Prof. Dr. Michael Jäger FB MNI

Lösungsvorschlag zur Klausur “Betriebssysteme” vom 29.3.2017

Aufgabe 1 (2+2+1+2+3 Punkte)

Punkte ____ *von* **10**

a) Wie wird bei einem Von-Neumann-Rechner das Befehlsregister genutzt?

In der FETCH-Phase der Befehlsauführung wird der nächste Befehl aus dem Hauptspeicher in das Befehlsregister kopiert.

b) Was ist ein Befehlszähler (Programmzähler) und wie wird er genutzt?

Spezialregister der CPU, enthält immer die Adresse der nächsten auszuführenden Maschineninstruktion. Sprungbefehle kopieren das Sprungziel hinein, bei anderen Befehlen wird es automatisch um die Befehlsgröße inkrementiert.

c) Zu welcher Prozessorkomponente gehören Befehlszähler und Befehlsregister?

Zum Steuerwerk

d) Was ist der „Von Neumann-Flaschenhals“?

Datentransfer zwischen zwei Rechnerkomponenten erfolgt über den nur exklusiv nutzbaren Systembus. Insbesondere bei mehreren Prozessoren kommt es durch häufige Hauptspeicherzugriffe zu Ausführungsverzögerungen durch Warten auf den Bus.

e) Wenn ein Programm ein Byte aus dem Hauptspeicher liest, wird nicht nur dieses Byte, sondern ein ganzer Byte-Block in den Prozessor-Cache kopiert. Wozu wird dieser Mehraufwand betrieben?

Programme greifen oft kurze Zeit später auf weitere Bytes aus dem selben Block zu, z.B. bei der Verarbeitung eines Strings. Dann ist kein weiterer Hauptspeicherzugriff mehr nötig, weil die Daten schon im Cache stehen.

Aufgabe 2 (6 Punkte)

Punkte ____ *von* **6**

Ein Programm ruft ein anderes Programm auf und verwendet dazu den Systemaufruf

```
execl("bin/p", "p", NULL)
```

Beschreiben Sie in Stichworten, wie der Systemkern diesen Programmaufruf implementiert. Hinweise: Geben Sie zuerst an, aus welchen Einzelschritten die Verarbeitung des Dateipfads `"bin/p"` besteht. Welche weiteren Aktionen sind nach dem Auffinden der Programmdatei nötig, um das Programm zur Ausführung zu bringen? Was passiert mit dem Prozessdeskriptor und dem Programmspeicher?

Pfadverarbeitung: Für jedes Verzeichnis im Pfad wird anhand des I-Node die Zugriffsberechtigung überprüft und dann im Datencluster der Verzeichnisdatei durch Stringvergleich nach der nächsten Pfadkomponente gesucht. Der passende Eintrag enthält die I-Node-Nummer dazu. Im Beispiel wird dies für das aktuelle Verzeichnis und das Verzeichnis `./bin` gemacht

(nicht für das Wurzelverzeichnis, denn es handelt sich um einen relativen Pfad!)

Programm laden: Wenn `./bin/p` ausführbar ist, wird das Programm geladen. Dabei wird der aktuelle Programmspeicher des Prozesses freigegeben, im Hauptspeicher Platz für das neue Programm reserviert, das Codesegment, die statischen Daten, Heap und Stack gemäß der Programmdatei initialisiert. Der Prozessdeskriptor wird entsprechend angepasst. Der Programmzähler wird auf die erste Instruktion gesetzt.

Wichtig: Es ist kein neuer Prozess im Spiel!

Aufgabe 3 (4+2+2+4 Punkte)

Punkte ____ von **12**

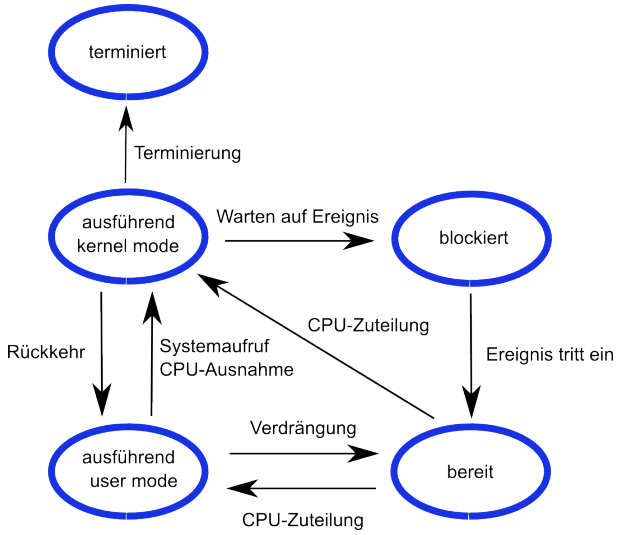
In einem Zustandsmodell für Threads werden folgende Zustände unterschieden: *bereit*, *ausführend im Anwendermodus*, *ausführend im Kernelmodus*, *blockiert* und *terminiert*.

Dabei ist ein Thread immer dann im Kernelmodus, wenn er gerade eine Funktion des Systemkerns ausführt.

a) Zeichnen Sie ein geeignetes Zustandsübergangsdiagramm

Es gibt hier nicht nur eine richtige Lösung. Wichtig ist aber folgendes:

1) Zu einem Zustandsübergangsdiagramm gehören auch vernünftige Kantenbeschriftungen, aus denen man sieht, wodurch die Zustandsübergänge ausgelöst werden.
2) Ein Thread blockiert typischerweise durch einen Systemaufruf (z.B. read), also im Zustand „*ausführend im Kernelmodus*“. Wenn das erwartete Ereignis eintrifft, muss der blockierende Systemaufruf zu Ende ausgeführt werden, also: Übergang nach „*bereit*“ und dann wieder nach „*ausführend im Kernelmodus*“.
3) Terminierung ist eine Metaoperation: Ein Thread terminiert immer im Kernelmodus
4) Ob im Kernelmodus eine Verdrängung möglich ist, hängt vom System ab. Der Übergang von „*ausführend im Kernelmodus*“ durch Verdrängung in „*bereit*“ ist also optional. Im Diagramm unten ist er weggelassen.



b) Welche Typen von Ereignissen bewirken einen Übergang vom Anwendermodus in den Kernelmodus?

Systemaufrufe und CPU-Ausnahmen (Division durch 0, Seitenfehler usw.).

c) Ist der Zustand „blockiert“ dem Kernelmodus oder dem Anwendermodus zuzurechen (Begründung)?

Kernelmodus, siehe (a), Punkt 1

d) Betrachten Sie folgendes Programm:

```
int main(){
  int i=0, j;
  for (i=500000; i>=0; i--)
     j=10/i;
}
```

Bei der Ausführung wird das Programm eine Zeit lang rechnen und dann beim letzten Schleifendurchlauf wegen der Division durch 0 abbrechen.

Beschreiben Sie, welche Zustände der zugehörige Prozess bzw. Thread bei einem typischen Ablauf annehmen wird und wodurch die Zustandswechsel ausgelöst werden.

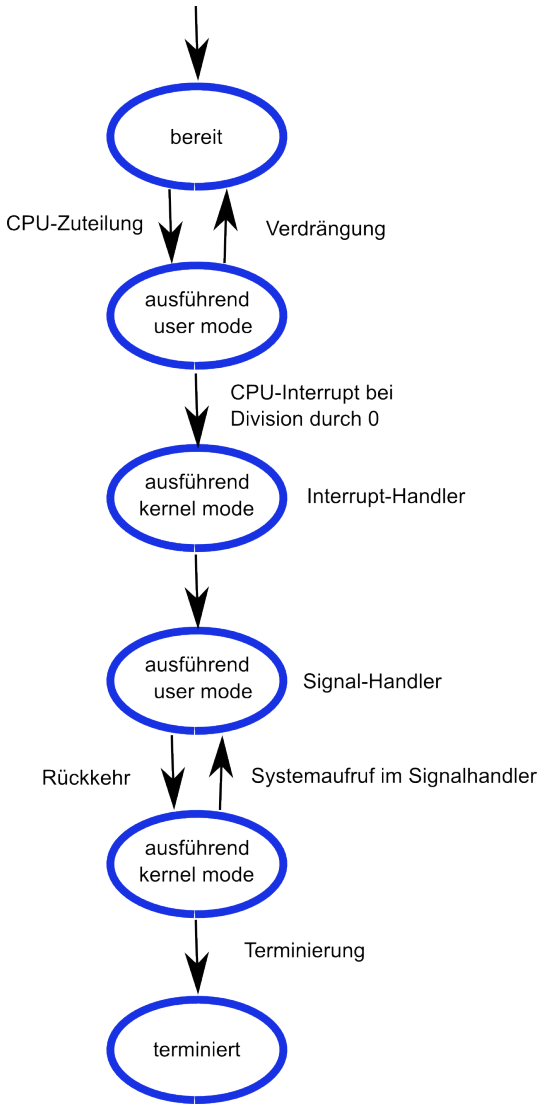
Vorabüberlegungen:

- Blockaden: Das Programm enthält keine blockierenden Systemaufrufe. Trotzdem kommt es zu Blockaden: Blockierende write-Aufrufe im Rahmen der Signalbehandlung nach der Division durch Null und Warten auf das Laden von Speicherseiten bei Seitenfehlern sind möglich (s. unten).
- Verdrängung: Das Programm läuft bei 500000 Schleifendurchgängen lange genug, um einige Male verdrängt zu werden. Also: diverse Übergänge von „*bereit*“ zu „*ausführend Anwendermodus*“ und zurück.
- Signalbehandlung: Signalhandler sind keine Funktionen des Systemkerns, sondern der Anwendungsebene: „*ausführend Anwendermodus*“. In der Signalbehandlung werden oft Systemaufrufe gemacht. Dabei wird in den Kernelmodus und zurück gewechselt.
- Division durch 0: Am Ende erfolgt im Zustand „*ausführend Anwendermodus*“ die Division durch Null, die zu einem Interrupt durch den Prozessor führt.
- Die Interrupt-Behandlung erfolgt im Zustand „*ausführend im Kernelmodus*“. Der Interrupt-Handler generiert ein SIGFPE-Signal für den Prozess, sorgt für den Aufruf des Signalhandlers und gibt die Kontrolle wieder an die Anwendung zurück. Diese ist jetzt „*ausführend im Anwendermodus*“.

  Da kein eigener Signalhandler registriert ist, erfolgt die Standardbehandlung: Fehlermeldung, Core-Dump, Terminierung des Prozesses. Dazu wird der Signalhandler diverse Systemaufrufe benötigen. Die Terminierung ist eine Systemoperation, dazu wechselt der Prozess am Ende in den Zustand „*ausführend Kernelmodus*“ und dann nach „*terminiert*“.
- Seitenfehler: Während der Ausführung kann es im Zustand „*ausführend Anwendermodus*“ (ggf. als auch im Kernelmodus) zu Seitenfehlern kommen. Die Interruptbehandlung erfolgt immer im Kernelmodus. Falls eine Seite vom Datenträger nachgeladen werden muss, blockiert der Prozess dabei. Dieser Aspekt ist im nachfolgenden Diagramm nicht dargestellt.

Beachte: Signalbehandlung führt nicht zur Blockade, Signalbehandlung erfolgt prinzipiell im Anwendermodus, erst durch Systemaufrufe im Signalhandler ist der Systemkern beteiligt.

Wie man sieht, ist die Analyse trotz der Einfachheit des Beispiels recht komplex. Eine vollständig korrekte Lösung ist daher für die volle Punktzahl auch nicht nötig!



Aufgabe 4 (2+2+6 Punkte)

Punkte ____ *von* **10**

Betrachten Sie den UNIX-Shell-Befehl `ftp < ftp.in | tee ftp.out`

(tee kopiert die Standardeingabe in die Standardausgabe und zusätzlich in die angegebene Datei)

a) Welche Prozesse sind an der Ausführung beteiligt und wie ist deren Verwandschaftsverhältnis?

Die Shell und deren Kindprozesse ftp und tee.

b) Unter welchen Umständen kann es dazu kommen, dass *tee* blockiert? Welche Systemaufrufe können dabei blockieren?

tee kann im read-Aufruf (Pipe) blockieren, wenn die Pipe leer ist, außerdem in den write-Aufrufen für die Standardausgabe und die Ausgabedatei ftp.out, falls diese synchron sind (Normalfall).

c) Geben Sie ein C-Programm an, das diesem Shell-Befehl entspricht. Das Programm soll zum Aufruf von `ftp` und `tee` den Systemaufruf *execlp* verwenden. Vervollständigen Sie dazu das nachfolgende Programmskelett. Lassen Sie dabei jegliche Fehlerbehandlung und die `#include` -Anweisungen weg.

```
int main(){
    pid_t ftppid, teepid;
    int pipefd[2];
```

```
    pipe(pipefd);
    switch (ftppid=fork()){
    case 0:
       /* ftp: Eingabe aus ftp.in, Ausgabe in die Pipe */
       /* Eingabeumlenkung */
       {
         int fd=open("ftp.in", O_RDONLY,0);
         dup2(fd,0);
         close(fd);
       }
       /* Ausgabeumlenkung */
       dup2(pipefd[1],1);
       close(pipefd[1]);

       /* Verklemmung vermeiden */
       close(pipefd[0]);

       execlp("ftp", "ftp", NULL);
   }

    switch (teepid=fork()){
    case 0:
       /* tee: Eingabe aus Pipe */

       /* Eingabeumlenkung */
       dup2(pipefd[0],0);
       close(pipefd[0]);

       /* Verklemmung vermeiden */
       close(pipefd[1]);

       execlp("tee", "tee", "ftp.out", NULL);
   }

   /* Verklemmung mit den Kindern vermeiden */
   close(pipefd[0]);
   close(pipefd[1]);

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

Aufgabe 5 (3+3 Punkte)

Punkte ____ *von* **6**

Der Befehl

```
kill -TERM 1234
```

bewirkt die Generierung des Signals SIGTERM für den Prozess mit der PID 1234. Die Wirkung des Befehl hängt von verschiedenen Einstellungen ab.

a) Welche Konsequenzen sind möglich und wovon hängt die Wirkung ab?

Die Wirkung hängt zunächst von der Signalbehandlung durch den Empfängerprozess ab. Dieser kann eine eigene Signalbehandlung vorsehen, das Signal ignorieren oder die voreingestellte Standardbehandlung, Terminierung des Prozesses, belassen. Außerdem kann er die eingestellte Signalbehandlung mittels Signalmaske vorübergehend blockieren. Natürlich müssen auch die Rechte zur Terminierung vorhanden sein, ein Benutzer kann nicht einfach Prozesse eines anderen terminieren.

b) Wenn der Signalempfänger das Signal erhält, könnte er gerade Anwendungscode oder aber einen Systemaufruf ausführen. Spielt das für die Signalbehandlung eine Rolle (Begründung)?

Es spielt eine Rolle: Anwendungscode wird durch die Signalbehandlung unterbrochen, Systemaufrufe dagegen nicht. Langsame Systemaufrufe werden abgebrochen, schnelle werden zuerst komplett abgearbeitet.

Aufgabe 6 (2+2+2 Punkte)

Punkte ____ *von* **6**

a) Wie und das System die Clusternummer des 2. Clusters einer Datei bei einem FAT-Dateisystem?

Die Nummer des ersten Clusters steht im Verzeichniseintrag der Datei. Sei diese Nummer *n*. Dann steht die Nummer des 2. Clusters im *n*-ten Eintrag der FAT (File Allocation Table).

b) Bei einem klassischen UNIX-Dateisystem wird für jedes Datencluster einer Datei ein Verweis benötigt. Bei Windows NTFS und modernen UNIX-Dateisystemen benutzt man stattdessen Extents. Was ist ein Extent? Wo ist der Vorteil von Extent-basierter Verwaltung der Datencluster?

Extent = aus mehreren Clustern bestehender zusammenhängend gespeicherter Teilbereich einer Datei

Dateisysteme sind normalerweise nicht sehr stark fragmentiert, so dass es sehr viel weniger Extents gibt, als Datencluster. Im Dateideskriptor sparen wenige Verweise auf die Extents der Datei viel Platz gegenüber vielen Verweisen auf die einzelnen Cluster und man kann durch Verzicht auf indirekte Verweise schneller die Cluster der Datei bestimmen.

c) Was ist Metadaten-Journaling bei einem Dateisystem und welchen Vorteil hat es?

Dateioperationen mit Änderungen an den Metadaten (Verzeichnisse, Dateideskriptoren, usw.) werden vor der Ausführung und nach Abschluss im Journal festgehalten. Nach einem Systemabsturz kann man im Journal sehr schnell die nicht vollständig ausgeführten Operationen finden und Inkonsistenzen beheben, was ohne Journal sehr aufwändig wäre.

Aufgabe 7 (2+2+2+2+2+2 Punkte)

Punkte ____ *von* **12**

Ein System verwendet virtuellen Speicher mit Paging. Für die Berechnung der 48 Bit großen realen Adressen wird eine zweistufige Seitentabelle benutzt. Eine virtuelle Adresse ist 32 Bit groß und von der Form | p_1 | p_2 | D | mit folgender Aufteilung:

p_1: 8 Bit für die Seitentabelle der 1. Stufe
p_2: 8 Bit für die Seitentabelle der 2. Stufe
D: 16 Bit für die Distanz

Angenommen, die virtuelle Adresse *v* adressiert das 12. Byte der 17. Seite eines Prozesses, die im 11. Rahmen des Hauptspeichers steht. Zur Ermittlung der realen Adresse *r* werden 2 Seitentabellen benötigt.

a) Wo genau steht die Adresse der 2. benötigten Seitentabelle?

Im ersten Eintrag der Seitentabelle erster Stufe. Eine Seitentabelle der 2. Stufe hat 2^8=256 Einträge. Die 1. Seitentabelle der 2. Stufe enthält also die Rahmennummern der Seiten 0-255, somit auch die im Beispiel benötigte Rahmennummer der Seite 16. Der erste Eintrag in der Haupt-Seitentabelle zeigt auf den Anfang der 1. Seitentabelle 2. Stufe.

b) Geben Sie die virtuelle Adresse dieses Bytes hexadezimal an:

0x10000B

c) Geben Sie die reale Adresse dieses Bytes hexadezimal an:

0xA000B

d) Geben Sie an, wie der zugehörige TLB-Eintrag aussieht:

0x10 $\rightarrow$ 0xA oder dezimal : 16 $\rightarrow$ 10

e) Was ist Demand-Paging?

Eine Hauptspeicher-Einlagerungsstrategie, bei der erst dann zu einer Seite ein Rahmen zugeordnet wird, wenn der Prozess auf die Seite zugreift.

f) Nennen Sie ein paar Vorteile von virtuellem Speicher mit Demand-Paging.

Programme können logisch größer sein als der reale Speicher. Weniger Rahmen werden pro Programm benötigt, so dass mehr Prozesse in den Hauptspeicher passen. Das Laden von Programmen geht schneller, da für den Start der Ausführung nur wenige Seiten benötigt werden.