# Lösungsvorschlag zum Aufgabenblatt 6 „Betriebssysteme“ (ehemalige Klausur)

**Beachte:** Die roten Anmerkungen gehören *nicht* zur Lösung der Aufgabe, sondern gehen auf häufige Fehler ein.

## Aufgabe 1 (2+2+2 Punkte)

a) Ein Prozess im Zustand „blockiert“ wartet auf Daten vom Netzwerk. Sobald die Daten vorhanden sind, wird der Prozess „aufgeweckt“. Was bedeutet eigentlich „aufwecken“?

Eine Zustandsüberführung nach „bereit“, d.h. Warten auf CPU-Zuteilung, und Aufruf des Schedulers

Dem Prozess wird nicht unbedingt die CPU zugeteilt!

b) Beschreiben Sie in Stichworten, wie in einem monolithischen Betriebssystem ein Systemaufruf ausgelöst und vom Systemkern bearbeitet wird.

- Systemaufruf-Funktion hinterlegt Parameter und erzeugt Interrupt (“Trap”) mit spez. Maschinenbefehl
- Interrupt-Mechanismus (Hardware) übergibt Kontrolle an Trap-Handler im Systemkern
- Trap-Handler aktiviert zuständige Funktion im Kernel, hinterlegt Resultat und gibt Kontrolle an Anwendung zurück

c) Ein Betriebssystem unterstützt Threads. Welche Ressourcen muss der Systemkern bei der Erzeugung eines neuen Thread bereitstellen?

(Thread-Deskriptor,) Laufzeitstack, Sicherungsbereich für Maschinenzustand

Der Thread hat keinen eigenen Heap oder eigene globale Variablen!

## Aufgabe 2 (4 Punkte)

Ein Benutzer eines UNIX-Systems betätigt die Tastenkombination Strg-C (Control-C) um ein Programm abzubrechen. Beschreiben Sie, was daraufhin passiert.

Tastatur erzeugt beim Drücken von Tasten Interrupt. Interrupt-Handler erzeugt wegen der Sonderfunktion der gedrückten Tasten Signal (SIGINT) für Vordergrundprozess(e). Bei CPU-Zuteilung führt Vordergrundprozess Signalbehandlung aus. Standard-Signalbehandlung bricht Prozess ab.

Kommt es in jedem Fall zum Abbruch eines Programms?

Nein, die Anwendung kann das Signal auch ignorieren oder eine sonstige Reaktion vorsehen.

Wenn ja, welches Programm ist betroffen?

Durch Window-Manager (aktives Fenster) und Shell (Vordergrund-Kommandoausführung) wird Tastatureingabe immer eindeutig einem Vordergrundprozess zugeordnet. Dieser Prozess erhält das Signal (genauer: Vordergrund-Prozessgruppe).

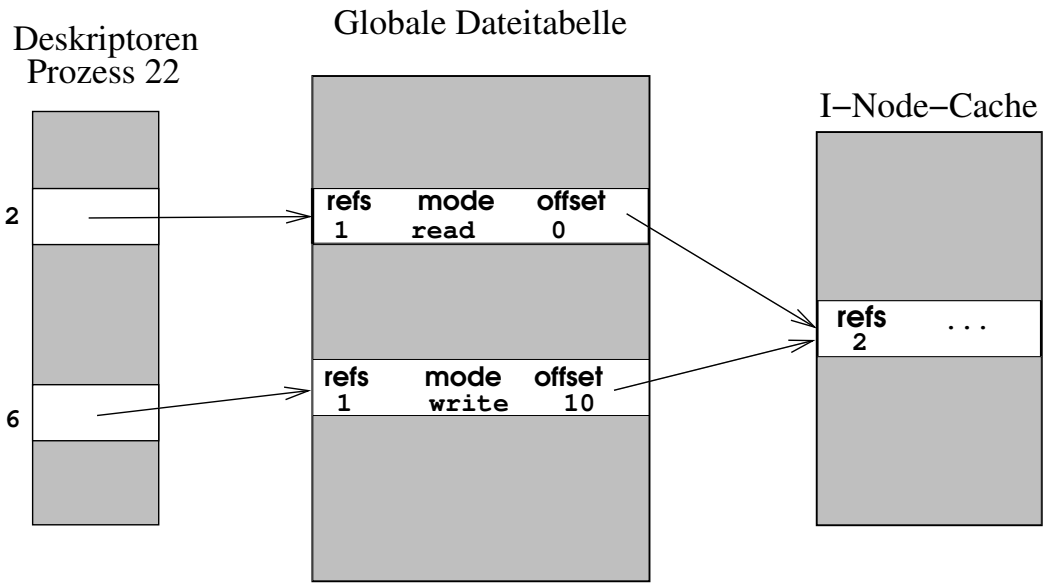
Es ist i.A. nicht der ausführende Prozess betroffen. Der Vordergrund-Prozess kann in einem beliebigen Zustand sein!

Wie kann überhaupt das Drücken von Tasten zur Terminierung eines Programms führen?

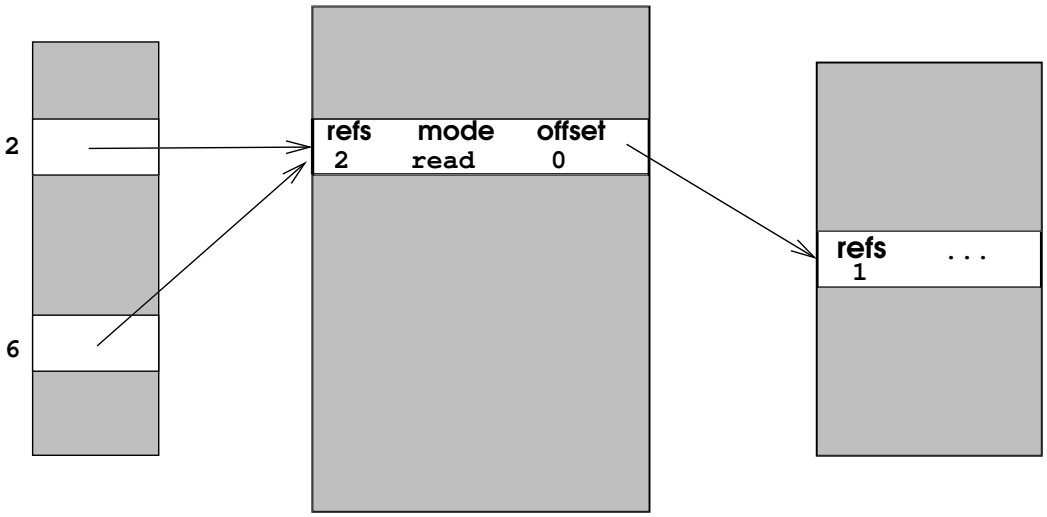
siehe oben

## Aufgabe 3 (4 Punkte)

Die Skizze unten enthält Informationen zu Dateien, die Prozess 22 zu einem bestimmten Zeitpunkt offen hat. Was passiert, wenn dieser Prozess nun den Systemaufruf `dup2(2,6)` ausführt. Wie ist der Zustand nach dem Aufruf?




Zustand nach dem dup(2,6)




Beachte die geänderten Referenzzähler! Der zweite Dateitabelleneintrag wird gelöscht, weil nach dem dup2 kein Verweis mehr auf ihn existiert.

## Aufgabe 4 (2+2 Punkte)

a) Das Working-Set-Prinzip garantiert jedem Prozess ein Mindestkontingent an Seitenrahmen. Was passiert aber, wenn zu viele Programme gestartet werden, so dass gar nicht genug Rahmen für alle da sind?

Einige blockierte Prozesse müssen mit einer “Swap-out”-Operation in den Swapbereich ausgelagert werden. Ausgelagert wird der komplette “Working-Set”. Falls kein Swapping möglich ist, misslingt die Prozesserzeugung.

b) Beschreiben Sie kurz, wie die Seitenauslagerungsstrategie „Second Chance“ funktioniert?

Die Seite, die am längsten im Speicher ist (FIFO-Liste), wird ausgelagert, wenn Sie nicht kürzlich (seit dem letzten Rücksetzen des Referenzbits) benutzt wurde. Ansonsten bekommt sie eine “2. Chance” und wird in der Liste wieder hinten angehängt.

Warum ist „Second Chance“ besser als „FIFO“?

Seiten, die ständig gebraucht werden, z.B. viel genutzte Bibliotheken, werden bei FIFO immer wieder ausgelagert, nicht aber bei Second-Chance. Die Seitenfehlerrate ist dadurch bei FIFO höher.

## Aufgabe 5 (1+1+2+2 Punkte)

Ein System verwaltet seine virtuellen Adressen mit 2-stufigen Seitentabellen. Eine Adresse v=(p1,p2,D) hat 32 Bit, davon p1 und p2 je 10 Bit und D 12 Bit.

Der virtuellen Adresse V=0140B704 ist eine reale Adresse im Rahmen 3 zugeordnet.

a) Geben Sie p1, p2 und D hexadezimal an.

p1=5 p2=B D=704

0140B in Binärdarstellung: 0000 0001 0100 0000 1010 →
p1=0000000101, p2=0000001011 (je 10Bit)

b) Geben Sie die reale Adresse hexadezimal an.

3704

genauer: 00003704, Distanz wie bei virtueller Adresse (704)

c) Geben Sie den TLB-Eintrag dazu hexadezimal an.

0140B → 00003

Seitennummer → Rahmennummer, Distanz steht nicht im TLB

d) Wo genau steht die Adresse der Seitentabelle der zweiten Stufe zu V?

im Eintrag 5 der Seitentabelle Stufe 1

## Aufgabe 6 (4 Punkte)

Von einer Shell mit der Prozessnummer 7 wird das nachfolgende Programm aufgerufen. Das Programm erhält die PID 8, der Kindprozess des Programms die PID 9. Welche Daten erscheinen in welcher Reihenfolge in der Standardausgabe?

```
#include <stdio.h>
#include <wait.h>
#include <unistd.h>

int main(){
   int p=fork();
   if(p) {
     printf("p=%d\n", p);
     printf("ppid=%d\n", getppid());
     }
   else
      printf("ppid=%d\n", getppid());

   waitpid(p, NULL, 0);
   printf("pid=%d\n", getpid());
}
```

Falls fork erfolgreich:

```
Ausgabe Hauptprozess:          Ausgabe Kind:

p=9                            ppid=8
ppid=7                         pid=9
pid=8
```

Reihenfolge: Die Ausgaben von Hauptprozess und Kind werden zeilenweise unvorhersehbar vermischt.

Ausgabe Hauptprozess, falls fork nicht erfolgreich:

```
p=-1
ppid=7
pid=8
```

### Aufgabe 7 (6 Punkte)

Die folgende Implementierung von vereinfachten Shell-Pipelines ist lückenhaft. Ergänzen Sie die Eingabeumlenkung im Leser und die Ausgabeumlenkung im Schreiber.

Auch im Hauptprozess fehlt noch etwas wichtiges. Erklären Sie warum. Ergänzen Sie das Fehlende.

```
int kommando::pipeline(){
    pid_t leser, schreiber;

    int pipefd[2];
    if(pipe(pipefd)){ perror("Fehler bei pipe"); exit(1); }

    switch(leser=fork()){
    case -1:  perror("Fehler bei fork"); return -1;
    case 0:
      // Pipe-Leser, Umlenkung der Eingabe
      dup2(pipefd[0},0);
      // nicht benötigte Deskriptoren schließen
      close(pipefd[0]);
      close(pipefd[1]);
      execlp(args[1], args[1], NULL);
      perror("exec-Fehler beim Pipe-Leser");
      exit(1);
    }

    switch(schreiber=fork()){
    case -1: perror("Fehler bei fork"); exit(1);
    case 0:
      // Schreiber: Ausgabeumlenkung
      dup2(pipefd[1],1);
      // nicht benötigte Deskriptoren schließen
      close(pipefd[0]);
      close(pipefd[1]);
      execlp(args[0], args[0], NULL);
      perror("exec-Fehler beim Pipe-Schreiber");
      exit(1);
    }
```

```
    // nicht benötigte Deskriptoren schließen
    close(pipefd[0]);
    close(pipefd[1]);


    // Warten auf Ende der beiden Pipeline-Prozesse
    waitpid(leser,NULL,0);
    waitpid(schreiber,NULL,0);
}
```

Die Pipe-Deskriptoren müssen im Hauptprozess geschlossen werden, da ansonsten eine Verklemmung zwischen Hauptprozess und Kindern möglich ist. Der Leser hängt ggf. im „read“, weil Hauptprozess Schreibdeskriptor offen hat. Der Schreiber hängt ggf. im „write“, weil Pipe voll und Hauptprozess Lesedeskriptor offen hat. Der Hauptprozess blockiert dann im entsprechenden waitpid für immer.

### Aufgabe 8 2+2+3 Punkte

a) Ein Round-Robin-Scheduler verdrängt den ausführenden Prozess nach Ablauf seines Quantums von 1ms. Welche Konsequenzen hat das?

   Das Quantum ist sehr kurz. Dadurch kommt jeder Prozess zwar schnell wieder an die Reihe, aber durch die vielen Kontextwechsel ist das System zu stark mit sich selbst beschäftigt: Der für die Ausführung des Anwendercodes verfügbare Anteil der CPU-Leistung sinkt stark ab.

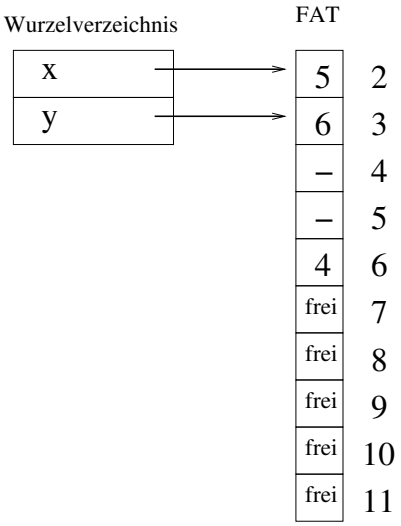
   (weitere sinnvolle Antworten möglich)

b) Wie könnte ein CPU-Scheduler dafür sorgen, dass das System immer schnell auf Mausklicks des Benutzers reagiert? Der auf die Mausklicks wartende Prozess müsste eine hohe dynamische Priorität erhalten. Dazu können spezifische Wartezustände benutzt werden, z.B. Zustand „Warten auf interaktive Eingabe“. Oder das System priorisiert Prozesse, die bei Ausführung immer wieder schnell wieder blockieren.

c) Welche Vorteile könnte es haben, wenn der Scheduler in einem Multiprozessorsystem immer versuchen würde, alle Threads eines Programms parallel auszuführen? Begründen Sie die Antwort. Um den exklusiven Zugriff der Threads auf gemeinsamen Hauptspeicher zu garantieren, könnte man in diesem Fall aktives Warten (Smartlocks/Spinlocks) verwenden, weil die Wartezeiten sehr kurz sind. Dies spart viele Kontextwechsel und führt zu einer sehr schnellen Ausführung des Programms.

## Aufgabe 9 (4 Punkte)

In einem FAT-Dateisystem gibt es 10 Cluster mit den Nummern 2-11. Es gibt 2 Dateien, beide im Wurzelverzeichnis: Datei x mit den Clustern 2 und 5, und Datei y mit den Clustern 3,6 und 4. Geben Sie als Skizze den Inhalt des Wurzelverzeichnisses und der FAT an.



## Aufgabe 10 (5 Punkte)

Ein Prozess P soll zwei nebenläufige Subprozesse erzeugen. Beide Subprozesse verhalten sich gleich: Zuerst wird die Funktion *f1*, danach die Funktion *f2* aufgerufen. Keiner darf aber *f2* aufrufen, bevor der andere den Aufruf von *f1* beendet hat. Beide Subprozesse von P müssen also ggf. auf ihre „Bruder"-Prozesse warten.

Geben Sie eine C-Implementierung an, die Pipes zur Synchronisation verwendet.

```
#include ...
...

void f1 () { printf("hier f1, aufgerufen von %d\n", getpid()); }
void f2 () { printf("hier f2, aufgerufen von %d\n", getpid()); }

int main(){
  int fd1[2], fd2[2]; //bidirektionale Kommunikation benötigt 2 (Einweg-)Pipes
  char c='x';

  if(pipe(fd1)) { perror("pipe1"); exit(1); }
  if(pipe(fd2)) { perror("pipe2"); exit(1); }

  switch(fork()){
  case -1: perror("fork1"); exit(1);
  case 0:
    f1();

    // dem anderen Subprozess mitteilen, dass f1-Aufruf fertig ist:
    // dazu ein beliebiges Byte in die erste Pipe schreiben
    if( write(fd1[1],&c,1) != 1 ) { perror("write 1"); exit(1); }

    // auf den f1-Aufruf des anderen warten,
    // dazu ein Byte aus zweiter Pipe lesen
    // blockiert, solange die zweite Pipe leer ist
    if( read(fd2[0],&c,1) != 1 )  { perror("read 1"); exit(1); }

    f2();
    exit(0);
  }

  // Subprozess 2 ist symmetrisch dazu
  switch(fork()){
  case -1: perror("fork2"); exit(1);
  case 0:
    f1();
    if( write(fd2[1],&c,1) != 1 )  { perror("write 2"); exit(1); }
    if( read(fd1[0],&c,1) != 1 )   { perror("read 2");  exit(1); }
    f2();
    exit(0);
  }

  exit(0);
}
```