Aufgabe 1: (14 Punkte)

Vervollständigen Sie das Programm `fitness`, das die Abläufe in einem Fitnessstudio synchronisiert. Zu Beginn des Tages öffnet die Inhaberin das Fitnessstudio und führt die nötige Initialisierung durch, erst danach kommen erste Kunden und Mitarbeitende. Aus Sicherheitsgründen dürfen sich dabei maximal 100 Kunden gleichzeitig im Fitnessstudio befinden. Außerdem kommt es bei der Benutzung des Fitnessstudios zu weiteren Einschränkungen, die beachtet werden müssen.

Alle Kunden möchten gerne die Laufbänder benutzen, davon sind im Fitnessstudio allerdings nur 20 verfügbar. Ist gerade kein Laufband verfügbar, begibt sich der Kunde stattdessen zu den Spinning-Fahrrädern von denen genug für alle verfügbar sind. Dort fährt er eine zufällige Anzahl zwischen 1 und 30 Runden auf dem Fahrrad, geht dann wieder zu den Laufbändern und versucht erneut, eins zu bekommen. Wenn das klappt, läuft er eine Runde, ansonsten fährt er wieder eine zufällige Anzahl Runden auf dem Fahrrad.

Nachdem ein Laufband benutzt wurde, muss es erst gereinigt werden, bevor ein neuer Kunde es benutzen kann. Damit das geschieht, gibt der Kunde nach dem Laufen jemandem aus dem Personal Bescheid, der das Laufband reinigen kann. Wenn das Personal ein Laufband gereinigt hat, gibt es es wieder für einen der nächsten Kunden frei.

Ist ein Kunde fertig mit dem Laufen und hat dem Personal Bescheid gegeben, möchte er gerne noch duschen. Das Fitnessstudio hat allerdings nur eine Dusche, sodass maximal ein Kunde gleichzeitig duschen kann und alle anderen ggf. warten müssen.

Ergänzen Sie das nachfolgende Codegerüst so, dass die oben beschriebenen Abläufe eingehalten werden. Verwenden Sie dafür insbesondere auch die bereits gegebenen Semaphoren und achten Sie, wo nötig, auf korrekte Fehlerbehandlung. Kunden und Personal werden jeweils in eigenen Threads ausgeführt und können in beliebiger Anzahl vorhanden sein.

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_trywait – 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).

```
#define GYM_CAPACITY 100
#define NUM_TREADMILLS 20
#define MAX_SPINS 30

sem_t gym, treadmills, shower, clean;

void owner(){
  sem_init(&gym,        0, [          ]);
  sem_init(&treadmills, 0, [          ]);
  sem_init(&shower,     0, [          ]);
  sem_init(&clean,      0, [          ]);
}

void staff() {
  while(true){
    [                                  ];
    clean_treadmill();
    [                                  ];
  }
}

void client(){
  [                                    ];

  while (true) {
    if ([                              ]) {
      break;
    }

    int spin_count = random_int(1, MAX_SPINS);
    while ([                           ]) {
      spin_one_round();
      [                                ];
    }
  }

  run();
  [                                    ];

  [                                    ];
  shower();
  [                                    ];

  [                                    ];
}
```


Aufgabe 2: (16 Punkte)

2.1: Nennen Sie jeweils das hauptsächliche Planungsziel bei der Prozessplanung im (1) Stapel-, (2) Dialog- und (3) Echtzeitbetrieb.

2.2: Was ist ein Zombieprozess?

2.3: Erläutern Sie horizontale und vertikale Isolation. Mit welchen grundlegenden Mechanismen werden diese jeweils umgesetzt?

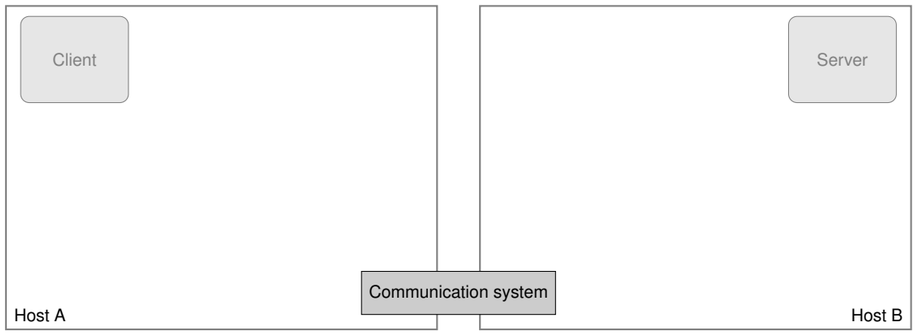
2.4: Im Echtzeitbetrieb wird unterschieden zwischen (1) weicher, (2) fester und (3) harter Echtzeit. Was ist das unterscheidende Merkmal und wie ist dieses Merkmal jeweils ausgeprägt?

2.5: Beschreiben Sie kurz das Verhalten des folgenden Shellprogramms.

```
cd ~
grep V_GBS.txt Speicher > Speicher.txt
chmod a+w Speicher.txt
cat Speicher.txt
```

Aufgabe 3: (10 Punkte)

3.1: Vervollständigen Sie die unten stehende Grafik, um den Ablauf eines Remote Procedure Calls (RPC) zu beschreiben. Achten Sie auch auf Nummerierung der einzelnen Schritte.



3.2: Nennen Sie zwei Einschränkungen von RPCs gegenüber lokalen Funktionsaufrufen.

3.3: Sind Pipes ein geeigneter IPC-Mechanismus zur Implementierung von RPC über ein Netzwerk? Begründen Sie!

Aufgabe 4: (5 Punkte)

Gegeben sei ein Dateisystem mit indizierter Speicherung. Jeder Indexknoten enthält 9 direkte Verweise, und je einen einfach, zweifach und dreifach indirekten Verweis. Eine Adresse ist 4 Byte groß, ein Block 6 KiByte.

4.1: Wieviele Blöcke werden benötigt, um eine Datei der Größe 28 KiByte darzustellen? Wie werden die Datenblöcke adressiert?

4.2: Wieviele Blöcke werden benötigt, um eine Datei der Größe 168 KiByte darzustellen? Wie werden die Datenblöcke adressiert?

4.3: Wie groß kann eine Datei maximal in diesem Dateisystem sein, wenn nur die direkten Verweise genutzt werden dürfen?

Aufgabe 5: (32 Punkte)

Für Experimente sollen Sie das Ausführungswerkzeug `executor` implementieren.

- Ein Experiment besteht aus mehreren Teilschritten (`jobs`), die nacheinander abgearbeitet werden sollen.
- Ein Teilschritt besteht aus einem auszuführenden Programm samt Parameter, einer Ausgabedatei und einer maximalen Bearbeitungszeit.
- Ist die maximale Bearbeitungszeit erreicht, muss der Teilschritt abgebrochen werden.
- Die Gesamtlaufzeit des Experiments ist ebenfalls begrenzt. Das Erreichen der Grenze soll zum Abbruch des aktuellen Teilschritts sowie des Gesamtexperiements führen.
- Alle Ausgaben eines Teilschrittes sind in einer Protokolldatei zu speichern.

Aufrufsyntax: `executor EXPERIMENT_FILE`

Die Parsingfunktion `parse_experiment(char *experiment_file_name)` ist gegeben. Sie öffnet die Experimentbeschreibung und liefert diese als strukturierte Daten.

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

- Prüfen der Anzahl der Parameter
- Nutzungsausgabe, wenn die Anzahl der Parameter falsch ist
- Registrieren des Signalhandlers für Alarme mittels `register_handler()`
- Jeden Teilschritt mittels `create_job(struct job *job)` ausführen
- Alarm für das Gesamtexperiment stellen.

Funktion: `create_job(struct job *job)`:

- Kindprozess für den Teilschritt erzeugen
- Programm des Teilschritts im Kindprozess mittels `execute_job(struct job *job)` einrichten und ausführen
- Auf das Beenden des Teilschrittes mittels `bury()` warten.
- Alarm für den Teilschritt stellen.

Funktion: `execute_job(struct job *job)`:

- Standardausgabe des Teilschrittes in Protokolldatei umleiten
- Falls sie bereits existiert, soll die Protokolldatei überschrieben werden.
- Die Protokolldatei soll nur für den Eigentümer les- und schreibbar sein.
- Programm des Teilschrittes ausführen

Funktion: `register_handler()`:

- Registrieren eines Signalhandlers für das Signal der abgelaufenen Zeit
  - Falls gerade ein Teilschritt als Kindprozess läuft, soll diesem das Signal SIGTERM zugestellt werden.
  - Beenden des aktuellen Prozesses mit Exitcode 14

Funktion: `bury()`:

- Warten auf das derzeitige Kind.
- Die Prozess-ID und den Exitstatus des Kindes ausgeben, wenn es sich normal beendet hat.

Nicht benötigte Resourcen müssen freigegeben werden.

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

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

pid_t job_pid = -1;
#define JP job_pid     /* Für schreibfaule ;-) */

#define READ 0
#define WRITE 1
#define ERROR 2

typedef struct experiment {
  int timeout_sec;
  int num_jobs;
  struct job *jobs;
} experiment_t;

typedef struct job {
  int timeout_sec;
  char *logfile;
  char **argv;
} job_t;


// Gegeben
/* parse and return experiment, null on failure */
experiment_t* parse_experiment(char *file);

// Zu Implementieren:
int main(int argc, char * argv[]);
void register_handler(void);
void create_job(job_t* job);
void execute_job(job_t* job);
void bury(void);
```

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

R:

M:


```
// Signal handling

void register_handler(void) {
```

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

P:

B:

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

I:


```
void bury(void) { // Wait for child termination
  pid_t child;
  int status;
```

Aufgabe 6: (13 Punkte)

Gegeben seien die Prozesse A bis C, sowie eine Liste von Ereignissen mit Zeitpunkten (s. unten). Die Prozesse und Threads werden nach dem Round-Robin-Verfahren gescheduled. Das Scheduling startet zum Zeitpunkt t_0. Zu diesem Zeitpunkt sind alle Prozesse in alphabetischer Reihenfolge zum Scheduling zugelassen. Eine Zeitscheibe ist genau eine Sekunde lang (z.B. t_1-t_2). Außerdem werden die Prozesse auf einem Einprozessorsystem ausgeführt.

6.1: Geben Sie jeweils den Prozesszustand (laufend, bereit, blockiert) an, in dem sich die Prozesse in den entsprechenden Intervallen befinden.

Ereignis	Intervall	Prozess A	Prozess B	Prozess C
	$t_0 - t_{0.5}$			
$t_{0.5}$: A: macht blockierenden syscall	$t_{0.5} - t_1$			
	$t_1 - t_{1.5}$			
	$t_{1.5} - t_2$			
t_2: Ergebnis für A liegt vor	$t_2 - t_{2.5}$			
	$t_{2.5} - t_3$			

6.2: Welches Problem, das generell beim Round-Robin-Verfahren auftreten kann, wird an der Tabelle deutlich? Nennen Sie das Problem und beschreiben Sie kurz die Auswirkung.

6.3: Wie löst Virtual Round-Robin das Problem? Wie würde sich der Schedule in der Tabelle ändern, wenn statt Round-Robin nun Virtual Round-Robin als Schedulingverfahren verwendet wird?

NAME

alarm – set an alarm clock for delivery of a signal

SYNOPSIS

#include <unistd.h>

unsigned int alarm(unsigned int *seconds***);**

DESCRIPTION

alarm() arranges for a **SIGALRM** signal to be delivered to the calling process in *seconds* seconds.

If *seconds* is zero, any pending alarm is canceled.

In any event any previously set **alarm**() is canceled.

RETURN VALUE

alarm() returns the number of seconds remaining until any previously scheduled alarm was due to be delivered, or zero if there was no previously scheduled alarm.

NOTES

alarm() and **setitimer**(2) share the same timer; calls to one will interfere with use of the other.

Alarms created by **alarm**() are preserved across **execve**(2) and are not inherited by children created via **fork**(2).

sleep(3) may be implemented using **SIGALRM**; mixing calls to **alarm**() and **sleep**(3) is a bad idea.

Scheduling delays can, as ever, cause the execution of the process to be delayed by an arbitrary amount of time.

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

dup, dup2 – duplicate a file descriptor

SYNOPSIS

#include <unistd.h>

int dup(int *oldfd***);**
int dup2(int *oldfd***, int** *newfd***);**
int dup3(int *oldfd***, int** *newfd***, int** *flags***);**

DESCRIPTION

The **dup()** system call duplicates a file descriptor.
The **dup2**() system call replaces the file descriptor specified in *newfd* with a copy of *oldfd*. If the file descriptor *newfd* was previously open, it is silently closed before being reused. The steps of closing and reusing the file descriptor *newfd* are performed *atomically*.
The **dup3**() is the same as **dup2**(), except that the **O_CLOEXEC** flag can be specified in *flags*. See the description of the same flag in **open**(2) for reasons why this may be useful.

RETURN VALUE

On success: the new file descriptor. 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

kill – send signal to a process

SYNOPSIS

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

int kill(pid_t *pid***, int** *sig***);**

DESCRIPTION

The **kill**() system call can be used to send any signal to any process.

If *pid* is positive, then signal *sig* is sent to the process with the ID specified by *pid*. If *pid* equals −1, then *sig* is sent to every process for which the calling process has permission to send signals.

If *sig* is 0, then no signal is sent, but existence and permission checks are still performed.

RETURN VALUE

On success (at least one signal was sent), zero is returned. On error, −1 is returned, and *errno* is set appropriately.

ERRORS

EINVAL An invalid signal was specified.
EPERM The process does not have permission to send the signal to any of the target processes.
ESRCH The process or process group does not exist.

NAME

pipe, pipe2 – create pipe

SYNOPSIS

#include <unistd.h>

int pipe(int *pipefd***[2]);**
int pipe2(int *pipefd***[2], int** *flags***);**

DESCRIPTION

pipe() creates a pipe, a unidirectional data channel that can be used for interprocess communication. The array *pipefd* is used to return two file descriptors referring to the ends of the pipe. *pipefd[0]* refers to the read end of the pipe. *pipefd[1]* refers to the write end of the pipe.

If *flags* is 0, then **pipe2**() and **pipe**() are the same. The following values can be bitwise ORed in *flags*:
O_CLOEXEC Enable the close-on-exec flag for the new file descriptor.

RETURN VALUE

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

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 Automatically close this file descriptor on **exec**(2).

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.

NAME

sigaction – examine and change a signal action

SYNOPSIS

#include <signal.h>

int sigaction(int *signum***, const struct sigaction *** *act***,**
struct sigaction * *oldact***);**

DESCRIPTION

The **sigaction**() system call is used to change the action taken by a process on receipt of the signal *signum.*

If *act* is non-NULL, the new action for signal *signum* is installed from *act*. If *oldact* is non-NULL, the previous action is saved in *oldact*.

The *sigaction* structure is defined as follows:

```
struct sigaction {
    void     (*sa_handler)(int);
    ...
};
```

sa_handler specifies the action to be associated with *signum* and may be **SIG_DFL** for the default action, **SIG_IGN** to ignore this signal, or a pointer to a signal handling function. This function receives the signal number as its only argument.

RETURN VALUE

sigaction() returns 0 on success; on error, −1 is returned, and *errno* is appropriately

NAME

signal − overview of signals

DESCRIPTION

Signal dispositions

Each signal has a current *disposition*, which determines how the process behaves on signal delivery.

The entries in the "Action" column of the table below specify the default disposition as follows:

Term Default action is to terminate the process
Ign Default action is to ignore the signal.
Core Default action is to terminate the process and dump core.
Stop Default action is to stop the process.
Cont Default action is to continue the process if it is currently stopped.

A process can change the disposition of a signal using **sigaction**(2) or **signal**(2).

A child created via **fork**(2) inherits a copy of its parent's signal dispositions. During an **execve**(2), the dispositions of handled signals are reset to the default; the dispositions of ignored signals are left unchanged.

Standard signals

Signal	Standard	Action	Comment
SIGABRT	P1990	Core	Abort signal from **abort**(3)
SIGALRM	P1990	Term	Timer signal from **alarm**(2)
SIGBUS	P2001	Core	Bus error (bad memory access)
SIGCHLD	P2001	Ign	Child stopped, terminated, or continued
SIGCONT	P1990	Cont	Continue if stopped
SIGEMT	−	Term	Emulator trap
SIGFPE	P1990	Core	Erroneous arithmetic operation
SIGHUP	P1990	Term	Hangup detected on controlling terminal or death of controlling process
SIGILL	P1990	Core	Illegal Instruction
SIGINT	P1990	Term	Interrupt from keyboard
SIGIO	−	Term	I/O now possible
SIGKILL	P1990	Term	Kill signal
SIGPIPE	P1990	Term	Broken pipe: write to pipe with no readers; see **pipe**(7)
SIGPROF	P2001	Term	Profiling timer expired
SIGPWR	−	Term	Power failure
SIGQUIT	P1990	Core	Quit from keyboard
SIGSEGV	P1990	Core	Invalid memory reference
SIGSTOP	P1990	Stop	Stop process
SIGTSTP	P1990	Stop	Stop typed at terminal
SIGSYS	P2001	Core	Bad system call
SIGTERM	P1990	Term	Termination signal
SIGTRAP	P2001	Core	Trace/breakpoint trap
SIGTTIN	P1990	Stop	Terminal input for background process
SIGTTOU	P1990	Stop	Terminal output for background process
SIGURG	P2001	Ign	Urgent condition on socket
SIGUSR1	P1990	Term	User-defined signal 1
SIGUSR2	P1990	Term	User-defined signal 2
SIGVTALRM	P2001	Term	Virtual alarm clock
SIGXCPU	P2001	Core	CPU time limit exceeded
SIGXFSZ	P2001	Core	File size limit exceeded
SIGWINCH	P2024	Ign	Window resize signal

The signals **SIGKILL** and **SIGSTOP** cannot be caught, blocked, or ignored.

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.