Vorlesungsmodul Betriebssysteme 1
- VorlMod BetrSys1 -

Matthias Ansorg

30. September 2002 bis 13. Januar 2003

Zusammenfassung

Studentische Mitschrift zur Vorlesung Betriebssysteme 1 bei Prof. Jäger (Wintersemester 2002/2003) im Studiengang Informatik an der Fachhochschule Gießen-Friedberg.

- **Bezugsquelle:** Die vorliegende studentische Mitschrift steht im Internet zum Download bereit: `http://homepages.fh-giessen.de/~hg12117/index.html`. Wenn sie vollständig ist, kann es auch über die Skriptsammlung der Fachschaft Informatik der FH Gießen-Friedberg `http://www.fh-giessen.de/FACHSCHAFT/Informatik/cgi-bin/navi01.cgi?skripte` downgeloadet werden.
- **Lizenz:** Diese studentische Mitschrift ist public domain, darf also ohne Einschränkungen oder Quellenangabe für jeden beliebigen Zweck benutzt werden, kommerziell und nichtkommerziell; jedoch enthält sie keinerlei Garantien für Richtigkeit oder Eignung oder sonst irgendetwas, weder explizit noch implizit. Das Risiko der Nutzung dieser studentischen Mitschrift liegt allein beim Nutzer selbst. Einschränkend sind außerdem die Urheberrechte der verwendeten Quellen zu beachten.
- **Korrekturen:** Fehler zur Verbesserung in zukünftigen Versionen, sonstige Verbesserungsvorschläge und Wünsche bitte dem Autor per e-mail mitteilen: Matthias Ansorg, ansis@gmx.de.
- **Format:** Die vorliegende studentische Mitschrift wurde mit dem Programm LyX (graphisches Frontend zu LaTeX) unter Linux erstellt und als pdf-Datei exportiert. Grafiken wurden mit dem Programm xfig unter Linux erstellt und als eps-Datei exportiert.
- **Dozent:** Prof. Jäger.
- **Verwendete Quellen:** .
- **Klausur:**

Inhaltsverzeichnis

Teil I
Lernstoff

1 Organisatorisches

1.1 Zeiten

4h Vorlesung, 2h Übungen.

Vorlesung: Mo 8:00h C130

Vorlesung: Fr 9:30h G3

Übung: Mo 14:00h F209 alternativ 15:45 F209; alternativ Di 08:00h F209 alternativ 09:50h; alternativ Di 09:50h C 403 (Letschert).

Übungen beginnen erst ab 14.10.

1.2 Klausur

Di, 2003-01-21 (auf jeden Fall vor den Semesterferien)
Es sind keine Unterlagen erlaubt!

1.3 Hausübungen

Gemäß Prüfungsordnung sind 2 Hausübungen Prüfungsvoraussetzung. Wer die Klausur wiederholt, muss keine Hausübungen mehr beibringen. Eine der Hausübungen wird die Implementierung eines Kommandointerpreters sein.

1.4 Skript

Alle Unterlagen finden sich in Form von pdf-Dateien auf der Homepage von Prof. Jäger http://homepages.fh-giessen.de/~hg52. Das Skript enthält alles, was man zur Lehrveranstaltungen und die Klausur braucht. Auch die Übungsaufgaben stehen dort, ebenso alte Klausuren und weitere Begleittexte.

1.5 Literatur

Das Skript enthält ein Literaturverzeichnis.

1.6 Praktikum

Es wird an UNIX-Rechnern gearbeitet, d.h. an den Terminals des Sun-Workgroup-Servers in F209, F211, C403. Es gibt BSD-Unix im Quellcode! Das im Praktikum verwendete Betriebssystem ist also Sun Solaris. Zu Hause kann man die Übungsaufgaben mit einem anderen Unix-System oder Linux bearbeiten. Die Aufgaben können auch unter Windows mit der kostenlosen Unix-Entwicklungsumgebung Cygwin http://www.cygwin.com bearbeitet werden.

Die Aufgabenstellungen zum Praktikum werden im Internet auf der Homepage von Prof. Jäger veröffentlicht.

Probleme nach Erfahrungen aus bisherigen Jahren: Die Grundkenntnisse zum Umgang mit dem Unix-System sollte man schnell beherrschen (Login, Quelltext editieren, Linken, Makefiles schreiben, Debugging). Die erste Übungsstunde beinhaltet lediglich dies als Stoff. Einige werden noch Nachholbedarf an Kenntnissen in C und C++ haben. Mit allen Problemen darf man die Tutoren nerven. Nur die wenigsten werden das Praktikum zu Hause alleine, ohne Hilfe, durchziehen können - man sollte das Praktikum also besuchen.

Man kann alle Probleme auch mit Prof. Jäger in der Sprechstunde oder per e-mail besprechen.

1.7 Lernziele und Inhalte

1.7.1 Motivation

Betriebssysteeme ist im Informatikstudium ein wichtiges Fach. Warum? Weil:

- jeder Informatiker stets das Betriebssystem als Anwender benutzt.
- er als Programmierer die Programmierschnittstelle des Betriebssystems (API[1]) benutzt. Dies geschieht über Systemaufrufe (system calls).
- ein Informatiker das Betriebssystem als Administrator oder Netzwerkadministrator benutzt. Diese Aufgabe ist beliebig kompliziert. Im Informatikstudium erlernt man lediglich das Grundlagenwissen und ggf. eine Vertiefung im Hauptstudium (Vorlesung Systemtechnik).
- er ggf. ein Betriebssystem implementiert. Dies ist auch für Informatiker von der FH nicht besonders unwahrscheinlich. Es gibt kaum einen Menschen, der ein Betriebssystem im Detail versteht - das geht nur im Team.

[1] application programming interface

1.7.2 Inhalte allgemein

- Konzepte von Betriebssystemen. Dies geschieht im Wesentlichen plattformunabhängig, d.h. es werden die Gemeinsamkeiten der Betriebssysteme herausgestellt.
  - Architektur. Aus welchen Bestandteilen ist ein Betriebssystem aufgebaut?
  - Funktionen von Betriebssystemen.
  - Implementierungsstrategien. Wie programmiert man z.B. ein Dateisystem? Es gibt stets verschiedene technische Möglichkeiten zur Implementierung; als Informatiker sollte man die verschiedenen Techniken kennen und die beste auswählen können. Kriterien sind z.B. Dauer, Wiederverwendbarkeit.
- C/C++ - API des Betriebssystems
  - Prozesskontrollaufrufe. Dies wird sehr intensiv behandelt.
  - Dateisysteme.
  - Interprozesskommunikation.
    - Pipes.
    - Socket-Schnittstelle zur Netzwerkkommunikation.
- Kommandoschnittstelle. Es wird in der Hausübung ein Unix-Kommandointerpreter geschrieben. Vorteile gegenüber GUI:
  - Es steht eine wesentlich größere Funktionsfülle zur Verfügung. Viele Systemdienste sind von der GUI aus nicht benutzbar.
  - Benutzt man als »Poweruser« einen Rechner für sehr verschiedene Aufgaben, muss man effizient arbeiten. Man sollte komplexe Funktionen schnell zur Verfügung haben. Diese Automatisierung von Arbeitsabläufen kann sehr gut über Shell-Skripte der Kommandoschnittstelle realisiert werden. In der Vorlesung Betriebssysteme 1 wird der Kommandointerpreter bash gelernt.

1.7.3 Stoffüberblick

1. Einführung. Was ist ein Betriebssystem? Grundbegriffe. Arten von Betriebssystemen. Modelle zur Architektur von Betriebssystemen.
2. Prozesse und Threads. Dies sind zwei verschiedene Konzepte zur Nebenläufigkeit (concurrency).
3. Synchronisation nebenläufiger Aktivitäten. Notwendig, wenn nebenläufige Aktivitäten gekoppelt sind. Auch hinter Unix-Pipes sind solche Synchronisationsmechanismen zu finden!
4. Verklemmungen (deadlocks)
5. Speicherverwaltung. Ein komplexes Thema, weil viele verschiedene Programme gleichzeitig den Hauptspeicher benutzen. Diese Funktionalität ist schwierig, weil sehr stark auf Geschwindigkeitsanforderungen geachtet werden muss (ansonsten wäre das Thema nur 25% so komplitziert ...).
6. Prozessorvergabe (processor scheduling). Betrachtet werden Ein- und Mehrprozessorsysteme.
7. Dateisysteme. Partitionierung; Nutzung der Partitionen; Organisationsmöglichkeiten für Dateisysteme.
8. Ein- und Ausgabe. Gerätetreiber; wie organisiert das Betriebssystem Ein- und Ausgabe?; wie kapselt man hardwareabhängige Software in Treibern?.
9. Kommunikation im Netz. Kurze Behandlung des OSI-Schichtenmodells; wie ist die Benutzung von Netzwerkprotokollen im Betriebssystem verankert; Socket-API und ihre Programmierung. Die Socket-API ist eine Programmierschnittstelle, die unabhängig vom verwendeten Transportprotokoll (etwa TCP/IP, IPX/SPX, Pipes) ist.
10. Einführung in verteilte Systeme (sofern noch Zeit zur Verfügung steht). Eine einfache Möglichkeit ist die Client/Server-Architektur, eine komplexere Möglichkeit Middleware-Software wie CORBA.

2 Stoffzusammenfassung

Die folgende kurze Zusammenfassung des Lehrstoffs sollte man möglichst auswendig beherrschen, da in der Klausur keine Hilfsmittel benutzt werden dürfen. Die Zusammenfassung ist so aufbereitet, dass sie einfach auswendig gelernt werden kann. Hier sind nur Fakten enthalten, die man zum Lösen der (bisher verfügbaren) Aufgaben braucht; das außerdem notwenige Verständnis sollte man sich durch Lesen und Verstehen der Skripte aneignen.

Syntax und Standardbibliothek von C und C++ werden hier nicht aufgeführt, sondern vorausgesetzt. Syntax und Sematik der benötigten Systemfunktionen werden hier allerdings behandelt.

2.1 Prozessdeskriptor

Nach [2, S.18]. Der Eintrag zu einem Prozess in der Prozesstabelle. Er enthält:

PID Prozessidentifikationsnummer

Zustand Hauptsächlich »ausführend«, »bereit« oder »blockiert«.

Zugriffsrechtsdeskriptor Rechte des Prozesses an Ressourcen.

Dateideskriptoren Bei UNIX Verweise auf Einträge in die globale Dateitabelle.

Hauptspeicherdeskriptor Finden und Zugreifen auf die Speichersegmente des Prozesses.

Maschinenzustand Gesichert während Unterbrechungen: Register, Programmzähler, Statuswort usw.

Priorität

Ressourcenverbrauch Für Abrechnungs- und Kontrollzwecke.

2.2 Dateisystem

2.3 Include-Dateien

2.4 Signalbehandlung

Wichtige Signale (vgl. [6]) in `signal.h`:

SIGINT Prozessunterbrechung durch Tastatur. Standardaktion: Prozessterminierung.

SIGKILL Z.B. durch `kill pid`. Standardaktion: Prozessterminierung. Kann nicht aufgefangen oder ignoriert werden.

SIGPIPE Nach Schreiben in eine Pipe ohne Leser. Standardaktion: Prozessterminierung.

SIGCHLD Subprozess stoppte oder terminierte. Standardaktion: ignorieren.

```
#include <signal.h>   //for sigaction, SIGCHLD, sigemptyset
#include <stdio.h>    //for perror
#include <sys/wait.h> //for wait(int *status)
struct sigaction oldAction, newAction;
void chldHandler(int signum) { //SIGCHLD handler
  //we know that a child is now a zombie. By wait()
  //we make it terminate finally, without knowing the childs pid
  wait(0);
  //reset SIGCHLD handling, as the child terminated now:
  sigaction(SIGCHLD, &oldAction, NULL);
}
int main() {
  newAction.sa_handler = &chldHandler; //signal handler function
  newAction.sa_flags   = 0;
  sigemptyset(&newAction.sa_mask); //ignore no signals
  if (sigaction(SIGCHLD, &newAction, &oldAction) ==-1)
    perror("main: Fehler bei sigaction()");
}
```

2.5 Subprozesse mit `fork()`

Das folgende Beispiel basiert auf [3, S. 2-3].

```
#include <unistd.h> //for pid_t fork(), pid_t vfork(), getpid(), getppid
#include <stdio.h>  //for perror()
#include <types.h> //for pid_t
#include <stdlib.h> //for exit
#include <iostream>
using namespace std;

main(){
  pid_t chldPid, myPid; //in normal case pid_t = "unsigned long"
  cout << "Before fork(), PID= " << (myPid = getpid()) << endl;
  switch (chldPid=fork()) {
    case -1: perror("fork() failure\n"); exit(1);
      break;
    case  0: cout << "I'm child: PID= " << (myPid=getpid())
                  << ", parent PID= "   << getppid() << endl;
      break;
    default: cout << "I'm parent: after fork, child PID= "
                  << chldPid;
      break;
  }
  cout << "I'm child or parent: I finish work, my PID= "
       << myPid <<  endl;
  exit(0);
}
```

Ausgabe des Beispielprogramms etwa (vgl. [3, S. 3]):

```
Before fork(), PID= 453
I'm prent: after fork, child PID= 454
I'm child or parent: I finish work, my PID= 453
I'm child: PID= 454, parent PID= 1
I'm child or parent: I finish work, my PID= 454
```

2.6 Prozessterminierung mit `exit`

```
#include <stdlib.h>
void exit(int status); //0: normal termination; 1: error
```

Der Exitstatus eines Subprozesses kann der Elternprozess mit `wait` oder `waitpid` erfahren.

2.7 Warten auf Subprozesse mit `wait` und `waitpid`

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *status)
pid_t waitpid(pid_t pid, int *status, int options);
```

Codebruchstück: Warten auf Subprozesterminierung, Art der Terminierung bestimmen (aus [3, S. 9]):

```
// Warten auf Subprozess-Ende
waitpid(kind_pid, &kind_status, 0);
// normale Terminierung mit exit ?
if (WIFEXITED(kind_status))
  // normale Terminierung, exit-Argument ausgeben
  cout << "Subprozess-Status:" << WEXITSTATUS(kind_status) << endl ;
else if (WIFSIGNALED(kind_status)) // Terminierung durch Signal, welches ?
  cout << "Subprozess durch Signal abgebrochen, Signalnummer:"
       << WTERMSIG(kind_status) << endl;
```

2.8 Aufruf externer Programme mit `exec`

Auszug aus [7], weitere Informationen dort.

```
#include <unistd.h> //for exec function family
extern char **environ;
int execl( const char *path, const char *arg, ...);
int execlp( const char *file, const char *arg, ...);
int execle( const char *path, const char *arg , ..., char * const envp[]);
int execv( const char *path, char *const argv[]);
int execvp( const char *file, char *const argv[]);
```

2.9 Dateibehandlung

Verwende zum Lernen die Liste »Wichtige Systemaufrufe des Dateisystems« [4, S. 16].

2.10 Pipes und Ein-/Ausgabeumlenkung

Verwende zum Lernen die Kapitel »7.4.13 pipe – Öffnen einer unbenannten Pipe« und »7.4.14 dup – Dateideskriptor duplizieren« in [4, S. 19-22]. Kommentierter Code zur Verwendung von Pipes und Ein- Ausgabeumlenkung ist enthalten in der beiligenden Datei `Aufg.Klausur.2000-03-09.03.cc`.

2.11 Shellprogrammierung

Teil II
Lösungen

3 Übungsaufgaben

Die Aufgaben stammen aus der Veranstaltung »Betriebssysteme 1« bei Prof. Dr. Michael Jäger (FH Gießen-Friedberg) aus dem Wintersemester 2002/2003. Es werden nur die Aufgabenstellungen und Lösungen der nichttrivialen Aufgaben angegeben.

3.1 Skript: Peterson-Algorithmus

Es soll bewiesen werden, dass der Peterson-Algorithmus verklemmungsfrei ist [2, S. 46]. Definition von »Verklemmung« in [2, S. 63], Kapitel »5.2. Systemmodell«: »Eine Menge von Prozessen heißt verklemmt, wenn jeder Prozess in der Menge auf eine Ressource wartet, die von einem anderen Prozess in der Menge reserviert ist.« Der Peterson-Algorithmus wird hier nur zum gegenseitigen Ausschluss von 2 threads angewandt, auch der Beweis in [2, S. 46] beschränkt sich auf 2 Threads.

Beweis durch Widerspruchsbeweis. Dazu Annahme: der Peterson-Algorithmus ist nicht verklemmungsfrei, d.h. es kann einen Zeitpunkt geben, zu dem Thread P_1 in der busy-wait-Schleife auf P_2 wartet und gleichzeitig P_2 in der busy-wait-Schleife auf P_1 wartet. Dazu müssen nach dem Sourcecode folgende Bedigungen gleichzeitig erfüllt sein:

$$\begin{aligned} turn &= 1 && (1)\\ interested[1] &= true \\ turn &= 2 && (2)\\ interested[2] &= true \end{aligned}$$

Offensichtlich widerspricht Gleichung 1 Gleichung 2. Die vier Bedingungen können also nicht gleichzeitig wahr sein, also muss die Annahme falsch sein, die zu dieser Forderug führte, nämlich dass der Peterson-Algorithmus nicht verklemmungsfrei ist. Also ist der Peterson-Algorithmus verklemmnungsfrei.

3.2 Übungsblatt 1, Aufgabe 1 (Umgang mit der Shell)

b) »Verwenden Sie den "ps"-Befehl, um herauszfinden, welche Prozesse unter ihrer Benutzeridentidentifikation existieren und welche Prozessnummern diese Prozesse haben.« Man verwende `ps --User=my_username` und setze entsprechend seinen eigenen Benutzernamen ein.

c) »Verwenden Sie den "kill"-Befehl, um einen ihrer Prozesse zu terminieren.« Nachdem man mit `ps` die PID n des zu killenden Prozesses erfahren hat, verwendet man: `kill` n.

3.3 Übungsblatt 1, Aufgabe 2

3.4 Übungsblatt 1, Aufgabe 3

3.5 Übungsblatt 2, Aufgabe 1

- »Überlegen Sie zuhause: Kann es passieren, dass folgende Systemaufrufe fehlschlagen? Wenn ja, welche Gründe könnte es dafür geben?«

  - `fork`: nicht genügend Hauptspeicher frei zum Kopieren des Prozesses; maximale Anzahl der im Mutitasking parallel laufenden Prozesse erreicht.
  - `execlp`: angegebenes Programm `file` kann in den Pfaden der Umgebungsvariable `PATH` nicht gefunden werden; Datei `file` existiert zwar, ist aber kein Programm oder sie ist ein Programm,das aber vom effektiven Benutzer nicht ausgeführt werden darf; Liste der Argumente ist nicht `NULL`-terminiert.
  - `waitpid`: es existiert kein Prozess mit der übergebenen PID; Prozess mit der übergebenen PID terminiert nicht vor Terminierung des aufrufenden Prozesses, was z.B. für `init` (PID 1) gilt.

- »In welcher Weise kann im Programm das Scheitern eines Systemaufrufs überprüft werden? Wie erfährt man die Fehlerursache? Schlagen Sie dazu im Online-Manual die Erklärung von execlp nach.« Bei Fehlern kehren die `exec`-Funktionen mit einem Rückgabewert `-1` zurück (ansonsten kehrt keine der `exec`-Funktionen jemals zurück). Die globale Variable `errno` enthält dann die Nummer des aufgetretenen Fehlers; die Beschreibung des Fehlers gibt der Aufruf der Funktion `perror` aus.

- »Wieso ist es günstig, bei einem fehlgeschlagenen Systemaufruf die perror-Funktion zur Ausgabe der Fehlermeldung zu verwenden, anstelle von `cerr << ...` ?« Weil `perror` zusätzlich zum übergebenen Text eine Beschreibung des aufgetretenen Fehlers ausgibt, durch Auswertung der Fehlernummer in der globalen Variable `perror`.

3.6 Übungsblatt 2, Aufgabe 2

»Schreiben Sie ein Programm `aktiviere`, dass zwei neue Prozesse erzeugt, um nebenläufig 2 weitere Programme aufzurufen und diesen Parameter übergeben, nämlich `xterm -e top` und `netscape http://www.fh-giessen.de/WEB_MNI`. Ihr Programm aktiviere gibt zuerst für beide Programme die Prozessnummern auf den Bildschirm aus. Dann wartet es die Terminierung beider Prozesse ab und zeigt auf dem Bildschirm an, ob `xterm` oder `netscape` zuerst beendet wurde.«

Lösung enthalten in Kapitel 3.7.

3.7 Übungsblatt 2, Aufgabe 3

»Das im `xterm`-Fenster ablaufende Programm `top` kann mit `Ctrl-C` abgebrochen werden. Probieren Sie es aus. Erweitern Sie dann aktiviere wie folgt: `xterm` und damit auch dessen Kindprozess `top` sollen das Signal `SIGINT` ignorieren, das durch die `Ctrl-C`-Taste generiert wird. Dazu muss in dem Prozess eine entsprechende Signalbehandlung für dieses Signal vereinbart werden. Wenn in einem Prozess ein Signal ignoriert wird, bleibt diese Einstellung auch bei einem Wechsel in ein anderes Programm (z.B. mit execlp) bestehen. Sie können also die Signalbehandlung vor dem Aufruf von `xterm` in aktiviere vornehmen. Benutzen Sie dazu `sigaction`, das im Skript und im Onlinemanual erklärt ist.«

Die Lösung ist in der beiligenden Datei `Aufg.2.3.cc` enthalten.

3.8 Übungsblatt 3, Aufgabe 1

»Schreiben Sie ein Programm, in dem ein Subprozess erzeugt und eine Datei namens `begruessung` erstellt wird, mit folgenden Varianten:«

a) »Der Subprozess erstellt die Datei mit dem Inhalt »Guten Morgen!«. Der Elternprozess wartet, bis der Subprozess fertig ist, öffnet und liest die Datei und gibt den Inhalt auf den Bildschirm aus.«
Auf dem Bildschirm steht »Guten Morgen!« Der Kontrollfluss ist hier äquivalent zu einer Sequenz, in der die Datei erstellt, geschlossen und wieder geöffnet wird, es findet also niemals gleichzeitiger Dateizugriff statt.

b) »Der Subprozess schreibt in die Datei »Guten « und terminiert. Der Elternprozess wartet, bis der Subprozess fertig ist, und schreibt in die Datei »Morgen!«. Überlegen Sie, was anschließend Ihrer Meinung nach in der Datei steht? Prüfen Sie es nach!«
Voraussetzung: der Elternprozess hat die Datei erstellt und zum Schreiben geöffnet. Danach wird der Subprozess mit `fork()` erstellt, d.h. er erhält eine Kopie der Deskriptorentabelle des Elternprozesses, also mit den Verweisen auf die gleichen Einträge in der globalen Dateitabelle. Es wird also ein gemeinsames Offset verwendet [4, S. 13]. Nach Ende des Subprozesses verwendet der Elternprozess also den aktuellen Wert des Offsets, er hängt »Morgen!« an. Auf dem Bildschirm steht am Ende also wieder »Guten Morgen!« (Lösung muss noch im Programm geprüft werden).

c) »Wie oben, aber der Subprozess setzt vor seiner Terminierung mit `lseek` seine Lese-/Schreibposition auf `0`. Was steht anschließend in der Datei?« Es wird ein gemeinsames Offset verwendet, der Elternprozess beginnt also ab Position 0 zu schreiben. In der Datei steht damit am Ende »Morgen!«.

d) »Wie oben, aber beide Prozesse öffnen erst nach dem `fork()` die Datei. Welche Auswirkungen hat das auf den Dateiinhalt?« Es wird kein gemeinsames Offset mehr verwendet, d.h. das Setzen der Lese-/Schreibposition im Subprozess hat keinerlei Auswirkung auf den Elternprozess. Beide schreiben also ab Position 0, zuerst der Subprozess (»Guten «), danach der Elternprozess, der auf das Ende des Subprozesses gewartet hat (»Morgen!«). Der Inhalt der Datei ist am Ende wieder »Morgen!«.

3.9 Übungsblatt 3, Aufgabe 2

Es gibt zwei Ebenen von gegenseitigem Ausschluss bei Zugriff auf dieselben Dateien:

- Der Kernel garantiert, dass ein einzelner Schreib- oder Lesezugriff nicht durch einen anderen Zurgiff unterbrochen wird. Das geschieht unabhängig von Dateisperren automatisch bei jedem Schreib- oder Lesezugriff und lässt auch zu, dass eine Datei mehrmals gleichzeitig zum Schreiben geöffent ist. So werden Wettbewerbsbedingen verhindert, die einzelne Datensätze einer Datei zerstören oder defekte einzelne Datensätze beim Lesen liefern würden. Diese Art der Synchronisation beim Dateizugriff ist in der Aufgabenstellung dieser Aufgabe nicht gemeint!
- Der Benutzer muss durch Dateisperren (in `fcntl`) oder exklusives Öffnen `O_EXCL` garantieren, dass mehrere hintereinanderfolgende Schreibzugriffe nicht unterbrochen werden und so eine unbeabsichtigte Reihenfolge der Datensätze entstehen würde. Diese Art der Synchronisation beim Dateizugriff ist in der Aufgabenstellung dieser Aufgabe gemeint!

»Geben Sie für die möglichen Zugriffskombinationen an, ob blockiert wird:«

- »Schreiber blockiert Schreiber: ja«
  Der nun blockierte Schreiber hatte versucht, mit `F_SETLKW` ein `F_WRLK` zu setzen, während ein `F_WRLK` gesetzt war.
- »Schreiber blockiert Leser: ja«
  Der nun blockierte Leser hatte versucht, mit `F_SETLKW` ein `F_RDLK` zu setzen, während ein `F_WRLK` gesetzt war.
- »Leser blockiert Schreiber: ja«
  Der nun blockierte Schreiber hatte versucht, mit `F_SETLKW` ein `F_WRLK` zu setzen, während ein `F_RDLK` gesetzt war.
- »Leser blockiert Leser: nein«
  Es ist dem Leser möglich, mit `F_SETLKW` ein `F_RDLK` zu setzen, während ein oder mehrere andere `F_RDLK` gesetzt sind.

3.10 Übungsblatt 4, Aufgabe 1

Die Lösung ist in den Dateien im beiliegenden Verzeichnis `Aufg.4.1` enthalten.

3.11 Übungsblatt 5, Aufgabe 1

a) Mit dem cat-Kommando soll in eine Datei der Text »hallo« geschrieben werden.

```
echo hallo | cat - >hallo.txt
```

b) Falls im aktuellen Verzeichnis kein Unterverzeichnis namens »tmp« vorhanden ist, soll eines erzeugt werden. Die Zugriffsrechte: Lesen/Schreiben/Ausführen für den Eigentümer.

```
test ! -d tmp && mkdir tmp && chmod 700 tmp
```

c) In allen normalen Dateien ihres Heimatverzeichnisses, deren Namen mit »c«, »C«, »cpp« oder »CPP« enden, soll nach dem Wort »class« gesucht werden.

```
find ~ -type f -a \( -iname *c -or -iname *cpp \) | xargs grep -F "class"
```

3.12 Übungsblatt 5, Aufgabe 2

Die Lösung ist in der beiligenden Datei `Aufg.5.2.bash` enthalten.

4 Klausur 2000-03-09

Da es sich um eine studentische, ungeprüfte Lösung handelt, können natürlich Fehler enthalten sein.

4.1 Aufgabe 1 (Prozesse - 4 Punkte)

Antwort siehe Kapitel 2.1.

4.2 Aufgabe 2 (Prozesse - 4 Punkte)

Verdrängung eines Prozesses durch einen anderen: das Betriebssystem speichert den Zustand des zu verdrängenden Prozesses im Prozessdeskriptor ab (Inhalt der Register, u.a. des Befehlszählers), so dass der momentane Zustand des Prozesses später wiederhergestellt werden kann, und stellt den (abgespeicherten) Zustand des vorher blockierten Prozesses wieder her: Inhalt der Prozessorregister setzen, Zustand auf »aktiv«, Befehlszähler auf gespeicherten Wert setzen. Das Betriebssystem »merkt« den Zustandswechsel, weil es ihn selbst durchführt: der Zustand eines Prozesses wird im Prozessdeskriptor gespeichert und geändert, der vom Betriebssystem verwaltet wird. Auch ein aktiver Prozess hat nicht ununterbrochen den Prozessor, denn der regelmäßige CLOCK-Interrupt übergibt die Kontrolle an das Betriebssystem, das dann den bisher aktiven Prozess verdrängen oder fortführen kann.

4.3 Aufgabe 3 (Prozesskontrolle, Pipes, E-/A-Umlenkung - 10 Punkte)

Weil an der FH Gießen-Friederg nun C++ statt C gelehrt wird, würde die Aufgabenstellung in einer aktuellen Klausur sein: »Schreiben Sie ein C++-Programm, das die Anzahl der Zeilen in der Ausgabe des Kommandos `"ls -l"` ausgibt. Dazu soll das `ls`-Programm als Subprozess ausgeführt werden, der seine Ausgabe durch eine Pipe an den Vaterprozess leitet.«

Die Lösung ist enthalten in der beiligenden Datei `Aufg.Klausur.2000-03-09.03.cc` enthalten.

4.4 Aufgabe 4 (Synchronisation - 8 Punkte)

```
class semaphor {
  public:
    semaphor(void);
    init(int anfangswert); // Initialisierung
    up(void);
    down(void);
};
class pipe {
  private:
    int puffergroesse;
    char *puffer;
    int frei, naechsteszeichen;
    semaphor kritisch, belegt, //Korrektur der Aufgabenstellung
      vorhanden;
  public:
    pipe(int groesse);
    void put(char c);
    char get(void);
};
pipe::pipe(int groesse) {
  puffergroesse=groesse;
  puffer=new char[groesse]; //Korrektur der Aufgabenstellung
  frei=0;
  naechsteszeichen=0;
  kritisch.init(1);
  belegt.init(groesse);
  vorhanden.init(0);
}
void pipe::put(char c) {
  vorhanden.down(); //freie Plaetze sind Ressourcen; eine fuer mich!
  kritisch.down();  //kritischer Bereich ist eine Ressource
  puffer[frei]=c;
  frei = (frei+1) % groesse;
  kritisch.up();
  belegt.up(); //Zeichen sind Ressourcen
}
char pipe::get() {
  belegt.down();
  kritisch.down();
  int c=puffer[naechsteszeichen];
  naechsteszeichen = (naechsteszeichen+1) % groesse;
  return c;
  kritisch.up();
  vorhanden.up();
}
```

4.5 Aufgabe 5 (Shellskript - 6 Punkte)

Die Lösung ist enthalten in der beiligenden Datei `Aufg.Klausur.2000-03-09.05.bash` enthalten. Testdateien sind in den beiliegenden Dateien `Aufg.Klausur.2000-03-09.05.test1.txt.gz` und `Aufg.Klausur.2000-03-09.05.te` enthalten.

4.6 Aufgabe 6 (Verklemmungen - 4 Punkte)

Siehe [2, S. 68-70]. Zum besseren Verständnis kann man Kanten, die zu später benötigten Ressourcen zeigen, mit »anforderbar« bezeichnen, die anderen beiden Kantenarten mit »angefordert« bzw. »erhalten«. Da das Beispiel in der Aufgabenstellung fehlt, kann sie hier nicht gelöst werden.

4.7 Aufgabe 7 (Hauptspeicher - 2+2 Punkte)

a) »Welche Möglichkeiten der Verwaltung freier Speicherblöcke (Platte oder Hauptspeicher) kennen Sie?« Siehe [2, S. 79-81]:

- Partitionsverwaltung mit verketteten Listen. Das Betriebssystem verwaltet die freien Hauptspeicherbereiche in einer Freiliste (doppelt verkettete Liste, jedes Element enthält Anfangsadresse und Größe der Partition). Strategien zur Auswahl eines freien Bereichs bei Aufruf eines Programms, sowohl bei fester als auch bei variabler Partitionierung:
  - first fit
  - best fit
  - worst fit
  - quick fit
- Buddy-System. Vergibt Speicherplatz in Blockgrößen, die Zweierpotenzen sind. Dadurch schnelle Verschmelzung benachbarter freier Blöcke, aber starke interne Fragmentierung.
- Bitmap-basierte Verwaltung. Vielfache von Blöcken fester Größe als Zuteilungseinheit, verwaltet in Bitmaps.
- Speicherkompaktierung. Zusammenschieben der belegten Bereiche, so dass stets ein einziger großer freier Bereich am Speicherende zur Verfügung steht.
- Paging. Seiten als einzige Zuteilungseinheit, freie Seiten in der Seitenrahmentabelle verwaltet.
- Liste freier Blöcke im Dateisystem, kombiniert aus direkter Liste, einfach indirekter Liste und doppelt indirekter Liste.

b) »Erläutern Sie, wie das Betriebssystem in einem System mit Paging und Segmentierung die Seiten-Seitenrahmenzuordnung eines *segmentierten* Prozesses verwaltet.« Siehe [2, S. 93].

4.8 Aufgabe 8 (Hauptspeicher - 4 Punkte)

FIFO Tritt ein Seitenfehler auf (angeforderte Seite in keinem der drei Rahmen enthalten), so wird derjenige Rahmeninhalt ersetzt, der schon am längsten konstant ist. Man prüft also, beginnend bei Rahmen 1, welcher Rahmen in den meisten unmittelbar vorhergehenden Spalten unveränderten Inhalt hatte.

angeforderte Seite	5	7	1	2	7	6	7	4	2	7	1	6	4	5
danach Rahmen 1:	5	5	5	2	2	2	2	4	4	4	4	6	6	6
danach Rahmen 2:		7	7	7	7	6	6	6	2	2	2	2	4	4
danach Rahmen 3:			1	1	1	1	7	7	7	7	1	1	1	5

Seitenfehler: 12

OPT Tritt ein Seitenfehler auf, so wird derjenige Rahmeninhalt ersetzt, der in Zukunft für die längste Zeit nicht mehr benötigt wird (also am besten gar nicht mehr, dann tritt bei Auslegerung auch kein Seitenfehler auf).

angeforderte Seite	5	7	1	2	7	6	7	4	2	7	1	6	4	5
danach Rahmen 1:	5	5	5	2	2	2	2	2	2	2	1	6	6	5
danach Rahmen 2:		7	7	7	7	7	7	7	7	7	7	7	7	7
danach Rahmen 3:			1	1	1	6	6	4	4	4	4	4	4	4

Seitenfehler: 9

4.9 Aufgabe 9 (CPU-Scheduling - 3 Punkte)

Aufgabenstellung: »Zur Berücksichtigung von Prioritäten verwalten viele Scheduler separate Prozesswarteschlangen für jede Prioritätsstufe. Wie könnte man mit nur einer Warteschlange für alle nicht blockierten Prozesse und einer einfachen »round robin«-Strategie dafür sorgen, dass Prozesse mit hoher Priorität begünstigt werden?«

Lösung: Indem man einen Prozess bei Verdrängung nicht ans Ende der Warteschlange anreiht, sondern an der Position (vom Kopf der Warteschlange an, beginnend mit 1) einfügt, die seine Priorität angibt. Jede Priorität darf dabei nur einmal vorkommen. Das bedeutet:

- Der Prozess mit Priorität 1 wird, sofern er existiert, immer wieder an den Kopf der Warteschlange gestellt, bis sie blockieren. So entsteht eine Art Echtzeitbetrieb.
- Der Prozess mit Priorität 2 kommt jedes 2. mal dran.
- Der Prozess mit Priorität 3 kommt jedes 3. mal dran.
- $\vdots$
- Der Prozess mit Priorität n kommt jedes n. mal dran.

Eine einfachere Lösung besteht darin, einem Prozess entsprechend seiner Priorität mehr oder weniger Prozessorzeit zuzuteilen und ihn nach Ablauf dieser Zeit wieder ans Ende der Warteschlange anzureihen.

4.10 Aufgabe 10 (Dateisystem - 4 Punkte)

Aufgabenstellung: »Wozu enthält bei einem UNIX-Dateisystem jeder Inode einen Referenzzähler?«

Lösung: Weil Inodes nur die Daten der Datei enthalten, nicht jedoch ihren Dateinamen definieren. Ein Indode kann mehrere Dateinamen haben (sog. Hardlinks), in solchen Fällen hat der Referenzzähler einen Wert größer 1. Nur wenn keine Referenz mehr auf einen Indode verweist (die »Datei keinen Namen mehr hat«), kann sie gelöscht weden. Dies kann daran erkannt werden, dass der Referenzzähler 0 ist.

Aufgabenstellung: »Wenn ein Prozess in einem Mehrbenutzersystem, z.B. UNIX oder Windows NT, eine Datei öffnet, führt das Betriebssystem eine Zugriffsrechtsprüfung durch. Beschreiben Sie kurz, wie diese Prüfung typischerweise stattfindet, insbesondere auch, welche Datenstrukturen das Betriebssystem zur Prüfung der Rechte braucht.«

Lösung: Bei Windows NT: die Kontrollinformationen einer Datei verwaltet das Betriebssystem in einer Datenstruktur »Dateikontrollblock«, in der u.a. auch der Dateideskriptor repräsentiert ist. Zu jedem Objekt, also auch zu jeder Datei, gehört eine ACL (Zugriffskontrollliste), die ebenfalls im Dateikntrollblock gespeichert ist. Verfahren der Zugriffsrechtsprüfung: ein Prozess hat ein Zugriffsrecht, wenn dieses der Domäne, der der Prozess angehört, in der ACL zugestanden wird, etwa durch einen Eintrag der Form: erlaube <Prozessdomäne> <Zugriffsrechte>.

4.11 Aufgabe 11 (Netzwerkprotokolle - 4 Punkte)

(Dieser Stoff wurde im Wintersemester 2002 / 2003 nicht behandelt.)

4.12 Aufgabe 12 (Serverarchitekturen - 3 Punkte)

Aufgabenstellung: »Was ist ein nebenläufiger Server (concurrent server)?«

Lösung: Ein Programm, das auf Aufträge von Client-Programmen über eine Schnittstelle wartet. Es zeichnet sich gegenüber iterativen Servern dadurch aus, dass jeder Auftrag durch einen separaten Thread (oder Subprozess) abgearbeitet wird, so dass mehrere Aufträge (quasi-)gleichzeitig bearbeitet werden können und der Server nahzu immer bereit ist, Aufträge anzunehmen.

4.13 Aufgabe 13 (Socket-Schnittstelle - 2+1+3 Punkte)

a) »Erläutern Sie die Begriffe *verbindungsorientierte* und *paketorientierte* Kommunikation.« Paketorientierte Kommunikation wird auch verbindungslose Kommunikation genannt, ebenso hier. Während bei verbindungsloser Kommunikation jede Nachricht mit der vollständigen Zieladresse versehen sein muss, ist bei verbindungsorientierter Kommunikation bei geöffneter Verbindung der Verbindungsendpunkt (z.B. Socket) von Client bzw. Server eineindeutig mit der Adresse des Kommunikationspartners gekoppelt und muss nicht mehr wiederholt werden. Bei geöffneter Verbindung kommen am Verbindungsendpunkt nur Nachrichten des Kommunikationspartners an und keine anderen.

b) »Welches der beiden Kommunikationsschemata wird durch die Socket-Schnittstelle unterstützt?« Beide. Verbindungsorientierte Kommunikation wird durch den Sockettyp SOCK_STREAM realisiert, verbindungslose Kommunikation durch den Sockettyp SOCK_DGRAM (Datagramm-Kommunikation).

c) »Nennen Sie einige Socket-Systemaufrufe mit kurzer Erläuterung ihrer Funktionalität.« Siehe [5, S. 5-9].

4.14 Aufgabe 14 (Internetadressen - 2+2 Punkte)

a) »Wie ist eine Internet-Adresse (`struct inet_addr`) aufgebaut?«

```
struct inet_addr {
  unsigned long int s_addr;
};
```

Diese einzige Komponente ist die IP-Adresse (eindeutige Identifikationsnummer) einer Netzwerkschnittstelle eines Hosts im Internet bzw. irgendeinem anderen TCP/IP basierten Netzwerk. Sie kann aufgeteilt werden in eine führende Netzwerkadresse und eine Rechneradresse in diesem Netzwerk. Im Internet gibt es Netzwerke in 3 Größen (Klassen A, B, C) und dafür reservierte Adressbereiche.

b) »Wozu dient eine TCP-Port-Nummer?« Es ist eine eindeutige Zieladresse für Nachrichten innerhalb eines Rechners (der durch eine oder mehrere IP-Adressen angesprochen werden kann). Es ist nicht praktikabel, jedem Prozess eine weltweit eindeutige Identifikation zu geben, also hat man Ports einerichtet: an einer solchen »Anlegestelle für Nachrichten« wartet ein Prozess auf für ihn bestimmte Nachrichten, nachdem er sich an diesen Port gekoppelt hat. Somit bieten TCP-Ports eine Möglichkeit zur Interprozesskommunikation über Systemgrenzen hinweg.

A Errata

Errata, Anmerkungen und Ergänzungen zu [2].

Seitenzahlen Die Seitenzahlen der Kapitel im Inhaltsverzeichnis sind konsequent um 3 zu niedrig, weil das Inhaltsverzeichnis die Seiten 2, 3, 4 belegt und somit alle anderen Seiten um 3 nach hinten verschiebt. Fehlender LaTeX-Lauf zur Aktualisierung der Referenzen?

S.15: »Speicherverwaltungseinheit (MMU)«, auch S. 28 »MMU«. Es wäre hilfreich zum Verständnis, bereits an dieser Stelle kurz zu erklären, was die MMU ist, inkl. der Bedeutung der Abkürzung. Oder ein Verweis auf S. 75 unten, wo die MMU behandelt wird.

S.22, Grafik: Die Beschriftung »CPU-Zuteilung« muss am Zustandsübergang zwischen »bereit« und »ausführend« stehen.

S.22: »Ereignissklassen«

S.24: »Ereignissklassen«

S.36: »`pthread_create(&f_thread, NULL, (void* (*)(void *)) f, NULL);`« Vielleicht ein Kommentar mit Hinweis, dass hier ein C-Style Cast in einen Funktionszeigertyp verwendet wird? Doch ein recht seltenes Konstrukt...

S.39: Ersetze »zwei nebenläaufige neu-Aufrufe sind aktiv.« durch »zwei nebenläaufige lies-Aufrufe sind aktiv.«.

S.40: Ersetze »C-Anweisungen anzahl– in lies und anzahl++ in neu sind atomar,« durch »C-Anweisungen anzahl– in lies und anzahl++ in neu sind nicht atomar,«.

S.41: »sich gerade im kritische Abschnitt befindet«

S.65: nach »anfordern („alles« ist der untere Seitenrand erreicht, einige Wörter fehlen. Vielleicht ein Poblem mit der im PDF-Dokument verwendeten Papiergröße Letter (215,9mm·279,4mm) statt A4 (210mm·297mm)? (verwendete Betrachter: Acrobat Reader 5.0.5 für Linux; kghostview 0.11 für Linux).

S.68: Ersetze »Philosoph 4 greift zuerst nach dem rechten Stäbchen (Nummer 0), dann nach dem linken (Nummer 1).« durch »Philosoph 4 greift zuerst nach dem rechten Stäbchen (Nummer 0), dann nach dem linken (Nummer 4).«

S.70: Dasselbe Problem mit fehlendem Text wie auf S.65.

S.75: »Ladezeit-Bindung« in Kap. »logischer und physikalischer Adressraum«: klarer mit »Adressbindung zur Ladezeit«?

S.83: »Hauptspeichersverwaltung«

S.85: »typischerweise 2 KB oder und 4 KB.«

S.96: »Zum besseren Verständnis vergleichen wir das Verhalten für einen 3 Rahmen großen Hauptspeicher.« Dieser Abschnitt ist ohne einen erläuternden Text recht schwer verständlich. Vorschlag: die Zugriffe in der Zugriffsreihenfolge als Spaltentitel der Tabellen für die verschiedenen Auslagerungsverfahren verwenden; erklärender Text: »Der Hauptspeicher sei 3 Rahmen groß, es ist jedoch ein Zugriff auf bis zu 10 Seiten, nummeriert mit 0-9, möglich. Eine Spalte gibt an, welche Seiten zu einem Zeitpunkt im Hauptspeicher stehen, darunter muss natürlich die durch den Zugriff angeforderte Seite sein. Die Hauptspeicherbelegung nach dem nächsten Zugriff steht immer eine Spalte weiter rechts.«

S.97: »verknüpfen diese Modell«

S.104: »Platen-E/A«

S.118, Tabelle: »B+-Baum« Was ist das?

S.119: »Sicherheit ist ein allerdings ein facettenreicher Begriff«

S.121: »Bei viele modernen UNIX-Varianten«

S.121: »wird auf die weiterführende Literatur verweisen:«

S.125: »Fließkommanfehler«

S.125: Setze Punkt in »hat eine bestimmte Priorität p Eine Service-Routine«

Literatur

[1] »Tanenbaum: »Moderne Betriebssysteme«; 2. Auflage 2002. Dies ist die Empfehlung für die Vorlesung Betriebssysteme bei Prof. Jäger, bes. wenn man etwas über die Vorlesung heraus nachlesen will. Preis ca. 50-55 EUR. Dieses Werk gehört zur Einführungsliteratur.

[2] Prof. Dr. Michael Jäger: »Betriebssysteme 1 - Eine Einführung«; WS 2000 / 2001; Version 1.1.0 vom 2000-10-04. Quelle: http://homepages.fh-giessen.de/~hg52/lv/bs1/root.html.

[3] Prof. Dr. Michael Jäger: »Unix-Prozesskonzept«; Version vom 2000-10-10. Quelle: http://homepages.fh-giessen.de/~hg52/lv/bs1/root.html.

[4] Prof. Dr. Michael Jäger: »UNIX-Dateisystem – Eine Einführung«; FH Gießen-Friedberg Version vom 2000-10-10.

[5] Prof. Dr. MichaelJäger: »Sockets – eine Programmierschnittstelle für Kommunikation im Netz«; Fachbereich MNI; FH Gießen-Friedberg; Version vom 2000-10-10.

[6] Manpage signal(7) »signal - list of available signals«. Auf jedem Unix-artigen System einzusehen mit `man 7 signal`.

[7] Manpage exec(3) »execl, execlp, execle, execv, execvp - execute a file«. Auf jedem Unix-artigen System einzusehen mit `man 3 exec`.