Aufgabe 1: Ankreuzfragen (12 Punkte)

a) Einfachauswahlfragen (6 Punkte)

Bei den Einfachauswahlfragen in dieser Aufgabe ist jeweils nur **eine** richtige Antwort eindeutig anzukreuzen. Auf die richtige Antwort gibt es die angegebene Punktzahl.

Wollen Sie eine Antwort korrigieren, streichen Sie bitte die falsche Antwort mit drei waagerechten Strichen durch (⊗) und kreuzen die richtige an.

Lesen Sie die Frage genau, bevor Sie antworten.

I) Zu welchem Zweck werden die Semaphore hier verwendet?

```
void insert( char item ){
  down(&slots);
  Buffer[++in % N] = item;
  up(&elem);
}
```

```
char remove( void ){
  char item;
  down(&elem);
  item = Buffer[++out % N]
  up(&slots);
  return item;
  }
```

- ○ Lost-Update Vermeidung
- ○ Atomare Ausführung
- ○ Koordinierter Zugriff auf konsumierbare Betriebsmittel
- ○ Koordinierter Zugriff auf wiederverwendbare Betriebsmittel

II) Ein Prozess wird vom Zustand `blockiert` in den Zustand `bereit` überführt. Welche Aussage passt zu diesem Vorgang?

- ○ Der Prozess wird wegen eines ungültigen Speicherzugriffs (Segmentation Fault) beendet.
- ○ Der Prozess hat auf Daten von der Festplatte gewartet, die nun verfügbar sind.
- ○ Ein anderer Prozess wurde vom Betriebssystem verdrängt und der erstgenannte Prozess wird nun auf der CPU eingelastet.
- ○ Es ist kein direkter Übergang von blockiert nach bereit möglich.

III) Bitte beantworten Sie, wie häufig Sie die Lehrangebote jeweils wahrgenommen haben. Jede Antwort ist richtig. Es ist ein Kreuz pro Spalte zu setzen.
Optional: Gerne auch ein oder zwei Worte zur Begründung.

Vorlesung	Tafelübung	Gruppenübung
○ 9-7	○ 5-4	○ 5-4
○ 6-4	○ 3-2	○ 3-2
○ 3-1	○ 1	○ 1
○ 0	○ 0	○ 0

b) Mehrfachauswahlfragen (6 Punkte)

In dieser Aufgabe sind jeweils m Aussagen angegeben. Davon sind n ($0 \leq n \leq m$) Aussagen richtig. Kreuzen Sie jeweils an, ob die entsprechende Aussage richtig oder falsch ist.
Jede korrekte Antwort gibt einen halben Punkt, jede falsche Antwort einen halben Minuspunkt. Nicht beantwortetete Aussagen gehen neutral in die Bewertung ein. Eine Teilaufgabe wird minimal mit 0 Punkten gewertet, falsche Antworten wirken sich nicht auf andere Teilaufgaben aus.

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.

I) Welche der genannten Aufgaben gehören zwingend zum Betriebssystemkern?

Richtig	Falsch	
☐	☐	Gerätetreiber
☐	☐	Multiplexing der Hardware
☐	☐	Benutzeroberfläche (z.Ḃ. Shell)
☐	☐	Isolation zwischen Prozessen

II) Welche der genannten Aussagen zum Thema Semaphore sind richtig?

Richtig	Falsch	
☐	☐	Ein Semaphor kann verwendet werden, um gegenseitigen Ausschluss zu implementieren.
☐	☐	Die `down()` Operation kann nur von dem Kontrollfluss aufgerufen werden, der sich gerade im kritischen Abschnitt befindet.
☐	☐	Die `up()` Funktion eines Semaphors erhöht den Zähler der Semaphore falls möglich und wartet sonst bis dies möglich ist.
☐	☐	Ein Semaphor kann zum Signalisieren von Ereignissen verwendet werden.

III) Beurteilen Sie folgende Aussagen zu UNIX-Prozessen

Richtig	Falsch	
☐	☐	Das Betriebssystem überwacht laufend im Hintergrund, dass kein Prozess auf Ressourcen eines Anderen zugreift.
☐	☐	Ein Prozess ist ein virtueller Computer.
☐	☐	Das Betriebssystem interpretiert den Programmcode des Programms.
☐	☐	Das Betriebssystem erweitert den Befehlssatz des realen Prozessors.

Aufgabe 2: Programmieraufgabe – `linkr` (22 Punkte)

Aufgabe 2.1: Dateisysteminteraktion

Schreiben Sie ein Programm `linkr`, welches gegebene Pfade und Unterverzeichnisse auf symbolische Links untersucht.

Die Funktion `void linkr(char *path)` iteriert über die Einträge eines Verzeichnisses.

- Für jeden Datei vom Typ Link soll mittels `print_link(char *path)` der Pfad der Zieldatei ausgelesen und anschließend ausgegeben werden.
- Für jede Datei vom Typ Verzeichnis soll die Funktion `process_subdir(char *path)` aufgerufen werden.
- Andere Dateitypen sollen ignoriert werden.

Das Auslesen und Ausgeben eines Links soll in der Funktion `void print_link(char *path)` implementiert werden. Diese liest den Zielpfad aus dem Link aus. Pfade können maximal 4095 Zeichen lang sein. War das Lesen erfolgreich, so sollen Dateipfad und Zielpfad ausgegeben werden. Im Fehlerfall soll das Programm mit Fehlerausgabe beendet werden.

Die Funktion `void process_subdir(char *path)` kann als gegeben angesehen werden.

Beispiel:

```
bjoe.fiedler@gbs:~ $ ./linkr dir1 dir2
dir1/lnk1 -> /foo/bar
dir2/test -> ../tmp
```

Beachten Sie die beigefügten Man-Pages (Seite 4). Diese geben Hinweise auf die Verwendung von Bibliotheksfunktionen und deren Verhalten. Achten Sie weiterhin darauf, dass Sie eventuelle Fehlersituationen der Bibliotheksfunktionen im Sinne der Aufgabenstellung behandeln und das Programm mit Fehlerausgabe terminieren.

Verwenden Sie zum Bearbeiten dieser Aufgabe die Vorgabe auf der Seite 5. Ergänzen Sie jeweils die fehlenden Teile des Programmtextes, sodass ein Programmablauf im Sinne der Aufgabenstellung entsteht.

closedir(3) Linux Programmer's Manual closedir(3)

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.

opendir(3) opendir(3)

NAME
opendir, fdopendir – open a directory

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

stat() stat()

NAME
stat, fstat, lstat, fstatat – get file status

SYNOPSIS
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
int stat(const char * *pathname*, **struct stat *** *statbuf*);
int lstat(const char * *pathname*, **struct stat *** *statbuf*);

DESCRIPTION
These functions return information about a file, in the buffer pointed to by *statbuf* .

stat() retrieves information about the file pointed to by *pathname*; **lstat**() is identical to **stat**(), except that if *pathname* is a symbolic link, then it returns information about the link itself, not the file that it refers to.

The stat structure contains the following fields:

```
struct stat {
    dev_t     st_dev;     /* ID of device containing file */
    ino_t     st_ino;     /* Inode number */
    mode_t    st_mode;    /* File type and mode */
    nlink_t   st_nlink;   /* Number of hard links */
    uid_t     st_uid;     /* User ID of owner */
    gid_t     st_gid;     /* Group ID of owner */
    dev_t     st_rdev;    /* Device ID (if special file) */
    off_t     st_size;    /* Total size, in bytes */
    blksize_t st_blksize; /* Block size for filesystem I/O */
    blkcnt_t  st_blocks;  /* Number of 512B blocks allocated */

    struct timespec st_atim;  /* Time of last access */
    struct timespec st_mtim;  /* Time of last modification */
    struct timespec st_ctim;  /* Time of last status change */ };
```

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

readdir(3) readdir(3)

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;     /* Inode number */
    off_t       d_off;     /* Not an offset; see below */
    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.

```
void wait_for_subdirs(void);
void print_link(char *path);
void process_subdir(char *path);
void die(char *cause);
int readlink(char *path, char *buf, int bufsize);

void linkr(char *path) {
    fprintf(stderr, "linkr %10d %s\n", getpid(), path);
    [                                          ];
    if ( [                    ] ) {
        die(path);
    }
    struct dirent *ent;
    while( [                                    ] ) {
        [                                  ]; //file_path Speicher
        if (ent->d_name[0] == '.')
            continue;
        strcpy(file_path, path);  //file_path construction
        strcat(file_path, "/");
        strcat(file_path, ent->d_name);
        if ( [                                  ] ) {
            print_link(file_path);
        }
        if ( [                                  ] ) {
            process_subdir(file_path);
        }
    }
    [                              ];
}

void print_link(char *path) {
    [                              ]; //Speicher für den Zielpfad
    int n = [                                                ];
    if ( [                    ] ) {
        printf("%s -> %s\n", path, [                ] );
    }
}
```

Aufgabe 2.2: Prozesse

Schreiben Sie die fehlenden Funktionen `process_subdir(char *path)` und `wait_for_subdirs()` für das `linkr`-Programm.

Die Funktion `process_subdir(char *path)` soll für das übergebene Verzeichnis einen neuen Prozess starten, der die `linkr`-Funktion für dieses ausführt. Der Elternprozess merkt sich jeweils die ID des Kindes und das zu bearbeitende Verzeichnis mittels der gegebenen Funktion `add_job(pid_t child, char *path)`. Nachdem das Verzeichnis abgearbeitet ist, wartet der Prozess mittels `wait_for_subdirs()` darauf, dass die Kindprozesse beendet werden.

Die Funktion `wait_for_subdirs()` soll auf alle Kindprozesse warten. Zu jedem beendeten Prozess soll eine Zeile mit dessen PID und Pfad ausgegeben werden. Die Pfade sind mittels `char * get_job(pid)` abrufbar. Der Speicher mit dem Pfad muss freigegeben werden. Hat `errno` nach dem Warten auf ein Kind den Wert `ECHILD`, so sind keine weiteren Kinder mehr ausstehend.

Beachten Sie die beigefügten Man-Pages (Seite 8). Diese geben Hinweise auf die Verwendung von Bibliotheksfunktionen und deren Verhalten. Achten Sie weiterhin darauf, dass Sie eventuelle Fehlersituationen der Bibliotheksfunktionen im Sinne der Aufgabenstellung behandeln und das Programm mit Fehlermeldung beenden.

Verwenden Sie zum Bearbeiten dieser Aufgabe die Vorgabe auf der Seite 9. Ergänzen Sie jeweils die fehlenden Teile des Programmtextes, sodass ein Programmablauf im Sinne der Aufgabenstellung entsteht.

NAME

exec, execl, execv, execle, execve, execlp, execvp – execute a file

SYNOPSIS

#include <unistd.h>

int execl(const char * *path***, const char *** *arg0***, . . ., const char *** *argn***, char *** /*NULL*/**);**

int execvp (const char * *file***, char *const** *argv[ ]***);**

DESCRIPTION

Each of the functions in the **exec** family overlays a new process image on an old process. The new process image is constructed from an ordinary, executable file. This file is either an executable object file, or a file of data for an interpreter. There can be no return from a successful call to one of these functions because the calling process image is overlaid by the new process image.

When a C program is executed, it is called as follows:

int main (int argc, char *argv[]);

where *argc* is the argument count, and *argv* is an array of character pointers to the arguments themselves. As indicated, *argc* is at least one, and the first member of the array points to a string containing the name of the file.

The *argv* argument is an array of character pointers to null-terminated strings. These strings constitute the argument list available to the new process image. By convention, *argv* must have at least one member, and it should point to a string that is the same as *path* (or its last component). The *argv* argument is terminated by a null pointer.

The *path* argument points to a path name that identifies the new process file.

The *file* argument points to the new process file. If *file* does not contain a slash character, the path prefix for this file is obtained by a search of the directories passed in the **PATH** environment variable (see **environ**(5)).

File descriptors open in the calling process remain open in the new process.

Signals that are being caught by the calling process are set to the default disposition in the new process image (see **signal**(3C)). Otherwise, the new process image inherits the signal dispositions of the calling process.

RETURN VALUES

If a function in the **exec** family returns to the calling process, 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

wait, waitpid – wait for process to change state

SYNOPSIS

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

pid_t wait(int * *wstatus***);**

pid_t waitpid(pid_t *pid***, int *** *wstatus***, int** *options***);**

DESCRIPTION

All of these system calls are used to wait for state changes in a child of the calling process, and obtain information about the child whose state has changed. In the case of a terminated child, performing a wait allows the system to release the resources associated with the child; if a wait is not performed, then the terminated child remains in a "zombie" state (see NOTES below).

wait() and waitpid()

The **wait**() system call suspends execution of the calling thread until one of its children terminates. The call *wait(&wstatus)* is equivalent to:

waitpid(−1, &wstatus, 0);

The **waitpid**() system call suspends execution of the calling thread until a child specified by *pid* argument has changed state. By default, **waitpid**() waits only for terminated children, but this behavior is modifiable via the *options* argument, as described below.

The value of *pid* can be:

−1 meaning wait for any child process.

> 0 meaning wait for the child whose process ID is equal to the value of *pid*.

The value of *options* is an OR of zero or more of the following constants:

WNOHANG return immediately if no child has exited.

WUNTRACED
also return if a child has stopped (but not traced via **ptrace**(2)). Status for *traced* children which have stopped is provided even if this option is not specified.

WCONTINUED (since Linux 2.6.10)
also return if a stopped child has been resumed by delivery of **SIGCONT**.

If *wstatus* is not NULL, **wait**() and **waitpid**() store status information in the *int* to which it points. This integer can be inspected with the following macros (which take the integer itself as an argument, not a pointer to it, as is done in **wait**() and **waitpid**()!):

WIFEXITED(*wstatus*)
returns true if the child terminated normally, that is, by calling **exit**(3) or **_exit**(2), or by returning from main().

WEXITSTATUS(*wstatus*)
returns the exit status of the child. This consists of the least significant 8 bits of the *status* argument that the child specified in a call to **exit**(3) or **_exit**(2) or as the argument for a return statement in main(). This macro should be employed only if **WIFEXITED** returned true.

RETURN VALUE

wait(): on success, returns the process ID of the terminated child; on error, −1 is returned. If no unwaited-for children exist, −1 is returned and *errno* is set to **WCHILD** ;

waitpid(): on success, returns the process ID of the child whose state has changed; if **WNOHANG** was specified and one or more child(ren) specified by *pid* exist, but have not yet changed state, then 0 is returned. On error, −1 is returned.

Each of these calls sets *errno* to an appropriate value in the case of an error.

Vorgabe zu 2.2: Prozesse

```
void die(char *cause);  //Exit with message
void linkr(char *path);
void add_job(int pid, char *path);
char * get_job(pid_t pid);
```

S:

```
    pid_t child;
    int status;
```

W:

Aufgabe 3: Textaufgabe (11 Punkte)

Die folgenden Beschreibungen sollen kurz und prägnant erfolgen (Stichworte, kurze Sätze).

a) Welches Problem kann hier auftreten? Wo tritt das Problem auf? Wie kann man dies vermeiden?

```
unsigned char pos = 0; int results[sizeof(char)];

//thread 1                          //thread 2
do {                                do {
  waitForJob();                       waitForJob();
  int r = doJob();                    int r = doJob();
  results[++pos] = r;                 results[++pos] = r;
} while(1);                         } while(1);
```

b) Ein Programm versucht einen Nullzeiger zu dereferenzieren. Beschreiben Sie in Stichpunkten den Ablauf.

c) Was bedeutet Teilinterpretation im Kontext eines Betriebssystems?

d) Was ist ein UNIX-Prozess, wodurch wird er identifiziert und welche Ressourcen sind mit ihm verknüpft?