Aufgabe 1: (14 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 0,5 Punkte, jede falsche Antwort 0,5 Punkte Abzug. Nicht beantwortete 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: Prozesszustände

Richtig	Falsch	
☐	☐	Ein Prozess kann sich selbst von „blockiert“ in „bereit“ überführen, wenn das Ereignis, auf das er wartet, eingetreten ist.
☐	☐	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 im Zustand „erzeugt“ kann sich durch Aufrufen des Systemaufrufs `exec()` in „bereit“ versetzen.
☐	☐	Prozesse können direkt von „blockiert“ in „laufend“ überführt werden.

b) Bereich: Dateisysteme

Richtig	Falsch	
☐	☐	Ein Hardlink kann verwaisen.
☐	☐	Für jede reguläre Datei existiert mindestens ein Hardlink im selben Dateisystem.
☐	☐	Ein Hardlink kann nur auf Verzeichnisse verweisen, nicht jedoch auf Dateien.
☐	☐	In einem UNIX-Dateisystem sind die Objekte immer in einer Baumstruktur angeordnet.

c) Scheduling

Richtig	Falsch	
☐	☐	Ein Prozess, der sich in einem kritischen Abschnitt befindet, kann vom Betriebssystem unterbrochen werden.
☐	☐	Die Umschaltung zwischen User-Threads ist eine privilegierte Operation und muss deshalb im Systemkern erfolgen.
☐	☐	Präemptives Scheduling ermöglicht es, die Monopolisierung der CPU zu verhindern.
☐	☐	Leichtgewichtige Prozesse (kernel-threads) können die Multiprozessorfähigkeit des Betriebssystems ausnutzen.

d) Man unterscheidet Traps und Interrupts (*Unterbrechungen*). Bewerten Sie die folgenden Aussagen:

Richtig	Falsch	
☐	☐	Eine Instruktion hat einen Trap ausgelöst. Es ist möglich, dass der ausführende Prozess die Fehlerursache behebt und fortfährt.
☐	☐	Ein Interrupt wird immer unmittelbar durch eine Aktivität des aktuell laufenden Prozesses ausgelöst.
☐	☐	Der Zeitgeber (Systemuhr) unterbricht die Programmbearbeitung in regelmäßigen Abständen. Die genaue Stelle der Unterbrechungen ist damit vorhersagbar. Somit sind solche Unterbrechungen in die Kategorie Trap einzuordnen.
☐	☐	Der Zugriff auf eine virtuelle Adresse kann zu einem Trap führen.

e) Bereich: UNIX-Prozesse

Richtig	Falsch	
☐	☐	Jedem UNIX-Prozess ist zu jeder Zeit ein aktuelles Arbeitsverzeichnis zugeordnet.
☐	☐	Mit dem Systemaufruf `chdir()` kann das aktuelle Arbeitsverzeichnis eines Prozesses durch seinen Elternprozess verändert werden.
☐	☐	Das aktuelle Arbeitsverzeichnis erbt der Prozess vom aktuellen Benutzer.
☐	☐	Ein Prozess im Zustand „beendet“ (Zombie) kann mit dem Systemaufruf `respawn()` neu gestartet werden.

f) Bewerten Sie die folgenden Aussagen zum Thema Betriebssysteme:

Richtig	Falsch	
☐	☐	Virtuelle Maschinen können aufgrund der benötigten priviligierten Operationen nur durch das Betriebssystem bereitgestellt werden.
☐	☐	Betriebssystemdienste laufen zwingend im priviligierten Modus des Prozessors.
☐	☐	Mehrprogrammbetrieb ermöglicht die simultane Ausführung mehrerer Programme innerhalb eines Prozesses.
☐	☐	Multiplexing und Isolation der Hardwareresourcen sind Kernaufgaben eines Betriebssystems.

g) Bewerten Sie die folgenden Aussagen zu UNIX/Linux-Dateideskriptoren:

Richtig	Falsch	
☐	☐	Das Verzeichnis `.` verweist auf das aktuelle Verzeichnis eines Prozesses. Das Verzeichnis `..` verweist auf dessen übergeordnetes Verzeichnis.
☐	☐	Der Verzeichniseintrag `.` verweist auf das Verzeichnis, in dem der Eintrag selber steht. Der Verzeichniseintrag `..` verweist auf das im Dateibaum übergeordnete Verzeichnis.
☐	☐	Nur Dateideskriptoren mit dem Flag `FD_CLOEXEC` werden beim Prozessende mit `exit()` vom Betriebssystem geschlossen.
☐	☐	Der Aufruf `newfd = dup(fd)` erzeugt eine Kopie der dem Dateideskriptor `fd` zugrundeliegenden Datei; `newfd` enthält einen Dateideskriptor auf die neu erzeugte Datei.

Aufgabe 2: (11 Punkte)

Vervollständigen Sie das folgende Programm, welches eine Simulation einer Fabrik übernimmt. Die Fabrik hat eine Personalleiter **Charles**, der für das Einstellen neuer Mitarbeiter zuständig ist. Die eigentliche Herstellung von Waren wird von mehreren Arbeitern durchgeführt, die zufälligerweise alle **Olaf** heißen und die nach jeder hergestellten Ware mit einer Wahrscheinlichkeit von 1% kündigen. Die fertigen Waren werden von der Verkaufschefin **Clara** verkauft. Zusammen haben sich alle Mitarbeiter der Fabrik darauf verständigt, die Arbeit wie folgt zu koordinieren:

- Charles sorgt dafür, dass immer genug Arbeiter eingestellt werden, wobei er darauf achtet, dass **maximal 16 Olafs** gleichzeitig angestellt sind. Jeder neue Mitarbeiter bekommt bei der Einstellung von Charles eine eindeutige `worker_id` zugewiesen.
- Immer wenn ein Olaf ein Produkt fertiggestellt hat, wird Clara darüber benachrichtigt, damit dieses Produkt in der Verkauf gehen kann.
- Immer wenn ein Olaf ein Produkt hergestellt hat, besucht er die **einzige** Toillete des Betriebes.
- Sobald ein Olaf keine Lust mehr auf die Arbeit hat (er bricht seine Schleife ab), meldet er sich bei Charles ab, sodass ein neuer Arbeiter eingestellt werden kann.
- Wenn Clara erfährt, dass ein Produkt fertig gestellt wurde, geht dieses in den Verkauf.

Ergänzen sie den C-Code so, dass das beschriebene Verhalten erreicht wird. Hierfür müssen Sie die gegebenen Semaphoren so initialisieren und verwenden, dass die Arbeitsabläufe passend synchronisiert werden. Dabei kann es sein, dass einzelnen Felder leer bleiben können. **Streichen Sie in diesem Fall das Feld durch!**

Hinweise:

- Zu Beginn laufen bereits zwei Threads, von denen einer die Funktion `Charles()` ausführt und der andere die Funtkion `Clara()`.
- `hire_Olaf()` startet einen neuen Thread, der die Funktionen `Olaf()` ausführt.
- Ergänzen Sie das folgende Codegerüst so, dass ein vollständig übersetzbares Programm entsteht.


```
sem_t worker_sem;
sem_t product_sem;
sem_t pause_sem;
int hired_workers = 0;

void init() {
  [                                    ];
  [                                    ];
  [                                    ];
}

void Olaf() {
  while (random() % 100 != 0) {
    produce_good();
    [                                  ];
    [                                  ];
    visit_bathroom();
    [                                  ];
  }
  [                                    ];
}
void Charles() {
  while (true) {
    [                                  ];

    [                                  ];
    int worker_id = (hired_workers++);
    [                                  ];
    hire_Olaf(worker_id);
    }
}
void Clara() {
  while (true) {
    [                                  ];
    sell_good();

  }
}
```

Aufgabe 3: (21 Punkte)

Das bestehende Programm `gz` soll netzwerkfähig gemacht werden. Hierzu ist es Ihre Aufgabe ein entsprechendes Vermittlerprogramm `neuland` zu schreiben, dass die Kommunikation und Verwaltung übernimmt. Für jede eingehende Verbindung wird ein Auftrag (=Job) ausgelößt, für den eine neue Instanz des Programms `gz` gestartet werden soll.

Die Aufgabe des Programms `gz` ist es, Datensätze zu verarbeiten. Wie üblich für UNIX-Programme liest `gz` von der Standardeingabe und schreibt auf die Standardausgabe. Ihr Programm `neuland` soll diese Datenströme so umleiten, dass `gz` von dem Netzwerkkanal liest und seine Ausgabe in eine Datei schreibt. Diese Datei ist mit der `UUID` des Jobs als Dateinamen mit den Rechten `0370` im aktuellen Verzeichnis anzulegen. Sie dürfen annehmen, dass eine `UUID` nur exakt einmal auftritt.

Die Funktionen für die Netzerkkommunikation sind bereits gegeben. Die Funktion `init_server()` initialisert den Netzwerksocket und bereitet die gesamte Netzwerkkommunikation entsprechend der Socketspezifikation (`SOCKET_SPEC`) vor, welche neuland über die Kommandozeile erhält. Die Funktion `job_t *get_job()` wartet auf eingehende Netzwerkverbindungen, initiert die Kommunikation und handelt den ersten (und einzigen) Kommandozeilenparameter für den Aufruf von `gz` aus.

Aufrufsyntax: `neuland SOCKET_SPEC`

Implementieren Sie dazu die folgenden Funktionen für `neuland`, um jeweils die beschriebene Funktionalität zu bieten:

Hinweis: Sie benötigen **keine** Socket-spezifischen Funktionen und keine Kenntnisse über den Aufbau von `SOCKET_SPEC`. Dieses wird vollkommen in der gegebenen Funktion `init_server()` gekapselt.

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

- Prüfen der Argumente
- Nutzungsausgabe bei falschem Aufruf
- Initialisieren der Netzwerkkommunikation
- eintreffende Jobs mit `get_job()` entgegennehmen
- für jeden Job mittels `start_worker()` einen `gz`-Prozess starten
- beendete Jobs mittels `collect_finished_workers()` aufräumen

Funktion `void start_worker(job_t *job)`

- Einen Prozess für `gz` erzeugen
- die Ein- und Ausgabe mittels `redirect_cummunication()` umleiten
- `gz` mit dem im Job angegebenen Parameter ausführen
- nicht mehr benötigte Daten und Ressourcen mittels `free_job()` freigeben
- Ausgabe auf die Standardausgabe:
  "`started <PID> with param <PARAM>`"

Funktion `void collect_finished_workers(void)`

- alle bereits beendeten `gz`-Prozesse einsammeln
- für jeden eingesammelten Prozess die folgende Ausgabe auf die Standardausgabe schreiben:
  "`<PID> beendet. Normal beendet: <OK>`"

Funktion `void redirect_communication(job_t *job)`

- Erstellen einer Datei mit dem Namen der `UUID` des aktuellen Jobs und den Rechten `0370` im aktuellen Verzeichnis
- umleiten der Standard**ausgabe** des aktuellen Prozesses in die erstellte Datei
- umleiten der Standard**eingabe** des aktuellen Prozesses, so dass aus dem in der Job-Struktur angegebenen Dateideskriptor gelesen wird

Funktion `void free_job(job_t *job)`

- Freigeben aller Datenstrukturen und Ressourcen, die mit der angegebenen Job-Struktur verbunden sind

Tritt ein nicht erwarteter Fehler auf, so ist das Programm mit Ausgabe einer Fehlermeldung zu beenden. Hierzu steht die Funktion `die(char *msg)` bereit.


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

void die(const char *msg) {
  perror(msg);
  exit(1);
}


typedef struct {
  char uuid[64];   // UUID des Jobs. NULL-terminiert
  char param[256]; // Parameter des aufzurufenden Programms, NULL-terminiert.
  int com;         // Dateideskriptor für die Kommunikation
} job_t;

//Gegeben
/* neuland initialisieren
 *
 * Parameter socket_spec: Zeichenkette, die den Socket spezifiziert
 * Rückgabe: keine
 * Fehler: keine
 */
void init_server(const char * socket_spec);

/* Bereitet einen neuen Job vor.
 *
 * Rückgabe:
 *    solange Jobs eintreffen: Zeiger auf ein job_t
 *    sonst: NULL
 *
 * Fehler: Die Funktion schlägt nicht fehl.
 */
job_t *get_job(void);


// Zu Implementieren:
void start_worker(job_t *job);
void collect_finished_workers(void);
void redirect_communication(job_t *job);
void free_job(job_t *job);
```

```
// Globale Variablen und Deklarationen

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

M:
M:


```
void start_worker(job_t *job) {
```

```
void collect_finished_workers(void) {
```

M:

M:

M:


```
void redirect_communication(job_t *job) {
```

```
void free_job(job_t *job) {
```

Aufgabe 4: (6 Punkte)

Gegeben sei ein Dateisystem mit indizierter Speicherung. Jeder Indexknoten enthält 8 direkte Verweise, und je einen einfach, zweifach und dreifach indirekten Verweis. Eine Adresse ist 5 Byte groß, ein Block 8 KiByte.

a) Wieviele Blöcke werden benötigt, um eine Datei der Größe 8 KiByte darzustellen? Wie werden die Datenblöcke adressiert?

b) Wieviele Blöcke werden benötigt, um eine Datei der Größe 248 KiByte darzustellen? Wie werden die Datenblöcke adressiert?

c) Wie groß kann eine Datei maximal in diesem Dateisystem sein?

Aufgabe 5: (9 Punkte)

Auf einem byteweise adressierenden Mikrocontroller ist segmentierter logischer Adressraum implementiert. Eine logische Adresse ist 16 Bit breit. Es sind 16 Bit Basis, 16 Bit Limit und 11 Bit Attribute im Segmentdeskriptor vorgesehen. Der Opcode enthält 9 Bit für den Segmentnamen.

a) Vervollständigen Sie die gegebene Skizze zur Abbildung einer von Ihnen gewählten realen Adresse aus der gegebenen logischen Adresse `9008`. Der aktuelle Opcode enthält den Wert `1dd` als Segmentname.

Logische Adresse | 9 | 0 | 0 | 8 |

Reale Adresse | | | | |

b) Bestimmen Sie die folgenden Größen: Größe einer Segmenttabelle; maximale Größe eines Segments; maximale Größe des logischen Adressraums

c) Nennes Sie ein alternatives Verfahren, um einen logischen Adressraum zu implementieren. Beschreiben Sie einen Vor/Nachteil gegenüber der segmentierten Implementierung.

Aufgabe 6: (18 Punkte)

a) Beschreiben Sie was geschieht, wenn ein Prozess auf eine ausgelagerte Seite zugreift.

b) Nehmen Sie an, dass das folgende Programmstück in einem Prozess eines UNIX-Systems ausgeführt wird:

```
// ...
int *p = malloc(sizeof(int));
*p = -1;
// ...
```

I) Welcher Fehler kann dabei auftreten? (1 Punkt)

II) Der Fehler wird von einer Hardwarekomponente zuerst detektiert. Welche Komponente ist das? (1 Punkt)

III) Was wird das Betriebssystem mit dem Prozess machen, der den Fehler hervorruft? (1 Punkt)

c) Teilinterpretation: Wann wird das Betriebssystem aktiv? Nenne ein Beispiel für eine zu interpretierende Instruktion.

d) Eine Freispeicherverwaltung gibt Speicher immer in Einheiten der nächstgrößeren Zweierpotenz der angefragten Menge heraus. Entsteht dadurch ein Problem? Wenn ja, welches? Begründen Sie stichwortartig.

e) Worin unterscheiden sich Online- und Offlinescheduling? Nennen Sie je einen Vorteil.

f) Erläutern Sie das Konzept **Semaphor**. Welche Operationen sind auf Semaphore definiert und was tun diese Operationen?

chdir(2) chdir(2)

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.

---

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.

---

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.

---

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.

---

opendir(3) opendir(3)

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.

strstr(3) strstr(3)

NAME

search and manipulate strings

SYNOPSIS

#include <string.h>

char *strstr(const char *_haystack_, const char *_needle_);

size_t strlen(const char *_s_);

#include <dirent.h>

DESCRIPTION

The **strstr**() function finds the first occurrence of the substring *needle* in the string *haystack*.

The **strlen**() function calculates the length of the string pointed to by *s*, excluding the terminating null byte ('\0').

RETURN VALUE

The function **strstr**() returns a pointer to the beginning of the located substring, or NULL if the substring is not found.

The Function **strlen**() returns the number of bytes in the string pointed to by *s*.

readdir(3) readdir(3)

NAME

readdir – read a directory

SYNOPSIS

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.

wait(2) sem_init(3)

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 any child process.

0 any child process whose process group ID is equal to that of the calling process.

> 0 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.

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_);

RETURN VALUE

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).