt, z. B. bei Aufteilung des Speichers in Partitionen fester, gleicher Größe und einem Prozess, der viel weniger Speicher benötigt

Extern: Im Zuge mehrfacher Vergabe und Rückgabe von Speicher bleiben kleine „Lücken": Speicherbereiche, die keinem Prozess zugeordnet sind und die so klein sind, dass vermutlich auch in der Zukunft kein Prozess sie verwenden kann. Z. B. bei dynamischer Aufteilung in (zusammenhängende) Partitionen.

b) **Threads** und **Prozesse** wechseln zwischen unterschiedlichen Zuständen hin und her. Nennen Sie zwei Zustände, die es nur bei Prozessen gibt, und begründen Sie jeweils kurz, warum es nicht sinnvoll ist, diese Zustände für Threads zu definieren.



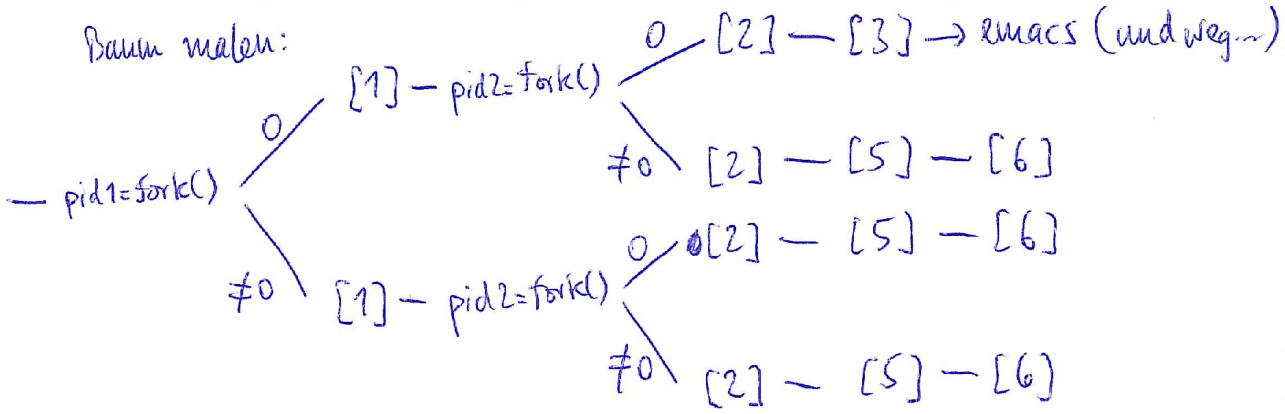
2. System calls (8/46 Punkte)

a) Betrachten Sie den folgenden Programmausschnitt:

```
...
int pid1 = fork();
printf ("%s\n","[1] Ein Fork ist durch, einer muss noch.");
int pid2 = fork();
printf ("%s\n","[2] Zeit für eine Fallunterscheidung.");
if ( (pid1==0) && (pid2==0) ) {
  printf ("%s\n","[3] Ich starte jetzt emacs.");
  execl ("/bin/emacs", "/etc/fstab", (char *)NULL);
  int pid3 = fork();
  printf ("%s\n","[4] Nach dem dritten Fork.");
} else {
  printf ("%s\n","[5] Ich gucke nur zu.");
};
printf ("%s\n","[6] Nach der if-Abfrage endet das Programm.");
```

} wird nie ausgeführt!

Wie oft und warum erscheinen die mit [1] bis [6] durchnummerierten Ausgaben? Schreiben Sie zu jeder Ausgabe die Anzahl und begründen Sie Ihre Antwort stichwortartig.



b) Beim Aufruf des System calls **`fork()`** erhält der Sohn den Wert 0 und der Vater die Prozess-ID des neu erzeugten Sohnes (≠0) zurück. Für eine reine Unterscheidung („wer bin ich?“) könnte man auch umgekehrt arbeiten, also im Vater den Wert 0 und im Sohn die Prozess-ID des Vaters (≠0) zurückgeben. Warum ist diese Alternative schlecht?

Der Sohn kann immer über getppid() (get parent process id) die ID des Vaterprozesses rausfinden, denn da gibt es ja nur einen. Anders rum kann aber der Vater nicht die PID eines bestimmten Sohnes über einen Funktionsaufruf erfahren – es könnte ja mehrere geben.

3. Scheduling-Verfahren (Uni-Prozessor) (11/46 Punkte)

a) Aus der Vorlesung kennen Sie die Scheduling-Verfahren **FCFS** (First Come First Served), **SRT** (Shortest Remaining Time Next) und Round Robin (**RR**).

Es gebe die folgenden fünf Prozesse mit den angegebenenen Ankunftszeiten und Gesamtrechenzeiten:

Prozess	Ankunft	Rechenzeit
P	0	10
Q	4	8
R	5	7
S	6	1
T	12	2

Zeit	0 1 2 3 4 5 6 7 8 9	0 1 2 3 4 5 6 7 8 9 10	0 1 2 3 4 5 6 7 20
FCFS	**P** P P P P P P P P P	**Q** Q Q Q Q Q Q Q **R** R	R R R R R **S** **T** T
SRT	**P** P P P P P P P P P	**S** **R** R R R R R R **T** T	**Q** Q Q Q Q Q Q Q
RR (q=2)	**P** P P P **Q** Q **P** P **R** R	**S** **Q** Q **P** P **R** R **T** T **Q**	Q **P** P **R** R **Q** Q **R**
RR (q=8)	**P** P P P P P P P **Q** Q	Q Q Q Q Q Q **R** R R R	R R R **S** **P** P **T** T

$q=2$:

0: [P]
[] (P)

4: [Q, P]
[P] (Q)
erst neues Q, dann unterbrochenes P

5: [P, R] (Q)
neues R

6: [P, R, S, Q]
[R, S, Q] (P)
neues S

8: [R, S, Q, P]
[S, Q, P] (R)

10: [S, Q, P, R]
[Q, P, R] (S)

11: [Q, P, R]
[P, R] (Q)

12: [P, R, T] (Q)
neues T

13: [P, R, T, Q]
[R, T, Q] (P)

15: [R, T, Q, P]
[T, Q, P] (R)

usw.

b) Wenn Sie FCFS mit RR(q=2) und RR(q=8) vergleichen, was fällt Ihnen dann auf? Bewerten Sie die Wahl des Zeitquantums q=2 bzw. q=8.

Kurze Prozesse (wie S) kommen früher dran (und werden damit auch früher fertig), weil die Warteschlange schneller rotiert. Dafür brauchen lange Prozesse (wie P) etwas länger, das ist aber nicht so tragisch. Bei q=2 fallen natürlich bis zu 4x so viele Context-Switches an wie bei q=8.

c) Was ist der Unterschied zwischen **I/O-lastigen Prozessen** und **CPU-lastigen Prozessen**?

I/O-lastig: verbringt die meiste Zeit mit dem Warten (Blockiert-sein) auf I/O-Operationen

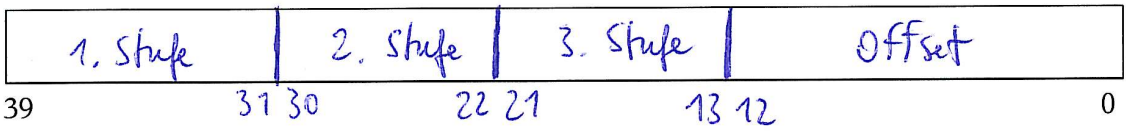
CPU-lastig: rechnet die meiste Zeit

4. Virtuelle Adressen (6/46 Punkte)

Eine CPU arbeitet mit folgenden Werten:

- Seitengröße: 8 KByte = 8192 Byte = 2^{13} Byte ⇒ 13 Bit Offset
- 40 Bit lange virtuelle Adressen → 40 − 13 = 27 Bit für Seitennummer
- 3-stufiges Paging; alle Seitentabellen sind gleich groß → 27/3 = 9 Bit pro Stufe
- Seitentabelleneinträge sind 8 Byte lang

a) Wie ist eine virtuelle Adresse aufgebaut (welche Bits der Adresse haben welche Bedeutung)?



Zeichnen Sie die die Unterteilung hier ein und beschriften Sie die Abschnitte geeignet.

b) Wie viele Seitentabellen der verschiedenen Stufen gibt es? Wie groß sind diese Tabellen?

Zur Größe: Auf jeder Stufe 2^9 = 512 (partielle) Seitennummern, alle je 2^9 Einträge. Jeder Eintrag ist 8 Byte lang
⇒ Größe = 512 × 8 = 4096 = 4 K (oder ½ Seite)

Zur Anzahl:

- 1. Stufe: 1 äußere Seitentabelle
- 2. Stufe: 2^9 = 512 mittlere Seitentabellen
- 3. Stufe: $2^9 \times 2^9$ = 262144 = 256 K innere Tabellen

5. Synchronisation (4/46 Punkte)

Mutexe und Semaphore helfen Programmierern bei der Synchronisation, ersparen ihnen aber nicht, den Programmcode sorgfältig nach kritischen Bereichen zu durchsuchen. **Monitore** sind eine Alternative – auf welche Weise erleichtern sie Programmierern die Arbeit?

Monitore kapseln die Daten (auf die potenziell von mehreren Threads zugegriffen wird) und die kritischen Bereiche (in denen diese Zugriffe stattfinden). Wie bei einer „private" deklarierten Variable kommen Programme nur noch über im Monitor definierte Monitorprozeduren (Funktionen) an diese Daten heran. Damit entfällt das manuelle Schützen aller kritischen Bereiche mit Mutexen.

6. Deadlocks (4/46 Punkte)

a) Auf einem System gebe es fünf Prozesse P_1, P_2, ..., P_5 und fünf exklusive Betriebsmittel R_1, R_2, ..., R_5. Der momentane Zustand sei wie folgt:

P_1 belegt keine Ressource und fordert R_1 an,

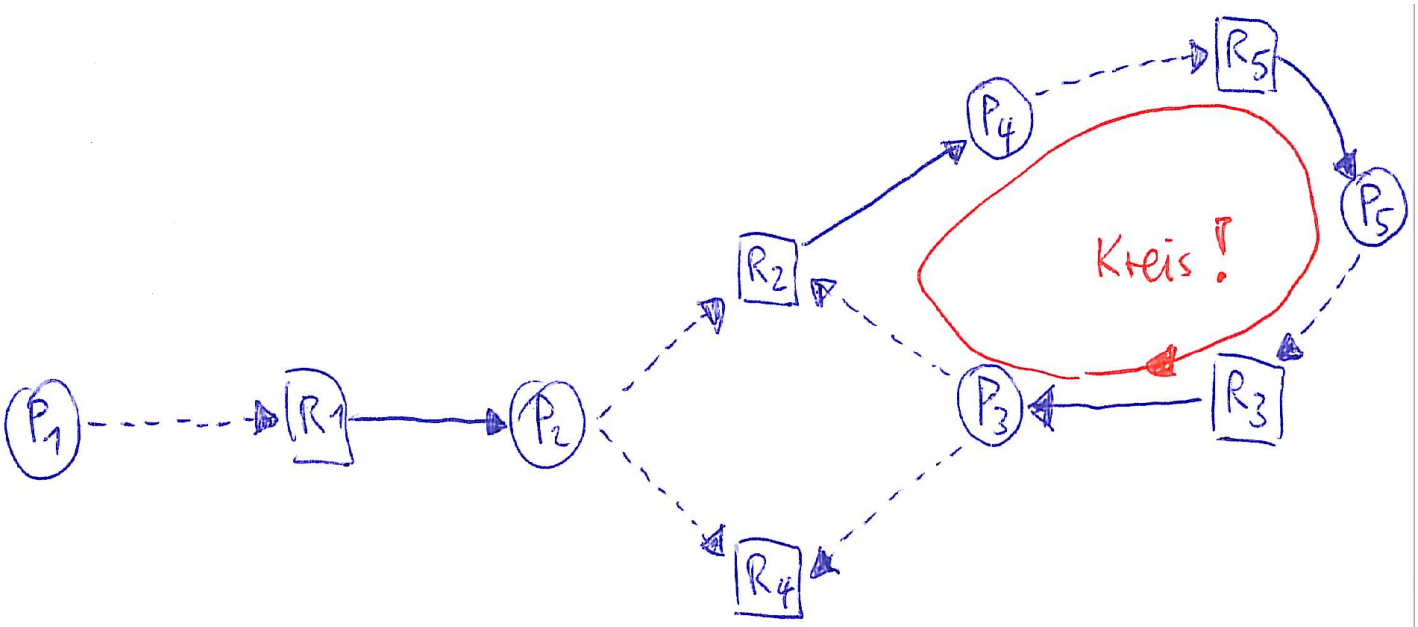
P_2 belegt R_1 und fordert R_2 und R_4 an,

P_3 belegt R_3 und fordert R_2 und R_4 an,

P_4 belegt R_2 und fordert R_5 an,

P_5 belegt R_5 und fordert R_3 an.

Überprüfen Sie mit dem Ressourcen-Zuordnungs-Graph, ob ein Deadlock vorliegt, und falls ja, welche Threads an dem Deadlock beteiligt sind.



Deadlocks der Prozesse P_3, P_4, P_5

7. Deadlocks: mit Matrix (5/46 Punkte)

Für drei Ressourcen-Klassen A, B und C sowie vier Prozesse P_1, P_2, P_3, P_4 seien folgende Ressourcenbelegungen (**C**) und Anforderungen (**R**) gegeben:

$$\mathbf{C} = \begin{pmatrix} 2 & 0 & 0 \\ 2 & 2 & 0 \\ 0 & 3 & 1 \\ 0 & 1 & 2 \end{pmatrix} \qquad \mathbf{R} = \begin{pmatrix} 0 & 1 & 0 \\ 3 & 5 & 0 \\ 2 & 2 & 0 \\ 3 & 8 & 0 \end{pmatrix}$$

$$E = (5\ 9\ 4)$$
$$A = (1\ 3\ 1)$$

A gibt die aktuell noch verfügbaren Ressourcen an, E die Gesamtressourcen.

Ist dieses System im Deadlock?

- Falls ja, geben Sie an, welche Prozesse und welche Ressourcen Teil des Deadlocks sind.
- Falls nein, geben Sie eine Ausführreihenfolge der Prozesse an, in der jeder Prozess zunächst seine vollständigen Ressourcenforderungen stellt und anschließend alle belegten Ressourcen freigibt.