Lösung zur
Klausur zu Systemnahe Software

WS 95/96

5. Februar 1996

Aufgabe		Punkte	Bewertung
1	a)	4	
	b)	2	
	c)	2	
2	a)	2	
	b)	2	
	c)	2	
	d)	2	
3	a)	4	
	b)	6	
4	a)	2	
	b)	4	
	c)	4	
5	a)	4	
	b)	4	
6	a)	2	
	b)	6	
7	a)	3	
	b)	4	
	c)	3	
Summe		**62**	

Aufgabe 1

Mit dem Systemaufruf `open()` soll eine Datei zum Lesen und Schreiben geöffnet werden, jedoch nur, falls sie noch nicht existiert. Außerdem soll nur der Besitzer Lese- und Schreibrechte besitzen.

(*a*) Geben Sie den entsprechenden Aufruf von `open()` an.

```
open(path, O_CREAT | O_EXCL | O_RDWR, 0600)
```

(*b*) In welcher System-Struktur werden die sogenannten *File Status Flags* (`oflag`) gespeichert?

Die *File Status Flags* befinden sich in demjenigen Eintrag der *Open File Table (OFT)* des Kernels, auf den der Filedeskriptor der geöffneten Datei verweist.

(*c*) Mit welchem Systemaufruf können die *File Status Flags* explizit verändert werden?

Mit `fcntl()` und dem Kommando `F_SETFL`.

Aufgabe 2

Sie wollen Informationen über eine Datei haben,
wie beispielsweise Zugriffsrechte, Dateigröße, Zeitstempel ...

(*a*) Wo im System befinden sich solche Informationen?

In einer Inode

(*b*) Mit welchem Systemaufruf bekommen Sie diese Informationen, falls Sie den Pfadnamen gegeben haben?

Mit `stat()`

(*c*) Mit welchem Systemaufruf bekommen Sie diese Informationen, falls Sie nur einen Filedeskriptor gegeben haben?

Mit `fstat()`

(*d*) Mit welchem Systemaufruf können die Zeitstempel einer Datei explizit gesetzt werden?

Mit `utime()`

Aufgabe 3

Sie sollen für eine C-Bibliothek eine Sortierfunktion `sort()` bereitstellen, welche die Elemente eines Vektors sortiert. Die Funktion soll jedoch unabhängig vom Datentyp der Elemente sein, d.h. die Elemente sind lediglich Speicherflächen gleicher Größe. Die Sortierordnung soll beim Aufruf durch Übergabe einer Ordnungsfunktion bestimmt werden.

(*a*) Geben Sie eine geeignete Funktionsdeklaration an.

```
void sort (
   void   *vector,
   size_t number_of_elements,
   size_t size_of_element,
   int    (* compar) (void *element1, void *element2)
);
```

(*b*) Geben Sie anhand der folgenden beiden Beispiele jeweils eine Ordnungsfunktion und den Aufruf Ihrer Sortierfunktion an.

```
int     int_vec[]  = { 4, 2, 7, 3, 9 };

char*str_vec[] = { "gelb", "rot", "blau", "schwarz" };


int_compair(int *element1, int *element2)
{
   return *element1 - *element2;
}

sort(int_vec, sizeof(int_vec) / sizeof(*int_vec),
     sizeof(*int_vec), int_cmp);


str_compair(char **element1, char **element2)
{
   return strcmp(*element1, *element2);
}

sort(str_vec, sizeof(str_vec) / sizeof(*str_vec),
     sizeof(*str_vec), str_cmp);
```

Aufgabe 4

(*a*) Wie kann erreicht werden, daß ein Kindprozeß
die Prozeßnummer seines Vaterprozesses erfährt?

Indem man vor dem `fork()`-Aufruf die Prozeßnummer des aktuellen Prozesses via `getpid()` einer Variable zuweist. Diese wird via `fork()` dem erzugten Kindprozeß vererbt.
Eine Alternative dazu ist, im Kindprozeß den Systemaufruf `getppid()` zu verwenden (*Get Parent Process ID*).

(*b*) Warum ist das `cd`–Kommando unter UNIX ein Shell-Builtin?

Der aktuelle Katalog befindet sich in der *u area* eines Prozesses. Mit `fork()` wird der aktuelle Katalog dem erzeugten Kindprozeß vererbert. Wechselt der Kindprozeß seinen aktuellen Katalog, so wird diese Veränderung natürlich nicht im Vaterprozeß wirksam. Somit muß das `cd`-Kommando, Systemaufruf `chdir()`, vom Shell-Prozeß selbst ausgeführt werden, d.h. die Shell wechselt ihren Arbeitskatalog.

(*c*) Gegeben sei folgender Ausschnitt eines C-Programms:

```
pipe(pfd);
switch(fork()) {
case -1: exit(1);
case  0: close(0); dup(pfd[0]);
         execlp("wc", "wc", (char *) 0);
default: write(pfd[1], "Hallo\n", 6);
         close(pfd[0]); close(pfd[1]);
         wait(&status);
         exit(0);
}
```

Vorausgesetzt sei, daß alle Systemaufrufe erfolgreich verlaufen. Begründen Sie, weshalb das obige Programm nicht terminiert und korrigieren Sie es entsprechend.

Das Problem ist der Filedeskriptor `pfd[1]` im Kindprozeß. Dieser wird über `exec()` hinweg an `wc` vererbt. Zwar schließt der Vaterprozeß das Schreibende der Pipe `pfd[1]`, jedoch bleibt der Filedeskriptor im Kindprozeß weiterhin geöffnet. Am Leseende einer Pipe kann man erst ein `EOF` erhalten, wenn **alle** Filedeskriptoren, die auf das Schreibende verweisen, geschlossen wurden. Dies ist im Kindprozeß nie der Fall.
Abhilfe schafft `close(pfd[1])` vor dem `exec`-Aufruf. Generell gilt: ein mit `exec()` auszuführendes Programm soll lediglich die Filedeskriptoren 0, 1 und 2 geöffnet haben.

Aufgabe 5

In einer UNIX-Shell werden folgende Kommandos eingegeben:

```
$ cat file                Inhalt der Datei file ausgeben
line 1
line 2
$ cat file > file
cat: file: input file is output file
$ cat file                Inhalt der Datei file nochmals ausgeben
$                         Die Datei ist leer
```

(*a*) Begründen Sie, weshalb die Datei `file` leer, also der Dateiinhalt gelöscht wurde.

Die Umlenkung der Standard-Ausgabe wird von der Shell vorgenommen **bevor** das Kommando mit `exec()` zur Ausführung gebracht wird. Die Ausgabedatei wird mit
`open(file, O_CREAT|O_WRONLY|O_TRUNC, mode)`
geöffnet, wodurch eine bereits existierende Datei auf die Länge 0 verkürzt wird (`O_TRUNC`). `cat` bearbeitet somit eine leere Datei.

(*b*) Wie kann das `cat`-Kommando feststellen, daß die Ein- und Ausgabedatei identisch ist? Hinweis: `fstat(2)`

`cat` vergleicht via `fstat` die Inodenummer der Ausgabedatei mit den Inodenummern seiner Eingabedateien und stellt somit eine eventuelle Identität fest.

Aufgabe 6

(*a*) Die Shell Notation `m>&n` lenkt die Ausgabe von Filedeskriptor `m` auf dieselbe Datei wie die Ausgabe von Filedeskriptor `n` um. Worin besteht der Unterschied zwischen den folgenden Kommandos?

```
a.out > filename 2>&1
a.out 2>&1 > filename
```

Hinweis: Die Shell arbeitet die Kommandozeile von links nach rechts ab.

Das erste Kommando lenkt sowohl die Ausgaben auf *stdout* wie auch die auf *stderr* in die Datei `filename` um.
Das zweite Kommando lenkt nur die Ausgaben auf *stdout* in die Datei `filename` um. Die Ausgaben auf *stderr* gehen in die Datei, auf die *stdout* ursprünglich verwies, also im Normalfall an das Terminal.

(*b*) Implementieren Sie eine Funktion, die sowohl die Standard-Ausgabe (*stdout*) wie auch die Diagnose-Ausgabe (*stderr*) in eine Datei umlenkt. Verwenden Sie folgende Schnittstelle:

```
int redirect(char *filename)    /* Returns 1 if OK, 0 on error */
{
int     fd;

if( (fd = open(filename, O_CREAT|O_WRONLY|O_TRUNC, 0666)) < 0)
   return 0;

if(close(1) < 0 || dup(fd) != 1 ||
   close(2) < 0 || dup(fd) != 2 ||
   close(fd) < 0)
   return 0;

return 1;
}
```


Aufgabe 7

Zur Bit-Manipulation einer 4-Byte Integerzahl sind folgende drei Grundoperationen zu realisieren:

(*a*) Setzen eines Bits

```
void set_bit(long *bitset, int index)
{
   *bitset |= (1 << index-1);
}

#define set_bit(bitset, index)    ((bitset) |= (1 << (index)-1))
```

(*b*) Löschen eines Bits

```
void clr_bit(long *bitset, int index);
{
   *bitset &= ~(1 << index-1);
}

#define clr_bit(bitset, index)    ((bitset) &= ~(1 << (index)-1))
```

(*c*) Feststellen, ob ein Bit gesetzt ist

```
int isset_bit(long bitset, int index);
{
   return bitset & (1 << index-1);
}

#define isset_bit(bitset, index)  ((bitset) & (1 << (index)-1))
```

Die Grundoperationen dürfen auch als Makros definiert werden.
Der Index und somit die Bits seien von 1 bis 32 durchnumeriert, wie folgende Abbildung zeigt: