Lösungsvorschlag zur Klausur “Betriebssysteme” – 5.2.2016

Aufgabe 1 (2+6 Punkte)

Punkte ____ *von* **8**

Ein Prozessor benutzt zur Optimierung der Hauptspeicherzugriffe einen schnellen Cache-Speicher. Dieser ist organisatorisch in „Zeilen“ unterteilt. In jede Cache-Zeile passt ein 128 Byte großer zusammenhängender Bereich des Hauptspeichers. Die Anfangsadresse jedes dieser Bereiche ist immer ein Vielfaches der Zeilengröße 128.

a) Nehmen Sie an, der Prozessor liest ein noch nicht im Cache vorhandenes 4 Byte großes Speicherwort mit der Adresse 264. Welche Speicherbytes stehen danach in der zugehörigen Cache-Zeile?

Der 128 Byte große Bereich mit den Adressen 256-383

Anmerkung:

Eine Zeilen-Aufteilung macht nur Sinn, wenn bei jedem Hauptspeicherzugriff eine ganze Zeile übertragen wird, z.B.

Zeile 0:	0-127
Zeile 1:	128-255
Zeile 2:	256-383
Zeile 3:	384-511

Das bei Adresse 264 beginnende Speicherwort gehört zur Zeile 2 (bei Zählung ab 0) des Hauptspeichers, so dass beim Lesen dieses Worts diese Zeile vollständig in den Cache kopiert wird.

b) Welche Informationen stehen außer dem Hauptspeicherinhalt sonst noch in dem Cache-Eintrag? Geben Sie die konkreten Werte dieser Informationen für das Beispiel an.

- Die Nummer der Zeile im Hauptspeicher, im Beispiel 2.
  Anmerkung:
  Die Hauptspeicheradresse der im Cache gepufferten Daten wird benötigt, um modifizierte Daten an die richtige Stelle zurückschreiben zu können. Aufgrund der Zeilen-Organisation reicht die Zeilennummer, denn durch Anhängen von 7 Null-Bits (Zeilengröße = 2^7 Byte) ergibt sich daraus die Hauptspeicheradresse des Zeilenanfangs. Aus diesem Grund ist die Zeilengröße immer eine Zweierpotenz.
- Statusbits zur Cache-Zeile:
  - Valid Bit: Gibt an, ob gültige Daten in der Zeile stehen. Wert im Beispiel: 1
  - Dirty Bit: Gibt an, ob die Daten in der Zeile modifiziert wurden. Wert im Beispiel: 0

Aufgabe 2 (4+4+6 Punkte)

Punkte ____ *von* **14**

a) Welches sind die einzelnen Schritte der Befehlsausführung bei einem Von-Neumann-Rechner?

- Befehl holen
- Befehl prüfen / dekodieren
- Speicheroperanden holen
- Befehl ausführen
- Ergebnis zurückschreiben

b) Was bedeutet „Pipelining“ im Zusammenhang mit der Ausführung von Maschinenbefehlen?

Die Ausführung eines Befehls wird in n Teilphasen zerlegt. Für jede ist eine andere Prozessorkomponente zuständig. Bei einer Befehlssequenz $I_1, I_2, I_3 \ldots$ wird z.B. gleichzeitig die 3. Phase von I_1, die 2. Phase von I_2 und die 1. Phase von I_3 bearbeitet. So können im Idealfall n Befehle gleichzeitig bearbeitet werden.

Beispiel mit 4 Teilphasen (FETCH, DECODE, EXECUTE, WRITEBACK), die jeweils 1 Takt benötigen:

WRITEBACK				I_1	I_2	$\ldots$
EXECUTE			I_1	I_2	I_3	$\ldots$
DECODE		I_1	I_2	I_3	I_4	$\ldots$
FETCH	I_1	I_2	I_3	I_4	I_5	$\ldots$
Zeit / Takt	1	2	3	4	5	$\ldots$

c) Betrachten Sie folgende Maschinenbefehlssequenz eines RISC-Prozessors:

```
add $7,$8,$9    # Register 7 <---  Register 8 + Register 9
jr  $7          # Sprung zum Befehl, dessen Adresse in Register 7 steht
```

Inwieweit ist Pipelining bei dieser Befehlsfolge möglich?

Pipelining ist eingeschränkt möglich: Dekodierung des add-Befehls und Holen der jr-Befehls geht parallel, Ausführung der Addition durch ALU und Dekodieren von jr ebenfalls.

Treten Probleme auf? (Begründung)

Welche Probleme genau auftreten, hängt von der konkreten Pipelining-Architektur ab. Zwei Probleme sind offensichtlich:

- Wegen gemeinsamer Nutzung von Register 7 muss mit der Ausführungsphase der jr-Befehls gewartet werden, bis der add-Befehl das Ergebnis seiner Ausführung zurückgeschrieben hat.
- Wegen des unbedingten Sprungs zu einer Adresse ADR, die erst nach der Addition feststeht, ist es sinnlos, mit der Ausführung weiterer Instruktionen zu beginnen. Erst, wenn das Additionsergebnis ADR feststeht, kann die FETCH-Phase für den nächsten Befehl (mit der Adresse ADR) beginnen.

Aufgabe 3 (3+3+3 Punkte)

Punkte ☐ *von* **9**

a) Ein blockierter Prozess wartet auf ein Ereignis E_1, z.B. eine Tastatureingabe. Typischerweise wird dieser Prozess irgendwann wieder aufgeweckt werden um dann später erneut auf ein anderes Ereignis E_2 zu warten. Geben Sie ein Beispiel-Szenario an, in dem ein Prozess zunächst auf ein Ereignis E_1 und danach auf ein anderes Ereignis E_2 wartet, **ohne** dass er dazwischen aufgeweckt wird.

Beispiele für korrekte Antworten:

- Der Benutzer stoppt den blockierten Prozess mit einem Signal (SIGSTOP oder SIGTSTP). Jetzt muss dieser auf ein Continue-Signal (SIGCONT) warten.
- Ein Prozess P_1 wartet auf Tastatureingabe, wird aber dabei (über den Fenstermanager oder die Shell) in den Hintergrund verschoben. Jetzt sind die Tastatureingaben einem anderen Prozess P_2 zugeordnet und P_1 muss warten, bis er wieder in den Vordergrund kommt.
- Bei Hauptspeichermangel wird ein blockierter Prozess auf den Swap-Space ausgelagert. Jetzt muss er auf die Wiedereinlagerung warten.

b) Ein Betriebssystem unterscheidet bei Prozessen in Ausführung die Zustände „Running in user mode“ und „Running in kernel mode“. Welche Ereignisse können neben Systemaufrufen zu einem Übergang von dem ersten zu dem zweiten führen?

Beispiele:

- Der Prozess verursacht eine Prozessor-Ausnahme (z.B. unzulässiger Speicherzugriff, Division durch Null)
- Ein prozessbezogener Interrupt muss behandelt werden (z.B. Interrupt von der Tastatur, weil der Benutzer STRG-C gedrückt hat)
- Seitenfehler

c) Gibt es einen Übergang von „Running in user mode“ zu „Zombie“? (Begründung)

Beispiel für eine korrekte Antwort:

Nein. Die Terminierung eines Prozesses erfolgt durch einen Systemaufruf ($exit$) oder durch ein Signal. Eine Signalbehandlungsfunktion kann aber einen Prozess auch nur terminieren, indem sie die Kontrolle an den Systemkern übergibt, d.h. $exit$ aufruft. In einem solchen Zustandsmodell erfolgt jede Terminierung aus dem Zustand „running in kernel mode“.

Anmerkung: Das zugrunde liegende Zustandsübergangsmodell ist durch die Unterscheidung „Running in user mode“ und „Running in kernel mode“ nicht exakt definiert. Jedoch muss der Systemkern bei Terminierung eines Prozesses immer den Prozessdeskriptor ändern und Ressourcen freigeben, z.B. offene Datein schließen. Dies erfordert die Ausführung von Systemcode mit engem Bezug zum Prozess. Wird dieser Systemcode im Kontext des zu terminierenden Prozesses ausgeführt, wechselt dieser Prozess zur Terminierung immer in den Zustand „Running in kernel mode“.

Andere Modelle - und damit andere korrekte Antworten - sind denkbar: Auf die Begründung kommt es an!

Punkte ____ *von* **8**

Aufgabe 4 (3+5 Punkte)

Betrachten Sie das nachfolgende C-Programm

```
#include <stdio.h>
#include <unistd.h>
int main(){
   pid_t pid;

   printf("%d, %d\n", getpid(), getppid());
   pid=fork();
   printf("%d, %d, %d\n", pid, getpid(), getppid());
}
```

Bei der Ausführung lautet die erste Ausgabezeile:

```
629, 621
```

a) Von welchen Faktoren hängt der Rest der Ausgabe des Programms ab?

Offensichtliche Faktoren sind:

- Erfolg oder Fehler beim fork-Aufuf
- Welche PID erhält der neue Prozess?
- Eltern-/Kind-Reihenfolge der CPU-Zuteilung: Welcher der beiden Prozesse führt das zweite *printf* zuerst aus?

Diese Faktoren reichen als vollständige Lösung. Daneben gibt es aber noch weitere Möglichkeiten (mit Sonderpunkten bewertet), z.B.

- Wenn ein Prozess noch vor seinen Subprozessen terminiert, werden die Subprozesse vom init-Prozess, dessen PID 1 ist, adoptiert. Dies kann beide Prozesse aus dem Beispiel betreffen:
  1) Prozess 629 terminiert, bevor sein Subprozess *getppid* aufruft. Dann liefert *getppid* im Subprozess 1 statt 629 und die Ausgabe des Subprozesses ist
     ```
     0, Subprozess-PID, 1
     ```
  2) Prozess 621 terminiert, bevor 629 *getppid* aufruft. Dann wird 629 vom Prozess 1 adoptiert und gibt aus:
     ```
     Subprozess-PID, 629, 1
     ```
     oder, falls *fork* scheitert:
     ```
     -1, 629, 1
     ```
- Ist die Ausgabepufferung von Zeilen-orientiert auf Zeichen-orientiert umgestellt worden (hängt auch vom Ausgabegerät ab), können die Ausgaben von Kind- und Elternprozess theoretisch zeichenweise gemischt erscheinen.
- Jeder der beteiligten Prozess kann durch Signale abgebrochen werden, bevor es zur Ausgabe kommt. Dann fehlen die entsprechenden Ausgaben.

b) Geben Sie die weiteren Ausgaben des Programms für verschiedene Ausführungs-Szenarien an?

Für die offensichtlichen Szenarien:

1) fork schlägt fehl: Der Elternprozess gibt aus -1, 629, 621
2) fork funktioniert: Der Elternprozess gibt aus: Subprozess-PID, 629, 621
   Der Kindprozess gibt aus: 0, Subprozess-PID, 629
   Die Reihenfolge dieser beiden Ausgaben hängt von Scheduler ab.

Für die speziellen Szenarien, siehe bei (a).

Punkte ____ *von* **12**

Aufgabe 5 (12 Punkte)

Betrachten Sie folgenden Shell-Befehl:

```
od -x /bin/bash | grep ^00000  > pipe.out
```

Schreiben Sie ein C-Programm, das in vergleichbarer Weise die Programme `od` und `grep` in einer Pipeline kombiniert und die Ausgabe nach `pipe.out` umlenkt. Bei Fehlschlag von Systemaufrufen soll eine Fehlermeldung erscheinen, die einen Rückschluss auf die Fehlerursache erlaubt.

Die Aufgabe soll nicht an eine Shell (bzw. einen sonstigen Interpretierer) delegiert werden, wie z.B. mit
`system("od -x /bin/bash | grep ^00000  > pipe.out")`

```c
#include <stdio.h>
#include <wait.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>

int main(){
  pid_t pid_grep;
  int pipefd[2];
  int outfilefd;

  if(pipe(pipefd)){ perror("Fehler bei pipe"); exit(1); }

  // grep wird im Subprozess ausgeführt
  switch(pid_grep=fork()){
  case -1:
    perror("Fehler bei grep fork");
    exit(1);
  case 0:

    // grep liest aus der Pipe
    // Umlenkung der Eingabe
    if(dup2(pipefd[0],0)<0){
      perror("dup2-Fehler bei pipe eingabe");
      exit(1);
    }
    close(pipefd[0]);
    close(pipefd[1]);

    // grep schreibt nach "‘pipe.out"’
    // Datei öffnen und Ausgabe umlenken
    if ((outfilefd = open("pipe.out", O_WRONLY|O_CREAT|O_TRUNC, 0644)) < 0){
      perror("open-Fehler für pipe.out");
      exit(1);
    }
    if(dup2(outfilefd,1)<0){
      perror("dup2-Fehler bei pipe.out");
      exit(1);
    }

    execlp("grep", "grep", "^00000", NULL);
    perror("exec-Fehler bei grep");
    exit(1);
  }

  // od wird im Elternprozess ausgeführt um 2. fork zu sparen
  if(dup2(pipefd[1],1)<0){
    perror("Fehler bei Ausgabeumlenkung für od");
    exit(1);
  }

  close(pipefd[0]);
  close(pipefd[1]);

  execlp("od", "od", "-x", "/bin/bash", NULL);
  perror("exec-Fehler bei od ");
  exit(1);
}
```

Aufgabe 6 (2+4+4+4+4 Punkte)

Punkte ____ *von* **18**

Ein System verwendet virtuellen Speicher mit Paging. Für die Berechnung der 32 Bit großen realen Adressen wird eine zweistufige Seitentabelle benutzt. Eine virtuelle Adresse ist 31 Bit groß und von der Form

p_1	p_2	D

mit folgender Aufteilung:

p_1: 5 Bit für den Seitentabellenindex der 1. Stufe
p_2: 10 Bit für den Seitentabellenindex der 2. Stufe
D: 16 Bit für die Distanz

Angenommen, die virtuelle Adresse *v* adressiert das 10. Byte der 3. Seite eines Prozesses, die im 7. Rahmen des Hauptspeichers steht.

a) Wie groß ist ein Seitenrahmen? 2^{16} Byte

b) Geben Sie v und r hexadezimal an. v = 0x20009 r = 0x60009

c) Geben Sie an, wie der TLB-Eintrag zu v und r aussieht: (2 $\longrightarrow$ 6)

d) Wenn der virtuelle Adressraum eines Prozesses 512 Seiten groß ist, wieviele Seitentabellen gibt es dann für diesen Prozess in jeder Stufe?

Anzahl der Seitentabellen in der 1. Stufe: 1 in der 2. Stufe: 1

Anmerkung: In der 1. Stufe gibt es immer nur eine Seitentabelle, in der 2. Stufe nur die tatsächlich benötigten Tabellen. Da der Prozess nur die Seiten 0-511 benutzt, die erste Seitentabelle der 2. Stufe aber 2^{10} Einträge hat und damit die Rahmenadressen der Seiten 0 bis $2^{10} - 1 = 1023$ speichern kann, benötigt man keine weitere Seitentabelle.

e) Was wäre ein geeigneter Speicherort für die Seitentabelle der ersten Stufe? (Begründung)

Normalerweise stehen Seitentabellen im Hauptspeicher, die Berechnung realer Adressen ist wegen der beiden Seitentabellen-Lookups langsam. Durch Caching im TLB wird dies abgemildert. Da die Seitentabelle der ersten Stufe hier aber nur $2^5 = 32$ Einträge hat, sollte man die MMU mit 32 Registern zur Speicherung der kompletten Tabelle ausstatten. Dies erspart bei jedem TLB-Miss einen Hauptspeicherzugriff.

Rechenbeispiel: TLB Hitrate 95%, TLB-Zugriffszeit 1 Takt, Hauptspeicherzugriffzeit 10 Takte. Berechne durchschnittliche Zeit für die Adressumrechnung bei 100 Hauptspeicherzugriffen (ohne die Seitentabellen-Lookups):

Fall 1: Seitentabelle Stufe 1 im Hauptspeicher
95 TLB Hits, jeweils 1 Takt, 5 TLB-Misses, jeweils $2 * 10$ Takte für die beiden Seitentabellen-Lookups. Gesamtdauer: $95 + (5 * 2 * 10)$ Takte = 195 Takte.

Fall 2: Seitentabelle Stufe 1 im Registersatz der MMU mit 1 Takt Zugriffszeit
Bei TLB-Miss jetzt nur 11 Takte statt 20. Gesamtdauer: $95 + (5 * 11)$ Takte = 150 Takte. Die durchschnittliche Adressumrechnungszeit ist ca. 25% geringer!

Aufgabe 7 (8 Punkte)

Punkte ____ *von* **8**

Ein Scheduler richtet sich bei der CPU-Vergabe nach den Prioritäten der Threads: Ein Thread darf nicht ausgeführt werden, wenn ein anderer mit höherer Priorität auf CPU-Zuteilung wartet. Die Prioritäten werden dynamisch berechnet und können jederzeit durch Benutzereingiffe geändert werden.

Welche Umstände können hier zu einem Kontextwechsel führen? Ergänzen Sie die Liste!

1. Der ausführende Thread muss auf irgendein Ereignis warten und blockiert.
2. Der ausführende Thread terminiert durch Signal oder mit *exit*.
3. Der ausführende Thread hat sein Quantum verbraucht und es gibt einen anderen Thread mit der gleichen Priorität, der auf den Prozessor wartet.
4. Ein neuer Thread mit höherer Priorität wird erzeugt.
5. Ein anderer Thread mit höherer Priorität, der blockiert war, wird aufgeweckt.
6. Der Benutzer senkt die Priorität des ausführenden Threads.

7. Der Benutzer erhöht die Priorität eines anderen Threads, der auf den Prozessor wartet.

Anmerkung: Die vorgegebene Antwort 1 der Liste umfasst beliebige Blockade-Möglichkeiten für den ausführenden Thread, z.B. Seitenfehler, Stoppen durch den Benutzer mit CTRL-Z, Lesen von der Tastatur. Spezialfälle und Beispiele zu 1 werden daher nicht extra bewertet.

Punkte ____ *von* **11**

Aufgabe 8 (3+4+4 Punkte)

a) Wozu enthält bei einem UNIX-Dateisystem jeder Inode einen Referenzzähler ?

Wenn der Referenzzähler 0 ist, kann die Datei (Inode+Datencluster) gelöscht werden. Ohne den Referenzzähler kann dies nur mit erheblichen Aufwand erkannt werden, man bräuchte einen Garbage-Collector für die Dateien.

b) Ein C-Programm, das auf einer UNIX-Plattform läuft, versucht eine Datei zum Lesen zu öffnen:

```
fd = open("/tmp/datei.txt", O_RDONLY, 0);
```

Der Systemkern muss dazu zunächst den Pfad Schritt für Schritt verarbeiten und dabei unter anderem diverse Zugriffsrechtsprüfungen durchführen. Geben Sie ausgehend vom I-Node des Wurzelverzeichnisses alle Teilschritte dieser Pfadverarbeitung an.

1) Leseberechtigung für Wurzelverzeichnis prüfen
2) Im Inhalt des Wurzelverzeichnisses nach „tmp“ und der zugehörigen I-Node-Nummer suchen
3) I-Node von „/tmp“ einlesen
4) Prüfen, ob „/tmp“ ein Verzeichnis ist.
5) Leseberechtigung für „/tmp“ prüfen
6) Im Inhalt des „tmp“-Verzeichnisses nach „datei.txt“ und der zugehörigen I-Node-Nummer suchen
7) I-Node von „/tmp/datei.txt“ einlesen
8) Lesezugriff auf „/tmp/datei.txt“-Verzeichnis prüfen

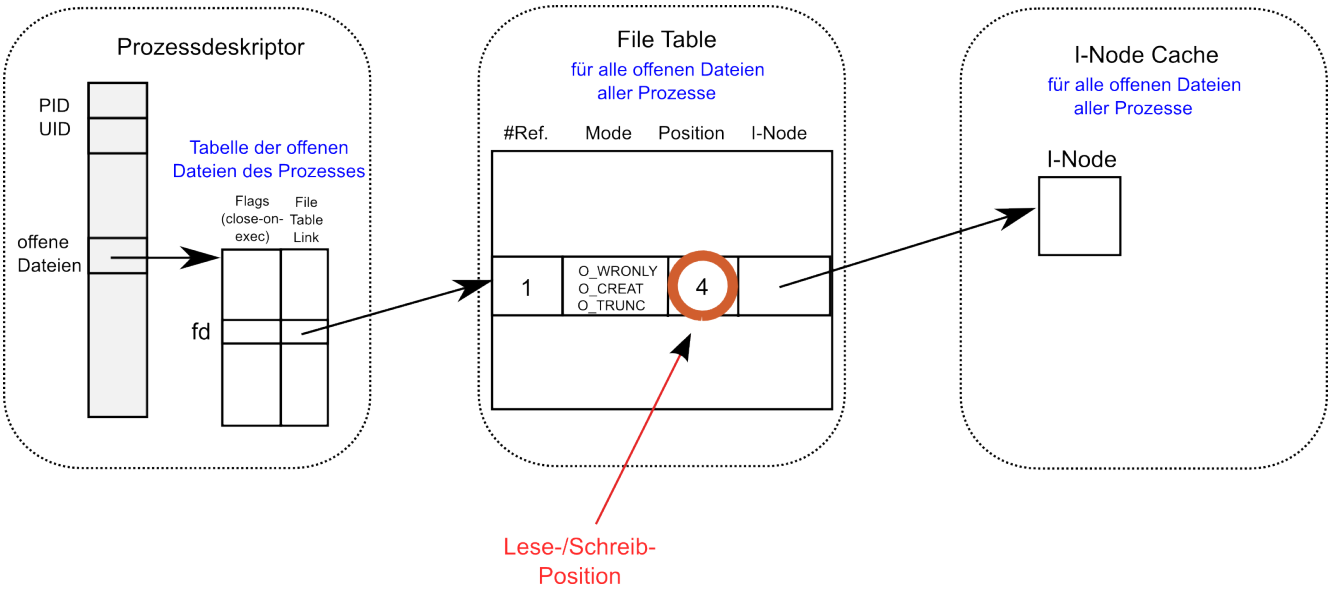
Anmerkung: Bei jeder Pfadkomponente muss in der Mount-Tabelle noch geprüft werden, ob mit dem Wurzelverzeichniss eines montierten anderen logischen Gerät weitergemacht werden muss. Ebenso können die Pfadkomponenten symbolische Links sein, die dann dereferenziert werden müssen. Dies wird am Dateityp im I-Node erkannt.

c) Nach dem erfolgreichen Öffnen gemäß (b) führt das Programm zwei Lesezugriffe aus:

```
read(fd, buffer, 4);
read(fd, buffer, 4);
```

Obwohl im `read`-Aufruf keine Leseposition angegeben ist, werden im ersten Aufruf die ersten 4 Byte der Datei gelesen und im zweiten Aufruf die dahinter stehenden nächsten 4 Byte. Geben Sie mit Hilfe einer Skizze an, welche Datenstrukturen für geöffnete Dateien hier im Spiel sind und wo die aktuelle Leseposition abgespeichert ist.

Für jeden Prozess hat der Systemkern eine Tabelle für die offenen Dateien (auch für Pipes und Sockets verwendet). Die von *open* zurückgelieferte Deskriptornummer *fd* ist der Index für diese Tabelle. In jedem Tabelleneintrag steht (neben dem close-on-exec-Flag) ein Verweis auf einen Eintrag in der prozessübergreifenden „file table“. In diesem Eintrag steht die aktuelle E/A-Position. Skizze:



Anmerkung: Die obige Grafik enthält Details, die für die vollständige Beantwortung der Frage nicht notwendig sind. Es genügt eine einfache Darstellung aus der hervorgeht, dass der Eintrag fd der Prozess-Deskriptoren-Tabelle auf einen Filetable-Eintrag verweist, der die Position enthält.