TECHNISCHE UNIVERSITÄT MÜNCHEN
Fakultät für Informatik
Lehrstuhl für Betriebssysteme F13
Prof. Dr. Uwe Baumgarten

Bachelorprüfung Modul IN0034
Betriebssysteme und hardwarenahe Programmierung für Games
28.02.2013
Prüfungsdauer: 90 Minuten

Hiermit bestätige ich, dass ich vor Prüfungsbeginn darüber in Kenntnis gesetzt wurde, dass ich im Falle einer plötzlich während der Prüfung auftretenden Erkrankung das Aufsichtspersonal umgehend informieren muss. Dies wird im Prüfungsprotokoll vermerkt. Danach muss unverzüglich ein Rücktritt von der Prüfung beim zuständigen Prüfungsausschuss beantragt werden. Ein vertrauensärztliches Attest - ausgestellt am Prüfungstag - ist unverzüglich nachzureichen. Wird die Prüfung hingegen in Kenntnis der gesundheitlichen Beeinträchtigung dennoch regulär beendet, kann im Nachhinein kein Prüfungsrücktritt aufgrund von Krankheit beantragt werden. Wird die Prüfung wegen Krankheit abgebrochen, wird die Klausur mit der Note "5,0 - nicht erschienen" gemeldet und - unabhängig von einem Rücktrittsantrag - nicht bewertet.

- Nur ausgeteilte Blätter werden bewertet
- Hilfsmittel: Doppelseitig beschriebenes Din-A4 Blatt
- Bitte kein Bleistift, Tintenkiller oder Tipp-Ex verwenden, sondern durchstreichen!
- Die Prüfung besteht aus 18 Blättern - Bitte vor Beginn überprüfen!
- Bitte achten Sie darauf, sowohl die Vorder- als auch die Rückseite zu bearbeiten
- Bei den Punktangaben handelt es sich um vorläufige Richtwerte

Vorname: _____ Name: _____
Matrikel-Nr.: _____ Unterschrift: _____
Hörsaal: _____ Reihe: _____ Platz: _____

Aufgabe:	1	2	3	4	5	6	7	8	
Punkte:	4	6	10	8	4	9	10	24	$\sum$ 75
Erreicht:									

1 Sichtenmodell (_/4P)

Nennen Sie die Funktionseinheiten der Von-Neumann Architektur und beschreiben Sie kurz deren jeweilige Aufgabe.

- Leitwerk/Steuerwerk (control unit)

  Holt und interpretiert die Maschinenbefehle

  Erzeugt Steuerkommandos für alle anderen Komponenten

- Rechenwerk (ALU: Arithmetical Logical Unit)

  Führt alle Befehle aus, bis auf Ein/Ausgabe- und

- Speicher (storage memory)

  Speichert Programme und Daten

- Ein-/Ausgabewerk (I/O unit)

  Steuert Ein/Ausgabe-Geräte, inkl. Periphere Speicher

2 Hardware-Software-Schnittstelle (_/6P)

Beschreiben Sie jeweils einen Vor- und Nachteil von RISC (Reduced Instruction Set Computer) und CISC (Complex Instruction Set Computer) und einen sich daraus ergebenden Einsatzbereich.

- CISC - Komfortabel und relativ mächtig

  Vorteil: einfache Programmierbarkeit, geringer Speicherbedarf

  Nachteile: komplexe (langsame) Implementierung insbesondere Dekodierung, evtl. viele Funktionen ungenutzt.

  Einsatzbereich: x86, Desktoprechner

- RISC - Minimalistisch auf das absolut Notwendige beschränkt

  Vorteil: einfache, effiziente, schnelle Implementierung

  Vorteil: Schnelle Interrupt-Behandlung

  Nachteil: schwierigere Programmierbarkeit

  Einsatzbereich: ARM/Embedded aufgrund geringer Speicherbedarf

3 Prozessoren & weitere Hardware (_/10P)

3.1 Flynnsche Klassifikation (4 Punkte)

Nennen Sie die Klassifikationen von Flynn und geben Sie jeweils ein Beispiel für einen Prozessor. Abkürzungen bitte ausschreiben.

- SISD: Single Instruction Stream Single Data Stream
  Einprozessor (Uniprocessor) ggf. mit ILP
- SIMD: Single Instruction Stream Multiple Data Stream
  Vektorrechner, Multimedia Erweiterungen, GPUs
- MISD: Multiple Instruction Stream Single Data Stream
  nie gebaut
- MIMD: Multiple Instruction Stream Multiple Data Stream
  Multiprozessoren für TLP

3.2 Leistung (_/1P)

Wie kann eine Leistungssteigerung bei der Verarbeitung von Prozessorbefehlen erreicht werden ohne den Takt zu erhöhen?
Durch Parallelität.

3.3 Caches (_/2P)

Welches Prinzip steckt hinter dem Einsatz von Caches? Beschreiben Sie die Grundidee hinter dem Prinzip.
Lokalitätsprinzip. Die Idee ist dass beim Ablaufen eines Programms der Zugriff auf nahe beieinander liegenden Daten sehr wahrscheinlich ist.

3.4 Pipelining (_/3P)

Angenommen ein Prozessor besitzt 15 Phasen (n) in der Pipeline und es sollen 9.986 Befehle (b) abgearbeitet werden. Bestimmen Sie den Speedup auf zwei Nachkommastellen genau:

$$Speedup = \frac{n \cdot b}{n + (b - 1)}$$

$$Speedup = \frac{15 \cdot 9986}{15 + (9986 - 1)} = 14.979$$

4 Prozesse (_/8P)

4.1 Threads (_/4P)

1. Nennen Sie einen Vorteil bei der Verwendung von Threads gegenüber von Prozessen bezüglich des Austauschs von Daten?

2. Nennen Sie eine Gefahr, die sich bei der Verwendung von Threads ergeben kann, und nennen Sie einen Mechanismus, um diese zu behandeln?

Leichtgewichtig. Einfacher Austausch von Daten. Gefahren sind Wettlaufsituationen und Deadlocks. Lösung sind Atomare Operationen, Sperrvariablen oder kritische Abschnitte (atomic, spinlock, mutex).

4.2 Kommunikation (_/4P)

Beschreiben Sie den Unterschied zwischen Benannten Pipes und einer regulären Datei und erläutern Sie den Vorteil für die Prozesskommunikation, welcher unter Verwendung regulärer Dateien nur schwer realisierbar wäre.
Ein wesentlicher Unterschied besteht darin, dass eine FIFO-Datei kein „Ende“ hat. Liest man ein Zeichen aus einer leeren, regulären Datei, erhält man den Dateiende-Marker *EOF*. Bei einer FIFO-Datei wird der Prozess so lange angehalten, bis ein Zeichen in die FIFO-Datei geschrieben wurde. Auch beim Schreiben in die FIFO-Datei wird der Prozess blockiert, bis ein anderer Prozess die FIFO-Datei zum Lesen öeffnet. Dadurch lassen sich zwei Prozesse mit Hilfe von FIFO-Dateien synchronisieren, was mit regulären Dateien nur schwer möglich ist.

5 Speicherverwaltung (_/4P)

Für die Referenzkette 0172723013 soll die Anzahl der Seitenfehler ermittelt werden. Es gibt acht Seiten und vier Seitenrahmen. Die Seitenrahmen sind dabei zu Beginn leer.

1. Wie viele Seitenfehler produziert der FIFO-Algorithmus?

   First-In-First-Out

RK		0	1	7	2	7	2	3	0	1	3
R0											
R1											
R2											
R3											

   Anzahl der Seitenfehler: _

2. Wie viele Seitenfehler produziert der LRU-Algorithmus?

   Least-Recently-Used

RK		0	1	7	2	7	2	3	0	1	3
R0											
R1											
R2											
R3											

   Anzahl der Seitenfehler: _

First-In-First-Out

RK		0	1	7	2	7	2	3	0	1	3
R0		0	0	0	0	0	0	3	3	3	3
R1			1	1	1	1	1	1	0	0	0
R2				7	7	7	7	7	7	1	1
R3					2	2	2	2	2	2	2

Anzahl der Seitenfehler: 7

Least-Recently-Used

RK		0	1	7	2	7	2	3	0	1	3
R0		0	1	7	2	7	2	3	0	1	3
R1			0	1	7	2	7	2	3	0	1
R2				0	1	1	1	7	2	3	0
R3					0	0	0	1	7	2	2

Anzahl der Seitenfehler: 7

6 Zeiger (_/9P)

In dieser Aufgabe geht es um die Adressierung von Elementen innerhalb eines Arrays. Schreiben Sie ein C-Programm. Verwenden Sie hierzu die Funktionsliste auf Seite 21. Es wird keine Fehlerbehandlung erwartet.

6.1 Funktionszeiger (_/3P)

Schreiben Sie eine Funktion:
`void map(int * array, int length, int(*f)(int))`. Diese Funktion erhält als Parameter ein Array, sowie dessen Länge und eine Funktion `f`. Die Funktion `map` wendet `f` auf alle Elemente des Arrays an.

6.2 Funktion to_upper (_/1P)

Schreiben Sie eine Funktion: `int to_upper(int c)`. Diese Funktion erhält als Parameter eine Ganzzahl, welche die dezimale Entsprechung eines ASCII Zeichens darstellt. Beispielsweise ist $A = 65$ und $a = 97$. Die Funktion soll alle $a, ..., z$ nach $A, ..., Z$ konvertieren.

6.3 Main (_/5P)

Erstellen Sie nun die main-Funktion. Legen Sie zunächst ein statisches Array `array` für 10 Ganzzahlen (Typ `int`) an. Schreiben Sie anschließend eine Schleife, welche das Array mit 10 zufälligen Kleinbuchstaben befüllt. Hinweis: Kleinbuchstaben liegen nach ASCII im Wertebereich zwischen 97 und 122. Wandeln Sie nun mittels der Funktionen `map` und `to_upper` die Kleinbuchstaben im Array in Großbuchstaben um.

```
1  #include <stdio.h>
2  #include <time.h>
3  #include <stdlib.h>
4
5  int to_upper(int c)
6  {
7          return c-32;
8  }
9
10 void map(int * array, int length, int(*f)(int))
11 {
12         unsigned int _cnt;
13         for(_cnt = 0; _cnt < length; _cnt++)
```

```
14                  array[_cnt] = f(array[_cnt]);
15  }
16
17  int main(int argc, char *argv[])
18  {
19          int _array[10] = {0};
20          int _cnt = 0;
21
22          srand(time(NULL));
23
24          while(_cnt < 10)
25                  _array[_cnt++] =  rand() % 26 + 97;
26
27          map(_array, 10, to_upper);
28
29          return 0;
30  }
```

7 Nebenläufigkeit (_/10P)

Ein Programm soll die Bearbeitung von Gehaltsdaten durch mehrere Personalmitarbeiter simulieren. Das Ziel des Programms ist es, eine möglichst realitätsnahe Simulation umzusetzen. In der Realität können die Gehaltsdaten von verschiedenen Mitarbeitern gleichzeitig bearbeitet werden. Aus diesem Grund hat man sich dazu entschieden, die Personalmitarbeiter als Threads zu modellieren. Betrachten Sie nun folgendes Codefragment.

```
1 #include <pthread.h>
2
3 void erhoehe_gehalt(float *altes_gehalt,
4                     float erhoehung)
5 {
6   (*altes_gehalt) += erhoehung;
7 }
```

Gehen Sie davon aus, dass die Funktion `erhoehe_gehalt` eine Thread-Funktion darstellt, die von mehreren Threads gleichzeitig ausgeführt wird. Was passiert in der Situtation, wenn Personalmitarbeiter A das aktuelle Gehalt von 1000,00 Euro um 100,00 Euro erhöhen will und Personalmitarbeiter B, aus einem anderen Grund, das gleiche Gehalt zusätzlich um 200,00 Euro? Erhält der Mitarbeiter in so einem Fall zuverlässig 1.300,00 Euro? Falls ein Problem bei diesem Vorgehen entsteht, so beschreiben Sie es und nennen Sie zwei Möglichkeiten zur Lösung und setzen Sie eine durch Erweiterung des obigen Codefragments um.
Nein der Mitarbeiter erhält keine 1300 Euro, korrekterweise erhält er 1200 Euro. Die Erklärung liegt darin, dass zwischen A und B eine klassische Wettlaufsituation eintritt (Racecondition). Erhöht A das Gehalt um 100 Euro auf 1100 Euro und kommt B vor dem Schreibprozess von A zur Ausführung hat B nicht etwa diesen neuen Wert, sondern arbeitet mit dem Ausgangswert von 1000 Euro und erhöht diesen um 200 Euro und überschreibt dabei den bereits errechneten Wert von A. Mögliche Lösungen.

- mutex
- semaphore

```
1 #include <pthread.h>
2
3 pthread_mutex_t a_lock = PTHREAD_MUTEX_INITIALIZER;
4
5 void erhoehe_gehalt(float *altes_gehalt, float erhoehung)
```

```
6  {
7  pthread_mutex_lock(&a_lock ) ;
8  (*altes_gehalt) += erhoehung;
9  pthread_mutex_unlock(&a_lock ) ;
10 }
```

8 Server/Client (_/24P)

In dieser Aufgabe sollen TCP/IP-basierte Clients und Server über die IP-Adresse 127.0.0.1 und dem Port 30000 Nachrichten austauschen können. Der Server soll die Verzeichnisse der zweiten Hierarchiestufe (z.b. /etc, /bin, /usr) seines Linux-Dateisystems unformatiert an die Clients ausgeben. Zusätzlich soll der Server gleichzeitige Anfragen von mehreren Clients bedienen können. Ergänzen Sie den folgenden Server- und Client-Code dementsprechend.

8.1 Server (_/19P)

```
1  char * get_directories(void)
2  {
3   char * listing = NULL;
4   char * sub_dir;
5   DIR * directory;
6   struct dirent *dir_entry;
7   directory = ..........("/");
8   while((dir_entry = ............(directory)) != NULL) {
9           listing = (char*).........(...........,
10                        (listing ? strlen(listing) : 1) +
11                         strlen(..............));
12          ............(listing, dir_entry->d_name,
13                        strlen(dir_entry->d_name));
14    }
15  }
16  closedir(directory);
17  return listing;
18 }
19
20 int main(void) {
21  int pid;
22  struct sockaddr_in serverAddress;
23  serverAddress.sin_family = PF_INET;
24  serverAddress.sin_port = htons(.......);
25  inet_pton(PF_INET, ....., &(serverAddress.sin_addr));
26  int socketID = ...........(PF_INET, ........., 0);
27  ...........(socketID,
28              (struct sockaddr *) &serverAddress,
29              sizeof(..........));
```

```
  ..........(socketID, 10);
  ......... {
   struct sockaddr_storage clientAddress;
   unsigned int addressSize = sizeof(.............);
   int connectionSocketID = ........(socketID,
                            (struct sockaddr *)&clientAddress,
                            &addressSize);
   ...... = .......;
   if(pid == 0) {
    close(socketID);
    char * listing = get_directories();
    .........(connectionSocketID, listing, strlen(message));
    free(listing);
    exit(0);
   } else {
    close(connectionSocketID);
   }
  }
}
```

```
char * get_directories(void)
{
 char * listing = NULL;
 DIR * directory;
 struct dirent *dir_entry;
 directory = opendir("/");
 while((dir_entry = readdir(directory)) != NULL) {
         listing = (char*)realloc(listing, (listing ? strlen(listin
                         strlen(dir_entry->d_name));
         strncat(listing, dir_entry->d_name, strlen(dir_entry->d_na
 }
 closedir(directory);
 return listing;
}

int main(void) {
 int pid;
 struct sockaddr_in serverAddress;
 serverAddress.sin_family = PF_INET;
 serverAddress.sin_port = htons(30000);
 inet_pton(PF_INET, "127.0.0.1", &(serverAddress.sin_addr));
```

```
22  int socketID = socket(PF_INET, SOCK_STREAM, 0);
23  bind(socketID,
24              (struct sockaddr *) &serverAddress,
25              sizeof(serverAddress));
26  listen(socketID, 10);
27  while(true){
28   struct sockaddr_storage clientAddress;
29   unsigned int addressSize = sizeof(clientAddress);
30   int connectionSocketID = accept(socketID,
31                            (struct sockaddr *)&clientAddress,
32                            &addressSize);
33   pid = fork();
34   if(pid == 0) {
35    close(socketID);
36    char * listing = get_directories();
37    send(connectionSocketID, listing, strlen(listing), 0);
38    free(listing);
39    exit(0);
40   } else {
41    close(connectionSocketID);
42   }
43  }
44  close(socketID);
45  return 0;
46 }
```

8.2 Client (_/5P)

```
1  int main(int argc, char ** argv)
2  {
3   int sock_fd;
4   struct sockaddr_in server_addr;
5   int err = 0, length;
6   char buffer[256] = {0};
7   sock_fd = socket(PF_INET, SOCK_STREAM, 0);
8   if(sock_fd == -1)
9   {
10   perror("socket() failed");
11  }
12  server_addr.sin_family = PF_INET;
13  server_addr.sin_port = htons(..........);
```

```
14   inet_pton(PF_INET, ..........  , &(server_addr.sin_addr));
15   ............(sock_fd, (struct sockaddr*)&server_addr,
16                            sizeof(struct sockaddr_in));
17   while(1)
18   {
19    length = ..........(sock_fd, buffer, 256, 0);
20    if(length == 0)
21    {
22     puts("Connection␣closed␣by␣remote␣host");
23     break;
24    }
25    ..........(STDOUT_FILENO, buffer, length);
26   }
27   close(sock_fd);
28   return 0;
29  }
```

```
1  int main(int argc, char ** argv)
2  {
3          int sock_fd;
4          struct sockaddr_in server_addr;
5          int err = 0, length;
6          char buffer[256] = {0};
7
8          // socket
9          sock_fd = socket(PF_INET, SOCK_STREAM, 0);
10         if(sock_fd == -1)
11         {
12                 perror("socket()␣failed");
13         }
14         // settings
15         server_addr.sin_family = AF_INET;
16         server_addr.sin_port = htons(30000);
17         inet_pton(AF_INET, "127.0.0.1", &(server_addr.sin_addr));
18
19         err = connect(sock_fd, (struct sockaddr*)&server_addr,
20                           sizeof(struct sockaddr_in));
21         if(err == -1)
22                 perror("connect()␣failed");
23
24         while(1)
```

```
25          {
26                  length = recv(sock_fd, buffer, 256, 0);
27                  if(length == 0)
28                  {
29                          puts("Connection␣closed␣by␣remote␣host");
30                          break;
31                  }
32                  write(STDOUT_FILENO, buffer, length);
33          }
34
35          close(sock_fd);
36          return 0;
37  }
```

Funktionen und Strukturen

Funktionen

```
1  int rand (void);
2  void srand (unsigned int seed);
3  time_t time (time_t * timer);
4  int fclose (FILE *stream);
5  int fgetc (FILE *stream);
6  int getchar (void);
7  int fputc (int c, FILE *stream);
8  int putchar (int c);
9  int fseek (FILE *stream, long int off, int whence);
10 long int ftell (FILE *stream);
11 void rewind (FILE *stream);
12 FILE *fopen (const char *filename, const char *modes);
13 int fprintf (FILE *stream, const char *format, ...);
14 int printf (const char *format, ...);
15 int sprintf (char *s, const char *format, ...);
16 int fscanf (FILE *stream, const char *format, ...);
17 int scanf (const char *format, ...);
18 int sscanf (const char *s, const char *format, ...);
19 void *malloc (size_t size);
20 void *realloc (void *ptr, size_t size);
21 void free (void *ptr);
22 pid_t fork (void);
23 int pipe (int pipedes[2]);
24 int shmget(key_t key, int size, int shmflag);
25 void * shmat(int shmid, void * shmaddr, int shmflag);
26 int shmctl(int shmid, int cmd, struct shmid_ds *buf);
27 struct dirent *readdir(DIR *dirp);
28 DIR *opendir(const char *name);
29 char * strncat (char * destination, char * source, size_t num);
30 size_t strlen (const char * str);
31 char * strncpy (char * destination,
32                 const char * source,
33                 size_t num);
```

Strukturen

```
1 struct dirent {
2  long d_ino;
3  off_t d_off;
4  unsigned short d_reclen;
5  char d_name[1];
6 };
```