Aufgabe 1: (8 Punkte)

Kreuzen Sie jeweils an, ob die entsprechende Aussage richtig oder falsch ist.
Jede korrekte Antwort gibt 0.5 Punkte, jede falsche Antwort 0.5 Punkte Abzug. Nicht beantwortete Aussagen gehen neutral in die Bewertung ein.

Wollen Sie eine falsch angekreuzte Antwort korrigieren, streichen Sie bitte das Kreuz mit drei waagrechten Strichen durch (☒).

Lesen Sie die Frage genau, bevor Sie antworten.

a) Bewerten Sie die folgenden Aussagen zum Thema Betriebssysteme:

2 Punkte

Richtig	Falsch	
☐	☐	Unteilbare Betriebsmittel müssen immer vom Betriebssystem zugeteilt werden.
☐	☐	Auch im Einprogrammbetrieb kann ein Mehrkernprozessor ausgenutzt werden.
☐	☐	Eine atomare Aktion ist eine primitive oder komplexe Aktion, deren Einzelschritte nach außen sichtbar nur im Verbund stattfinden.
☐	☐	Nur das Betriebssystem kann virtuelle Ressourcen bereit stellen, da dazu privilegierte Instruktionen benötigt werden.

b) Bereich: Prozesszustände

2 Punkte

Richtig	Falsch	
☐	☐	Wartet ein Prozess passiv auf ein Betriebsmittel, befindet er sich im Zustand „blockiert“.
☐	☐	Wartet ein Prozess aktiv auf ein Betriebsmittel, kann er sich im Zustand „bereit“ befinden.
☐	☐	Ein Prozess kann sich selber von „blockiert“ in „bereit“ versetzen.
☐	☐	Es ist kein direkter Übergang von „blockiert“ in „laufend“ möglich.

c) Bereich: POSIX-Systemaufrufe

2 Punkte

Richtig	Falsch	
☐	☐	Das Betriebssystem kehrt stets in den Prozess zurück, der es mittels eines Systemaufrufs aktiviert hat.
☐	☐	Systemaufrufe werden synchron zum Betriebssystemcode ausgelöst.
☐	☐	Der Systemaufruf `exec()` startet das übergebene Programm in einem neuen Prozess.
☐	☐	`fork()` ist besonders, weil es im Normalfall zweimal mit unterschiedlichem Ergebnis aus einem Aufruf zurückkehrt.

d) Bereich: Traps, Exceptions, Interrupts, Signale

2 Punkte

Richtig	Falsch	
☐	☐	Interrupts treten stets synchron zum Programmfluss auf. Sie sind daher nicht unterdrückbar.
☐	☐	Das Betriebssystem läuft aktiv im Hintergrund und überwacht die Prozesse.
☐	☐	Zur Behandlung eines Interrupts kann ein Prozess einen Interrupthandler beim Betriebssystem registrieren.
☐	☐	Das Signal `SIGKILL` führt zwingend zum Beenden des Prozesses und ist nicht anderweitig behandelbar.

Aufgabe 2: Programmieraufgabe – `fu` (18 Punkte)

Schreiben Sie ein Programm `fu` (**f**ile **u**ser), welches diejenigen Prozesse auflistet, die eine bestimmte Datei als Standard-Ausgabe-Datei verwenden.

Das Betriebssystem stellt dazu unter `/proc` ein Dateisystem bereit. Die dortigen Verzeichnisse repräsentieren die laufenden Prozesse. Darin sind die geöffneten Dateien mittels symbolischer Links abgebildet. Die folgende Grafik zeigt einen exemplarischen Ausschnitt des Prozesses 30167, welcher die Datei `/tmp/dev/pts/3` als Standard-Eingabe geöffnet hat.

```
/proc/
├── 30167/
│   ├── exe -> /usr/bin/date
│   └── fd
│       ├── 0 -> /dev/pts/3
│       ├── 1 -> /tmp/now
│       ├── 2 -> /var/log/error.log
│       └── 3 -> /tmp/storage.data
├── 0815/
│   ├── exe -> /usr/bin/cat
│   └── fd
│       ├── 0 -> /tmp/bar
│       ├── 1 -> /tmp/foo
│       └── 2 -> /dev/null
├── 4711/
│   ├── exe -> /usr/bin/echo
│   └── fd
│       ├── 0 -> /dev/pts/2
│       ├── 1 -> /tmp/bar
│       └── 2 -> /dev/null
└── ...
```

Implementieren Sie die Funktionalität in den Funktionen `handle_proc()` und `main()`:

Funktion `handle_proc(char* pid, char* path)`

- Wechseln in das Prozess-Verzeichnis.
- Auslesen des Symlinks, der die Standard-Ausgabe repräsentiert.
- Vergleichen mit `char* path`, Rückgabe `true` falls die Pfade gleich sind.

Funktion `main(int argc, char* argv[])`

- Iterieren über alle Prozesse (`proc`-Verzeichnis)
- Mittels `handle_proc(char* pid, char* path)` für jeden Prozess prüfen, ob dieser `path` als Standard-Ausgabe verwendet. Dabei ist das erste Aufrufargument des Programms als Pfad zu verwenden.
- Falls der Prozess `path` verwendet, dessen Prozess-ID auf die Standard-Ausgabe schreiben.

Fehlerbehandlung
Sollte ein unerwarteter Fehler auftreten, so soll das Programm mitels der Funktion `die(char *msg)` eine Fehlermeldung ausgeben und sich beenden.

Diese Seite darf herausgetrennt werden.

Beispielfunktionsweise:
Aufruf und Ausgabe:

```
student@srabmit:/proc$ fu /tmp/bar
4711
```

Zusicherungen und Annahmen

- Pfade sind maximal 100 Zeichen lang.
- Das Programm wird immer im `/proc`-Verzeichnis ausgeführt.
- Das Programm wird immer mit richtigen Argumenten ausgeführt.

Diese Seite darf herausgetrennt werden.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <dirent.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <errno.h>
#include <string.h>

#define true 1
#define false 0

// Gegeben:

// Beenden mit Fehlerausgabe
void die(char *msg);

// Zu implementieren

// Eine Datei prüfen und behandeln
int handle_proc(char* pid, char *path);

// Die Liste der Prozesse durchlaufen
int main(int argc, char *argv[]);
```

NAME

chdir, fchdir – change working directory

SYNOPSIS

#include <unistd.h>

int chdir(const char ****path*);**
int fchdir(int *fd***);**

DESCRIPTION

chdir() changes the current working directory of the calling process to the directory specified in *path*.
fchdir() is identical to **chdir**(); the only difference is that the directory is given as an open file descriptor.

RETURN VALUE

On success, zero is returned. On error, −1 is returned, and *errno* is set appropriately.

NAME

close – close a file descriptor

SYNOPSIS

#include <unistd.h>

int close(int *fd***);**

DESCRIPTION

close() closes a file descriptor, so that it no longer refers to any file and may be reused.

RETURN VALUE

close() returns zero on success. On error, −1 is returned, and *errno* is set appropriately.

NAME

closedir – close a directory

SYNOPSIS

#include <sys/types.h>
#include <dirent.h>

int closedir(DIR ****dirp*);**

DESCRIPTION

The **closedir**() function closes the directory stream associated with *dirp*.

RETURN VALUE

The **closedir**() function returns 0 on success. On error, −1 is returned, and *errno* is set appropriately.

NAME

execl, execlp, execv, execvp – execute a file

SYNOPSIS

#include <unistd.h>

int execl(const char ****pathname*, const char ****arg***, ... , NULL */);**
int execlp(const char ****file*, const char ****arg***, ... , NULL);**
int execv(const char ****pathname*, char *const** *argv*[]**);**
int execvp(const char ****file*, char *const** *argv*[]**);**

DESCRIPTION

The **exec**() family of functions replaces the current process image with a new process image.

The initial argument for these functions is the name of a file that is to be executed.

The functions can be grouped based on the letters following the "exec" prefix.

l - execl(), execlp()
The *const char *arg* and subsequent ellipses can be thought of as *arg0*, *arg1*, ..., *argn*. The list of arguments *must* be terminated by a null pointer.

By contrast with the 'l' functions, the 'v' functions (below) specify the command-line arguments of the executed program as a vector.

v - execv(), execvp()
The *char *const argv[]* argument is an array of pointers to null-terminated strings that represent the argument list available to the new program. The first argument, by convention, should point to the filename associated with the file being executed. The array of pointers *must* be terminated by a null pointer.

p - execlp(), execvp()
These functions duplicate the actions of the shell in searching for an executable file if the specified filename does not contain a slash (/) character.

RETURN VALUE

The **exec**() functions return only if an error has occurred. The return value is −1, and *errno* is set to indicate the error.

NAME

fork – create a child process

SYNOPSIS

#include <sys/types.h>
#include <unistd.h>

pid_t fork(void);

DESCRIPTION

fork() creates a new process by duplicating the calling process. The new process is referred to as the *child* process. The calling process is referred to as the *parent* process.

The child process is an exact duplicate of the parent process except for the following points:

* The child has its own unique process ID.
* The child's parent process ID is the same as the parent's process ID.

RETURN VALUE

On success, the PID of the child process is returned in the parent, and 0 is returned in the child. On failure, −1 is returned in the parent, no child process is created, and *errno* is set appropriately.

NAME

opendir, fdopendir – open a directory

SYNOPSIS

DIR *opendir(const char *_name_);

DESCRIPTION

The **opendir**() function opens a directory stream corresponding to the directory *name*, and returns a pointer to the directory stream.

RETURN VALUE

The **opendir**() function returns a pointer to the directory stream. On error, NULL is returned, and *errno* is set appropriately.

NAME

readdir – read a directory

SYNOPSIS

#include <dirent.h>

struct dirent *readdir(DIR *_dirp_);

DESCRIPTION

The **readdir**() function returns a pointer to a *dirent* structure representing the next directory entry in the directory pointed to by *dirp*. It returns NULL on reaching the end of the directory or if an error occurred.

The *dirent* structure is defined as follows:

```
struct dirent {
    ino_t          d_ino;      /* i-node number */
    unsigned short d_reclen;   /* Length of this record */
    unsigned char  d_type;     /* Type of file */
    char           d_name[256]; /* Null-terminated filename */ };
```

d_type This field indicates the file type:

DT_DIR	This is a directory.
DT_FIFO	This is a named pipe (FIFO).
DT_LNK	This is a symbolic link.
DT_REG	This is a regular file.
DT_SOCK	This is a UNIX domain socket.

RETURN VALUE

On success, **readdir**() returns a pointer to a *dirent* structure.

If the end of the directory is reached, NULL is returned and *errno* is not changed. If an error occurs, NULL is returned and *errno* is set appropriately.

NAME

readlink – read value of a symbolic link

SYNOPSIS

#include <unistd.h>

ssize_t readlink(const char *_pathname_, char *_buf_, size_t _bufsiz_);

DESCRIPTION

readlink() places the contents of the symbolic link *pathname* in the buffer *buf*, which has size *bufsiz*. **readlink**() does not append a null byte to *buf*. It will (silently) truncate the contents (to a length of *bufsiz* characters), in case the buffer is too small to hold all of the contents.

RETURN VALUE

On success, these calls return the number of bytes placed in *buf*. On error, −1 is returned and *errno* is set.

NAME

strcmp, strncmp – compare two strings

SYNOPSIS

#include <string.h>

int strcmp(char *_s1_, char *_s2_);
int strncmp(char _s1_[.*n*], char _s2_[.*n*], size_t _n_);

DESCRIPTION

The **strcmp**() function compares the two strings *s1* and *s2*.

strcmp() returns an integer indicating the result of the comparison, as follows:

- 0, if the *s1* and *s2* are equal;
- a negative value if *s1* is less than *s2*;
- a positive value if *s1* is greater than *s2*.

The **strncmp**() function is similar, except it compares only the first (at most) *n* bytes of *s1* and *s2*.

RETURN VALUE

The **strcmp**() and **strncmp**() functions return an integer less than, equal to, or greater than zero if *s1* (or the first *n* bytes thereof) is found, respectively, to be less than, to match, or be greater than *s2*.

NAME

strtok – extract tokens from strings

SYNOPSIS

#include <string.h>

char *strtok(char *_str_, const char *_delim_);

DESCRIPTION

The **strtok**() function breaks a string into a sequence of zero or more nonempty tokens. On the first call to **strtok**(), the string to be parsed should be specified in *str*. In each subsequent call that should parse the same string, *str* must be NULL.

The *delim* argument specifies a set of bytes that delimit the tokens in the parsed string. The caller may specify different strings in *delim* in successive calls that parse the same string.

RETURN VALUE

strtok() returns a pointer to the next token, or NULL if there are no more tokens.

```
int handle_proc(char *pid, char* path) {
```

H:

```
int main(int argc, char *argv[]) {
```

M:

Aufgabe 3: Textfragen (19 Punkte)

a) Ein Programm hat sich einen Alarm gestellt, den das Betriebssystem mittels eines Hardware-Timers implementiert. Dieser ist nun abgelaufen. Beschreiben Sie in Stichpunkten den Ablauf. Ist der Ablauf unterdrückbar?

6 Punkte

b) Was bedeutet Teilinterpretation im Kontext eines Betriebssystems? Wann wird das Betriebssystem dazu aktiv?

3 Punkte

c) Betriebsmittelverwaltung: Ein Einkern-Linux-System wird so geändert, das es einem Prozess beim Betreten eines kritischen Abschnitts für dessen Dauer die höchste Priorität aus der Gruppe der möglicherweis konkurrierenden Prozesse zuweist. Hat diese Prioritätsänderung einen Einfluss auf Deadlocks? Begründen und erklären sie!

3 Punkte

d) Es solle ein mathematisches Programm geschrieben werden. Dieses besteht aus vielen parallelisierbaren Teilproblemen. Jedes Teilproblem kann unabhängig von den anderen in einem neuen Thread berechnet werden. Zur Lösung steht eine Ein-Prozessor-Maschine bereit. Das Betriebssystem bietet Kernel-Level Threads, die Standardbibliothek bietet User-Level-Threads.

3 Punkte

Der Gesamtablauf besteht aus drei Phasen:
(1) Einlesen der Aufgabe (I/O);
(2) Parallelisierbare Problemlösung (CPU);
(3) Ergebnis zurückschreiben (I/O).
Entscheiden und begründen Sie: Welche Parallelisierung verwenden Sie für Phase zwei unter den gegebenen Voraussetzungen zur effizienten Problemlösung?

---

e) Ordnen sie den folgenden Echtweltvorgängen ihre entsprechende Nachrichtensemantik zu:

4 Punkte

`Eis am Tresen kaufen`	
`Gruß per Flaschenpost senden`	
`Anmeldung zu einer Klausur`	
`Aufgaben zur Korrektur abgeben`	