Aufgabe 1: Ankreuzfragen (24 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.

a) Bereich: Synchronisation

```
void insert( char item ){
  down(&sem);
  stack.push(item);
  up(&sem);
}
```

```
char remove( void ){
  char item;
  down(&sem);
  item = stack.pop();
  up(&sem);
  return item;
}
```

Richtig Falsch

- ☐ ☐ Der Semaphor `sem` setzt hier koordinierten Zugriff auf konsumierbare Betriebsmittel durch.
- ☐ ☐ Der Semaphor `sem` setzt hier koordinierten Zugriff auf wiederverwendbare Betriebsmittel durch.
- ☐ ☐ Der Zähler eines Semaphors kann konzeptionell nicht negativ werden.
- ☐ ☐ Der Zugriff auf geteilte Daten muss zur Konsistenzerhaltung immer mit einem Semaphor koordiniert werden.

b) Bereich: Prozesszustände

Richtig Falsch

- ☐ ☐ Es ist kein direkter Übergang von “blockiert” in “bereit” möglich.
- ☐ ☐ Prozesse können direkt von “blockiert” in “laufend” überführt werden.
- ☐ ☐ Übergang von “blockiert” in “bereit” bedeutet: Ein anderer Prozess wurde vom Betriebssystem verdrängt und der aktuelle Prozess wird nun auf der CPU eingelastet.
- ☐ ☐ Ein Prozess wird von “blockiert” in “bereit” überführt, wenn dieser z.B. auf Daten von der Festplatte gewartet hat, die nun verfügbar sind.
- ☐ ☐ Ein Prozess wird wegen eines ungültigen Speicherzugriffs (Segmentation Fault) beendet, wenn er sich selber blockiert.
- ☐ ☐ Ein Prozess kann sich selber von “blockiert” in “bereit” überführen, wenn das Ereignis, auf das er wartet, eingetreten ist.

c) Bereich: UNIX-Prozesse

Richtig Falsch

- ☐ ☐ Das Betriebssystem überwacht aktiv laufend im Hintergrund, dass kein Prozess auf Ressourcen eines anderen Prozesses zugreift.
- ☐ ☐ Ein Prozess kann als ein virtueller Computer verstanden werden.
- ☐ ☐ Das Betriebssystem interpretiert den gesamten Programmcode des Programms.
- ☐ ☐ Das Betriebssystem erweitert konzeptionell den Befehlssatz des realen Prozessors.
- ☐ ☐ Ein Prozess wird durch das von ihm ausgeführte Programm identifiziert.
- ☐ ☐ Ein Prozess im Zustand “beendet” (Zombie) kann mit dem Systemaufruf `respawn()` neu gestartet werden.

d) Bereich: POSIX-Systemaufrufe

Richtig Falsch

- ☐ ☐ Der an `exec()` übergebene Funktionszeiger wird durch einen neuen Prozess ausgeführt.
- ☐ ☐ `fork()` ist besonders, weil esim Normalfall zweimal mit unterschiedlichem Ergebnis aus einem Aufruf zurückkehrt.
- ☐ ☐ Der Rückgabewert von `fork()` ist im Elter- und Kindprozess die Prozess-ID des Kindes.
- ☐ ☐ Der `pipe()`-Systemaufruf erzeugt einen bidirektionalen Kommunikationskanal.
- ☐ ☐ Ein durch `fork()` erzeugter Prozess erbt alle Ressourcen des Elternprozesses.
- ☐ ☐ Durch Ausführen eines Programms als Administrator gelangt man in den priviligierten Modus des Prozessors.

e) Bereich: Adressräume

Richtig Falsch

- ☐ ☐ Das Betriebssystem kann in die Adressräume der Prozesse schreibend zugreifen.
- ☐ ☐ Die Prozesse können in den Adressraum des Betriebssystems lesend zugreifen.
- ☐ ☐ Segmentierung ermöglicht es, die gleiche logische Adresse in unterschiedlichen logischen Adressräumen auf die gleiche physikalische Adresse im realen Adressraum mit unterschiedlichen Zugriffsrechten abzubilden.
- ☐ ☐ Segmentierung ermöglicht es, die gleiche logische Adresse in unterschiedlichen logischen Adressräumen auf verschiedene physikalische Adressen im realen Adressraum mit gleichen Zugriffsrechten abzubilden.
- ☐ ☐ Interne Fragmentierung kann mit Verschmelzung und Kompaktifizierung durch das Betriebssystem aufgelöst werden.
- ☐ ☐ Der logische Adressraum kann größer sein als der virtuelle Adressraum.

f) Bereich: Dateisysteme

Richtig	Falsch	
☐	☐	Hierarchische Namensräume ermöglichen die Verwendung des gleichen Namens in verschiedenen Kontexten.
☐	☐	In einem UNIX-Dateisystem sind Dateiobjekte stets in einer Baumstruktur angeordnet.
☐	☐	Ein Hardlink kann verwaisen.
☐	☐	Der Systemaufruf `dup()` erzeugt eine Kopie der angegebenen Datei.
☐	☐	Ein Dateideskriptor repräsentiert eine prozesslokale Befähigung zum Zugriff auf eine Datei.
☐	☐	Verzeichnisse sind spezielle Dateien des Dateisystems, die Namen an Dateiobjekte binden.

g) Bereich: Kommunikation, Signale und Fernaufrufe

Richtig	Falsch	
☐	☐	Ein synchroner Auftrag blockiert den Sender bis zum Eintreffen des Ergebnisses der Berechnung.
☐	☐	Der Sender einer asynchronen Meldung erhält eine Quittung über den Empfang der Nachricht.
☐	☐	Fernaufrufe ermöglichen vollständige Ortstransparenz von Client und Server.
☐	☐	Signale können jederzeit den normalen Programmfluss unterbrechen, um eine Information mitzuteilen.
☐	☐	Ein Prozess kann für jedes mögliche Signal eine eigene Behandlungsfunktionen registrieren.
☐	☐	`sprintf()` ist ein Systemaufruf zum Schreiben in eine Datei. Er wird daher atomar ausgeführt und muss nicht synchronisiert werden.

h) Bereich: Betriebssysteme allgemein

Richtig	Falsch	
☐	☐	Beim Mehrprogrammbetrieb befinden sich stets mehrere Prozesse im Zustand “laufend”.
☐	☐	Multiplexing und Isolation der Hardwareressourcen sind Kernaufgaben eines Betriebssystems.

i) Bereich: Traps und Interrupts

Richtig	Falsch	
☐	☐	Ein Trap wird immer unmittelbar durch eine Aktivität des aktuell laufenden Prozesses ausgelöst.
☐	☐	Ein Trap führt zwingend zum Abbruch des laufenden Prozesses, da dieser einen schwerwiegenden Fehler darstellt.
☐	☐	Speicherzugriffe und Rechenoperationen können einen Trap auslösen.
☐	☐	Systemaufrufe sind im Programmfluss möglich und in die Kategorie Trap einzuordnen.

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

Vorlesung	Tafelübung	Gruppenübung
○ 12–10	○ 6–5	○ 6–5
○ 9 –5	○ 4–3	○ 4–3
○ 4 –1	○ 2–1	○ 2–1
○ 0	○ 0	○ 0

Aufgabe 2: Virtueller Speicher (7 Punkte)

Auf einem Mikrocontroller ist seitenorientierter logischer Adressraum implementiert. Die 20 Bit breiten Adressen sind in 8-Bit Seitennummer und 12-Bit Offset geteilt. Es sind 8 Bit für Attribute im Seitendeskriptor vorgesehen.

a) Vervollständigen Sie die gegebene Skizze zur Abbildung einer von Ihnen gewählten realen Adresse aus der gegebenen logischen Adresse `0x2b01c`.

Logische Adresse	2	b	0	1	c

Reale Adresse					

b) Bestimmen Sie die folgenden Größen: Größe einer Seitentabelle; Größe einer Seite; maximale Größe des logischen Adressraums

Aufgabe 3: Programmieraufgabe – mach (28 Punkte)

Schreiben Sie ein C-Programm `mach`, welches rekursiv den Verzeichnisbaum durchläuft. Falls ein Verzeichnis eine Datei mit dem Namen `Machdatei` enthält, so werden die darin enthaltenen Kommandozeilen ausgeführt.

Der Verzeichnisdurchlauf soll, beginnend vom aktuellen Verzeichnis, in Postordnung (erst die Kinder, dann der aktuelle Knoten) durchgeführt werden.

Die Funktion `mach()` führt die Abarbeitung des aktuellen Verzeichnisses durch. Für jedes enthaltene Unterverzeichnis wird in dieses gewechselt und wiederum die `mach`-Funktion aufgerufen. Verzeichniseinträge, die mit einem Punkt beginnen (z.B. `..`), sollen ignoriert werden. Die Funktion `process_machdatei()` arbeitet schließlich eine im Verzeichnis enthaltene `Machdatei` ab.

Die Funktion `process_machdatei()` öffnet die `Machdatei` im aktuellen Verzeichnis, liest diese zeilenweise ein und lässt jede darin enthaltene Kommandozeile in einem neuen Prozess ausführen. Das Einlesen wird mittels wiederholter Aufrufe der gegebenen Funktion `char ** get_machline(char *filename)` durchgeführt.

Die Funktion `char ** get_machline(char *filename)` verhält sich dabei wie folgt:

- Wird ein Dateiname angegeben, beginnt die Funktion das Einlesen der gegebenen Datei. Wird `NULL` übergeben, so wird die vorherige Datei weitergelesen.
- Rückgabewert ist jeweils ein Zeiger auf ein Feld von Zeichenketten, in denen ein auszuführende Befehl und seine Argumente stehen. Das Feld, sowie die Zeichenketten sind jeweils `0`-terminert.
- Ist keine Zeile mehr vorhanden, wird `NULL` zurückgegeben und `errno` ist auf 0 gesetzt.
- Im Fehlerfall wird `NULL` zurückgegeben und `errno` ist auf den entsprechenden Fehlerwert gesetzt. Ist die gegebene Datei nicht vorhanden, so steht `errno` auf `ENOENT`.

Jedes von der Funktion `get_machline` erhaltene Feld muss mit der Funktion `free_machline(char **machline)` wieder freigegeben werden.

Die Funktion `process_command(char **argv)` übernimmt die Abarbeitung einer einzelnen Kommandozeile. Sie erzeugt dazu jeweils einen Prozess und lässt diesen die übergebene Kommandozeile ausführen.

Die `main()`-Funktion startet den Verzeichnisdurchlauf ausgehend vom aktuellen Verzeichnis. Ist dieser abgeschlossen, so wartet sie auf das Terminieren aller gestarteten Prozesse bevor sie selber terminiert.

Sollte ein unerwarteter Fehler auftreten, so soll sich das Programm mit der Funktion `die(char *hint)` beenden. Die `die`-Funktion soll hierzu den gegebenen Hinweis gefolgt vom formatierten Wert des Fehlers, der in `errno` kodiert steht, ausgeben und dann das Programm mit Fehlerstatus beenden.


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

//Gegeben:

// Einen Eintrag aus einer Machdatei auslesen
char **get_machline(char *filename);

// Einen gelesenen Eintrag aus einer Machdatei freigeben
void free_machline(char **argv);

//Zu implementieren:

// Das aktuelle Verzeichnis bearbeiten
void mach(void);

// Machdatei Öffnen und ausführen
void process_machdatei(void);

// Einen Eintrag der Machdatei ausführen
void process_command(char **argv);

// Fehlerausgabe und Beenden
void die(char *hint);

// Die Verzeichnisdurchlauf starten und auf Kindprozesse warten
int main(int argc, char **argv);
```

NAME

closedir – close a directory

SYNOPSIS

#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

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

perror – print a system error message

SYNOPSIS

#include <stdio.h>

void perror(const char * *s*);

#include <errno.h>

int *errno*;

DESCRIPTION

The **perror**() function produces a message on standard error describing the last error encountered during a call to a system or library function.

First (if *s* is not NULL and *s is not a null byte ('\0')), the argument string *s* is printed, followed by a colon and a blank. Then an error message corresponding to the current value of *errno* and a new-line.

When a system call fails, it usually returns −1 and sets the variable *errno* to a value describing what went wrong.

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 */
    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

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

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

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

DIR *opendir(const char * *name*);
DIR *fdopendir(int *fd*);

DESCRIPTION

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

After a successful call to **fdopendir**(), *fd* is used internally by the implementation, and should not otherwise be used by the application.

RETURN VALUE

The **opendir**() and **fdopendir**() functions return a pointer to the directory stream. On error, NULL is returned, 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 **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.

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

}

void die(char *hint) {

}
```

MD:


```
void mach(void) {

}
```

MA:

```
void process_machdatei(void) {
```

```
}
```

PM:


```
void process_command(char **argv) {
```

```
}
```

PC:

Aufgabe 4: Synchronisation (8 Punkte)

Ein nebenläufiges Programm soll mit geteilten Daten arbeiten. Es sollen hierbei Jobs an mehrere Workerthreads verteilt werden. Die Workerthreads vermerken jeweils ihren Zustand in zwei geteilten Zustandsvariablen.

Ihre Aufgabe ist es, den Zugriff auf diese entsprechend zu schützen. Verwenden Sie dazu geeignete, aus der Vorlesung bekannte Mittel und fügen Sie deren Verwendung im gegebenen (C-ähnlichen) Pseudocode ein.

Es kann angenommen werden, dass alle Zugriffe über die gegebenen Funktionen durchgeführt werden. Initialisierungen sollen in der `init()`-Funktion durchgeführt werden.

Folgendes ist sicherzustellen:

- Es darf immer nur ein Aktivitätsträger auf die Variablen `running_workers` und `idle_workers` zugreifen. Das Verändern dieser beiden Variablen soll zusammen atomar geschehen.
- Die Liste an Jobs ist zu Programmstart leer. Versucht ein Aktiviträtsträger einen Job zu erhalten, während die Jobliste leer ist, so soll dieser blockieren, bis ein Job verfügbar ist.
- Es darf nur ein Aktivitätsträger zeitgleich die Jobliste bearbeiten.

sem_destroy(3) sem_destroy(3)

NAME
sem_destroy – destroy a semaphore

SYNOPSIS
#include <semaphore.h>

int sem_destroy(sem_t ****sem*);**

DESCRIPTION
sem_destroy() destroys the semaphore at the address pointed to by *sem*.

Destroying a semaphore that other processes or threads are currently blocked on (in **sem_wait**(3)) produces undefined behavior.

Using a semaphore that has been destroyed produces undefined results, until the semaphore has been reinitialized using **sem_init**(3).

RETURN VALUE
sem_destroy() returns 0 on success; on error, −1 is returned, and *errno* is set to indicate the error.

---

sem_getvalue(3) sem_getvalue(3)

NAME
sem_getvalue – get the value of a semaphore

SYNOPSIS
#include <semaphore.h>

int sem_getvalue(sem_t ****sem*, int ****sval***);**

DESCRIPTION
sem_getvalue() places the current value of the semaphore pointed to *sem* into the integer pointed to by *sval*.

RETURN VALUE
sem_getvalue() returns 0 on success; on error, −1 is returned and *errno* is set appropriately.

---

sem_init(3) sem_init(3)

NAME
sem_init – initialize an unnamed semaphore

SYNOPSIS
#include <semaphore.h>

int sem_init(sem_t ****sem*, int** *pshared***, unsigned int** *value***);**

DESCRIPTION
sem_init() initializes the unnamed semaphore at the address pointed to by *sem*. The *value* argument specifies the initial value for the semaphore.

The *pshared* argument indicates whether this semaphore is to be shared between the threads of a process (0), or between processes (1). Initializing a semaphore that has already been initialized results in undefined behavior.

RETURN VALUE
sem_init() returns 0 on success; on error, −1 is returned, and *errno* is set appropriately.

---

sem_post(3) sem_post(3)

NAME
sem_post – unlock a semaphore

SYNOPSIS
#include <semaphore.h>

int sem_post(sem_t ****sem*);**

DESCRIPTION
sem_post() increments (unlocks) the semaphore pointed to by *sem*. If the semaphore's value consequently becomes greater than zero, then another process or thread blocked in a **sem_wait**(3) call will be woken up and proceed to lock the semaphore.

RETURN VALUE
sem_post() returns 0 on success; on error, the value of the semaphore is left unchanged, −1 is returned, and *errno* is set to indicate the error.

---

sem_wait(3) sem_wait(3)

NAME
sem_wait, sem_timedwait – lock a semaphore

SYNOPSIS
#include <semaphore.h>

int sem_wait(sem_t ****sem*);**
int sem_trywait(sem_t ****sem*);**

DESCRIPTION
sem_wait() decrements (locks) the semaphore pointed to by *sem*. If the semaphore's value is greater than zero, then the decrement proceeds, and the function returns, immediately. If the semaphore currently has the value zero, then the call blocks until either it becomes possible to perform the decrement (i.e., the semaphore value rises above zero), or a signal handler interrupts the call.

sem_trywait() is the same as **sem_wait**(), except that if the decrement cannot be immediately performed, then call returns an error (*errno* set to **EAGAIN**) instead of blocking.

RETURN VALUE
on success: 0; on error, the value of the semaphore is left unchanged, −1 is returned, and *errno* is set to indicate the error.

ERRORS
EINTR The call was interrupted by a signal handler
EINVAL *sem* is not a valid semaphore.
EAGAIN The operation could not be performed without blocking (sem_trywait() only).

```
                                                                    ;
                                                                    ;
                                                                    ;

void init(char *path) {
                                                                    ;
                                                                    ;
                                                                    ;
}

void set_workers(int running, int finished) {
                                                                    ;
  running_workers = running;
  finished_workers = finished;
                                                                    ;
}
int get_total_workers() {
                                                                    ;
  int result = running_workers + finished_workers;
                                                                    ;
  return result;
}

job * get_job() {
  job * my_job;
                                                                    ;
                                                                    ;
  my_job = job_queue.get();
                                                                    ;
  return my_job;
}

void add_job(job * j) {
                                                                    ;
  job_queue.put(j);
                                                                    ;
                                                                    ;
}
```


Aufgabe 5: Textfragen (15 Punkte)

a) Kann es bei dem hier gegebenen Beispiel zu einer Verklemmung kommen? Begründen Sie stichwortartig unter Verwendung der Bedingungen für einen Deadlock.

```
void * have_dinner(void* param) {
  int me = (int) param;
  for(;;) {
    sem_wait(forks[me]);
    sem_wait(forks[(me+1)%N]);
    eat();
    sem_post(forks[(me+1)%N]);
    sem_post(forks[me]);
  }

void main() {
  int N = 5;
  sem_t forks[N];
  for (int i=0; i < N; ++i) sem_init(&forks[i], 1);
  for (int i=0; i < N; ++i) pthread_create(NULL, NULL, have_dinner, i);
  ...
}
```

---

b) Wie könnte in dem Beispiel der vorherigen Teilaufgabe die Verklemmung verhindert werden? Skizzieren Sie eine Lösung und begründen Sie stichwortartig.

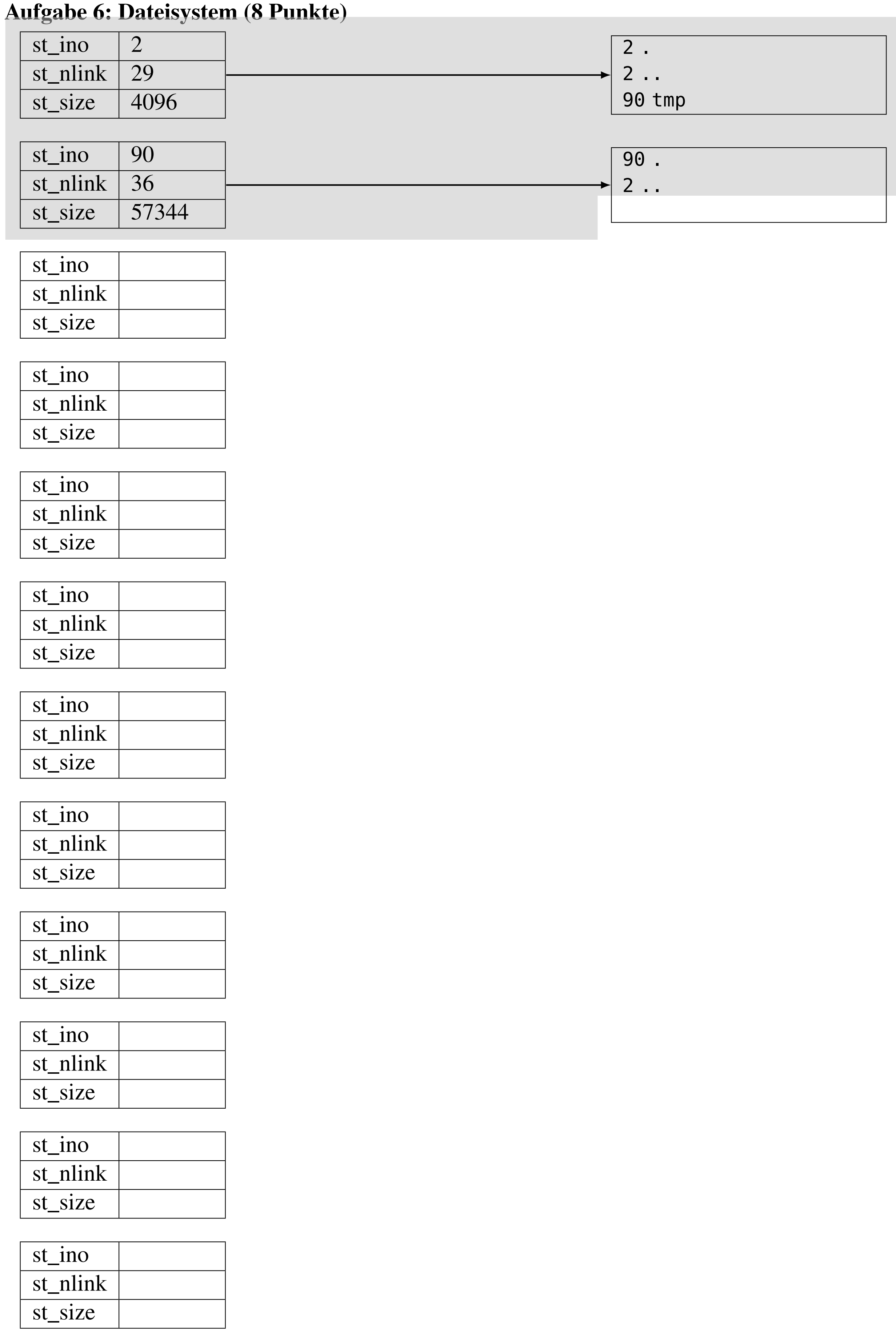
---

c) Welche Klasse von Problemen kann in einem Programmablauf auftreten, wenn asynchrone Signale Teil der Programmlogik werden. Begründen Sie stichwortartig.

d) Warum dürfen nicht alle Bibliotheksfunktionen gefahrlos im Zusammenhang mit Signalen verwendet werden? Begründen Sie stichwortartig und nennen Sie ein Beispiel.

e) Gegeben sei ein Dateisystem mit indizierter Speicherung. Jeder Indexknoten enthält 5 direkte Verweise, und je einen einfach, zweifach und dreifach indirekten Verweis. Eine Adresse ist 4 Byte groß, ein Block 4 KiByte. Wieviele Blöcke werden benötigt, um je eine Datei der Größe 19 KiByte und 23 KiByte darzustellen.

Aufgabe 6: Dateisystem (8 Punkte)



Gegeben ist die folgende Ausgabe des Kommandos `ls -aoRi /tmp/folder` (rekursiv absteigende Ausgabe aller Dateien und Verzeichnisse unter `/tmp/folder` mit Angabe der Inode-Nummer, des Referenzzählers und der Dateigröße) auf einem Linux-System.

```
/tmp/folder:
total 80
  41 drwxr-xr-x  3 bjoern  4096 Mär  4 14:55 .
  90 drwxrwxrwt 36 root   57344 Mär  4 15:56 ..
  42 drwxr-xr-x  2 bjoern  4096 Mär  4 14:53 foo
  71 -rw-r--r--  1 bjoern 10678 Mär  2 18:21 listings.c
  79 lrwxrwxrwx  2 bjoern    10 Mär  2 22:21 rebos -> /gbs/repos

/tmp/folder/foo:
total 44
  42 drwxr-xr-x 2 bjoern  4096 Mär  4 14:53 .
  41 drwxr-xr-x 3 bjoern  4096 Mär  4 14:55 ..
  75 -rw-r--r-- 1 bjoern 34005 Mär  2 18:22 dump.hex
  79 lrwxrwxrwx 2 bjoern    10 Mär  2 22:21 repos -> /gbs/repos
```

Ergänzen Sie im weißen Bereich die auf der Vorlage im grauen Bereich bereits angefangene Skizze der Inodes und Datenblöcke des Linux-Dateisystems um alle entsprechenden Informationen, die aus obiger Ausgabe entnommen werden können.