Aufgabe 1: (11 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:

3 Punkte

Richtig	Falsch	
☐	☐	Das Betriebssystem erweitert konzeptionell den Befehlssatz des Prozessors.
☐	☐	Betriebssystemdienste laufen zwingend im priviligierten Modus des Prozessors.
☐	☐	Im Einprogrammbetrieb kann ein Mehrkernprozessor nicht ausgenutzt werden.
☐	☐	Virtuelle Hardwareressourcen werden durch Schutzmechanismen (räumlich und zeitlich) voneinander isoliert.
☐	☐	Eine atomare Aktion ist eine primitive oder komplexe Aktion, deren Einzelschritte nach außen sichtbar nur im Verbund stattfinden.
☐	☐	Virtuelle Maschinen können aufgrund der benötigten priviligierten Operationen nur durch das Betriebssystem bereitgestellt werden.

b) Bereich: Prozesszustände

2 Punkte

Richtig	Falsch	
☐	☐	Es ist ein direkter Übergang von „blockiert“ in „bereit“ möglich.
☐	☐	Ein Prozess wird wegen eines ungültigen Speicherzugriffs (Segmentation Fault) beendet, wenn er sich selbst blockiert.
☐	☐	Ein Prozess im Zustand „erzeugt“ kann sich durch Aufrufen des Systemaufrufs `exec()` in „bereit“ versetzen.
☐	☐	Prozesse können direkt von „blockiert“ in „laufend“ überführt werden.

c) Bereich: POSIX-Systemaufrufe

3 Punkte

Richtig	Falsch	
☐	☐	Durch Ausführen eines Programms als Administrator (root) gelangt man in den priviligierten Modus des Prozessors.
☐	☐	Das Programm mit dem an `exec()` übergebene Programmpfad wird durch den aktuellen Prozess ausgeführt.
☐	☐	Ein durch `fork()` erzeugter Prozess erbt konzeptionell alle Ressourcen des Elternprozesses.
☐	☐	`fork()` ist besonders, weil es im Normalfall zweimal mit unterschiedlichem Ergebnis aus einem Aufruf zurückkehrt.
☐	☐	Systemaufrufe sind eine Erweiterung des Befehlssatzes der CPU.
☐	☐	`stat()` liefert den Zustand eines Prozesses (bereit, laufend, ...).

d) Bereich: Dateisysteme

2 Punkte

Richtig	Falsch	
☐	☐	Ein Hardlink kann auf andere Dateisysteme verweisen.
☐	☐	Ein Dateideskriptor repräsentiert eine prozesslokale Zugriffsbefähigung auf eine Datei.
☐	☐	In einem UNIX-Dateisystem sind Dateiobjekte stets in einer Baumstruktur angeordnet.
☐	☐	Hardlinks erlauben es, dasselbe Dateiobjekt in verschiedenen Kontexten mit verschiedenen Berechtigungen einzubinden.

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

1 Punkt

Vorlesung	Tafelübung	Gruppenübung
○ 10-7	○ 5-4	○ 5-4
○ 6-4	○ 3-2	○ 3-2
○ 3-1	○ 1	○ 1
○ 0	○ 0	○ 0

Aufgabe 2: Programmieraufgabe – papro (20 Punkte)

Schreiben Sie ein Programm `papro` (**pa**rallel **pro**cesses), welches Prozesse zur parallelen Verarbeitung von Aufträgen ausnutzt. `papro` erhält bei Aufruf mindestens einen Dateipfad, welcher zeilenweise Aufträge in Form von Shell-Programmaufrufen enthält. Zu Verarbeitung dieser müssen Kindprozesse erzeugt werden.

Die bereitgestellte Funktion `int getNextJob(int fd, job_t* job)` (nicht selber implementieren!) übernimmt das Einlesen der Jobs. Die Rückgabe signalisiert, ob alle Jobs abgearbeitet wurden (0) oder ein neuer Job erfolgreich angenommen wurde (1). Die übergebene Struktur enthält nach dem Aufruf den gelesenen Job in `input` und die einzelnen Teile dessen aufgetrennt (mittels `strtok()`) in `argv`. Der Eintrag `invalid` signalisiert, ob die gelesene Zeile valide für die Ausführung ist (0) oder übersprungen werden muss (1). Die Funktion allokiert Speicher, den Sie an geeigneter Stelle mit `freeJob(job_t *job)` wieder freigeben müssen.

Die Aufträge sollen immer im Hintergrund gestartet werden. Dies heißt, dass `papro` **nicht** auf das Terminieren des Kindprozesses wartet bevor der nächste Job angenommen wird. Jeweils beim Starten eines Kindes soll eine Zeile mit dem zu auszuführenden Befehl und der Prozess ID ausgeben werden.

Bei dem Lesen eines neuen Jobs sollen alle eventuell terminierte Kindprozesse eingesammelt werden. Dies geschieht in der Funktion `bury()`. Für jeden "begrabenen" Kindprozess wird eine Zeile mit der Prozess ID und dem Exitstatus ausgegeben. Um die Aufgabe zu vereinfachen, darf der Elterprozess vor den Kindprozessen terminieren. Fehler hier aufgerufener Funktionen werden ignoriert. Alle Kinder beenden sich normal (nicht durch Signale o.ä.).

Bei nicht sachgemäßem Start soll ein Nutzungshinweis ausgegeben werden.

Sollte ein unerwarteter Fehler auftreten, so soll sich das Programm mit der Funktion die(char *msg) mit einer Fehlermeldung beenden.

Beispielfunktionsweise:
infile-Inhalt:

```
echo 1
cat invalid_file
```

Aufruf und Standard-Out:

```
./job ./infile ...(eventuell >1 infiles)
Started [echo 1] pid=44
Started [cat invalid_file pid=43
Exited pid=44 exitstatus=0
Exited pid=43 exitstatus=1
```


Diese Seite darf herausgetrennt werden.

Diese Seite darf herausgetrennt werden.

Diese Seite darf herausgetrennt werden.

close(2) close(2)

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.

fork(2) fork(2)

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.

exec(3) exec(3)

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.

open(2) open(2)

NAME
open, creat – open and possibly create a file

SYNOPSIS
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int open(const char * *pathname***, int** *flags***);**
int open(const char * *pathname***, int** *flags***, mode_t** *mode***);**
int creat(const char * *pathname***, mode_t** *mode***);**

DESCRIPTION
The **open**() system call opens the file specified by *pathname*. If the specified file does not exist, it may optionally (if **O_CREAT** is specified in *flags*) be created by **open**().

The argument *flags* must include one of the following *access modes*: **O_RDONLY**, **O_WRONLY**, or **O_RDWR**. These request opening the file read-only, write-only, or read/write, respectively.

In addition, zero or more of the following flags can be bitwise-*or*'d in *flags*:
O_APPEND The file is opened in append mode.
O_TRUNC The file is opened in overwrite mode. Existing content is deleted.
O_CREAT If *pathname* does not exist, create it as a regular file.
O_CLOEXEC
Enable the close-on-exec flag for the new file descriptor.

The owner (user ID) of the new file is set to the effective user ID of the process.

The *mode* argument specifies the file mode bits be applied when a new file is created. This argument must be supplied when **O_CREAT** is specified in *flags*; otherwise *mode* is ignored.

A call to **creat**() is equivalent to calling **open**() with *flags* equal to **O_CREAT|O_WRONLY|O_TRUNC**.

RETURN VALUE
on success: return the new file descriptor; on error:return −1, errno is set appropriately.

Diese Seite darf herausgetrennt werden.

Diese Seite darf herausgetrennt werden.

wait(2) wait(2)

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 **ECHILD** ;

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.

Diese Seite darf herausgetrennt werden.

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

//Gegeben:
typedef struct job {
  int invalid;                  // 0=valid, 1=skip this job
  char * input;                 // the full job as received
  size_t input_length;          // size of the mem behind input
  char ** argv;                 // the parsed arguments (after strtok)
} job_t;

int  getNextJob(int fd, job_t *job); //read from filedescriptor, store in job
void die(char* msg);
void freeJob(job_t *job);   //free allocated memory

// Zu implementieren:
int main(int argc, char* argv[]); // Jobs verarbeiten
void bury(void); // Kinder einsammeln
```

```c
void bury(void) { //Kinder einsammeln
```

B:


```c
int main(int argc, char* argv[]) {
  // Usage Message:

  job_t cur_job;
```

M:

Aufgabe 3: Textfragen (14 Punkte)

a) Ein Programm versucht auf eine ungültige Speicheradresse zuzugreifen. Beschreiben Sie in Stichpunkten den Ablauf. Ist der Ablauf unterdrückbar?

6 Punkte

b) Was bedeutet Teilinterpretation im Kontext eines Betriebssystems?

2 Punkte

c) Warum sind allgemeine Hardlinks auf Verzeichnisse üblicherweise verboten? Warum existiert diese Einschränkung bei symbolischen Links nicht?

2 Punkte

d) Welche Voraussetzungen müssen gegeben sein, damit zwei verschiedene Dateien mit dem Namen `klausur.pdf` zeitgleich auf einem Dateisystem existieren können?

2 Punkte

e) Welcher Effekt kann bei kooperativen Planungsverfahren (Scheduling) auftreten? Beschreiben sie stichwortartig das Problem. Tritt das Problem auch bei verdrängenden Planungsverfahren auf?

2 Punkte