Aufgabe 1: (15 Punkte)

1.1: Worin unterscheiden sich Online- und Offlinescheduling? Nennen Sie je einen Vorteil.

1.2: Was bedeutet Teilinterpretation im Kontext eines Betriebssystems?

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

1.4: Worin unterscheiden Sich die Behandlung der folgenden Trap-Ursachen: (a) Die logische Adresse zeigt auf eine in den Hintergrundspeicher ausgelagerte Seite; (b) Die Adresse ist nicht Teil des logischen Adressraums.

1.5: Das folgende Programmstück wird in einem Prozess eines UNIX-Systems ausgeführt:

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

1.5.1: Welcher Fehler kann dabei auf Ebene E_2 auftreten? (1 Punkt)

1.5.2: Benennen Sie die Hardwarekomponente, die diesen Fehler detektiert. (1 Punkt)

1.5.3: Wie aktiviert diese Komponente das Betriebssystem? (1 Punkt)

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

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

Aufgabe 2: (17 Punkte)

Vervollständigen Sie das Programm `brownie_factory`, welches eine Simulation eines berühmten Süßigkeitenfabrikanten übernimmt.

Die Fabrik stellt für die Produktion Arbeiter ein, die die Schokoladenziegen melken, bis diese keinen Ertrag mehr bringen. Um Protesten der Belegschaft wegen Massenbeschäftigung entgegenzuwirken dürfen maximal 12 Arbeiter zeitgleich angeheuert werden. Implementieren Sie die einzelnen Arbeitsbereiche des Fabrikanten Frank, des Beobachters Jeff und der als Arbeiter eingestellten Koalas.

Per Eule (Signal) kommen zwischendurch neue Börsen- und Einkaufskurse in die Fabrik. Deren konkrete Behandlung soll hier aber nicht implementiert werden. Diese Informationen werden beispielsweise in der Funktion `buy_goat` verwendet.

Der Fabrikant Frank übernimmt die folgenden Aufgaben:

- Er initialisiert alle notwendigen Variablen und Zustände der Fabrik.
- Er lädt Jeff ein, die Fortschritte zu beobachten.
- Solange es neue Ziegen gibt, heuert er Koalas als Arbeiter an (max 12 gleichzeitig) und gibt jedem eine Ziege zum Melken.
- Er wartet darauf, dass die verbleibenden Arbeiter fertig werden.
- Er sagt Jeff Bescheid, dass seine Aufgabe beendet ist.
- Er wartet, dass Jeff seine Aufgabe beendet hat und beendet dann die Simulation.

Der Beobachter Jeff übernimmt folgende Aufgabe:

- Er wartet darauf, dass ein Koala Fortschritt gemacht hat.
- Er liest den Zustand aus.
- Er ruft den Zustand und die Beschäftigungssituation aus (`printf`).
- Wenn Frank nicht das Ende verkündet, wiederholt Jeff sein Vorgehen.
- Er macht eine letzte Zustandsmeldung und sagt Frank, dass er auch fertig ist.

Die Koalas übernehmen folgende Aufgabe:

- Solange die zugewiesene Ziege Schokolade liefert, wird diese gemolken.
- Jeder Teilertrag wird verzeichnet und Jeff gemeldet.
- Liefert die Ziege keinen Ertrag mehr, so kündigt der Koala.

Ergänzen sie den C-Code so, dass das beschriebene Verhalten erreicht wird. Hierfür müssen Sie die gegebenen Semaphoren so verwenden, dass die Arbeitsabläufe passend synchronisiert werden. Hinweise:

- Achten Sie auf eine korrekte Fehlerbehandlung der verwendeten Funktionen.
- Bei geteiltem Zustand muss stets für Konsistenz gesort werden.
- Die Ausgabe von Fehlern soll auf dem `stderr`-Kanal erfolgen.
- Ergänzen Sie das folgende Codegerüst so, dass ein vollständig übersetzbares Programm entsteht.
- `hire_koala()` und `invite_jeff()` starten jeweils neue Threads, die die Funktionen `koala()` und `jeff()` aufrufen.

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.

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

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).

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.

```
#define MAX_KOALAS 12

volatile bool jeff_done = False;
sem_t available_jobs, finished_koalas, jeff_finished;
struct global_statistics{
  long long int brownie;
  sem_t mutex;
  sem_t change;
} statistics = {0};

void hire_koala(goat_t * goat);
void Frank(void);
void invite_jeff(void);

void frank(void) { //called by main()
  //init the factory
  sem_init(&available_jobs,      0, [          ]);
  sem_init(&finished_koalas,     0, [          ]);
  sem_init(&jeff_finished,       0, [          ]);
  sem_init(&statistics.mutex,    0, [          ]);
  sem_init(&statistics.change,   0, [          ]);

  // do regular stuff
  goat_t* cur_goat;
  int job_counter = 0;
  invite_jeff();
  while ((cur_goat = buy_goat()) != NULL) {
    [                                    ];
    [                                    ];
    [                                    ];
  }
  // Wait for Koalas to finish
  [                                      ] {
    [                                    ];
  }
  //Notify Jeff to terminate
  jeff_done = True;
  [                                      ];

  //wait for Jeff to terminate
  [                                      ];
  printf("Everything done. Bye\n");
}
```



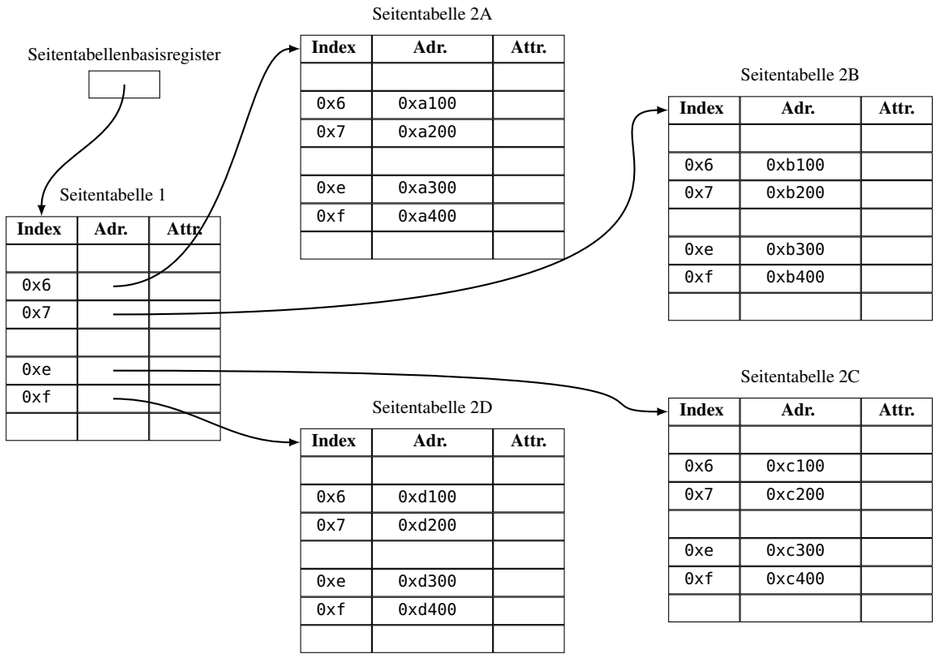
```
void koala(goat_t *goat) {
  int brownie_output;
  while ((brownie_output = milk_goat(goat) != 0)) {
    [                                    ];
    statistics.brownie += brownie_output;
    [                                    ];
    [                                    ];
  }
  release_goat(goat);
  [                                      ];
  [                                      ];
}


void jeff(void) {
  while (!jeff_done) {
    [                                    ];
    [                                    ];
    long long int choc = statistics.brownie;
    [                                    ];
    int koala_slots;
    [                                    ];
    printf("brownie = %lld, workers = %d\n", choc, MAX_KOALAS - koala_slots);
  }
  printf("Well done, we produced %lld units.\n", statistics.brownie);
  [                                      ];
}
```

Annahme: Sie verwenden Prozesse statt der Threads. Nennen Sie einen daraus resultierenden Unterschied in der Interaktion und dessen Ursache.

Aufgabe 3: (13 Punkte)

Auf einem Mikrocontroller mit byteweise adressiertem Speicher ist seitenorientierter logischer Adressraum mit zweistufiger Hierarchie implementiert. Die Adressen sind 16 Bit breit. Pro Stufe werden 4 Bit zur Indizierung verwendet. Die verbleibenden Bits werden als Offset in die Seite verwendet. Es sind außerdem 4 Bit für Attribute im Seitendeskriptor vorgesehen. Die Wortbreite beträgt 2 Byte. Gegeben sei unten dargestellte Hierarchie zweistufiger Seitentabellen.



3.1: Bestimmen sie die **reale Adresse** zur logischen Adresse 0x6e7f. Geben Sie hierbei die zur Bestimmung notwendigen Zwischenschritte stichpunktartig an!

3.2: Nehmen Sie an, dass alle gültigen Seitentabelleneinträge oben abgebildet sind. Was würde in diesem Fall mit einem Prozess passieren, der schreibend auf die Adresse 0xb8b3 zugreift und wieso?

3.3: Bestimmen Sie die folgenden Größen:

- Mindestanzahl Seitentabellen für einen Prozess mit ausführbarem Code und nicht ausführbaren Daten und nur lesbaren Konstanten
- Größe einer Seite
- maximale Größe des logischen Adressraums

3.4: Alternativ zu der gegebenen zweistufigen Implementierung könnte man auch eine einstufige Implementierung mit 8 Bit Seitennummern verwenden. Beschreiben sie einen Vor- und einen Nachteil dieser Alternative.

3.5: Welchen alternativen Mechanismus gibt es zur Implementierung von virtuellem Speicher? Benennen Sie diesen und einen Vor oder Nachteil gegenüber der seitenorientierten Implementierung!

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 6 Byte groß, ein Block 8 KiByte.

4.1: Wieviele Blöcke werden benötigt, um eine Datei der Größe 18 KiByte darzustellen? Wie werden die Datenblöcke adressiert?

4.2: Wieviele Blöcke werden benötigt, um eine Datei der Größe 408 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: (10 Punkte)

5.1: Interprozesskommunikation kann sowohl mittels gemeinsamem Speicher (shared memory) als auch mittels Nachrichtenaustausch (message passing) implementiert werden. Beantworten Sie dazu die folgenden gegenüberstellenden Fragen:

5.1.1: Wie wird dabei die Anwendungslogik der Kommunikationspartner jeweils synchronisiert? (2 Punkte)

5.1.2: Welche Variante eignet sich besser, um Ortstransparenz herzustellen? Begründen Sie ihre Antwort! (2 Punkte)

5.2: Sie haben in der Vorlesung verschiedene Semantiken für nachrichtenbasierte Interprozesskommunikation (IPC) kennen gelernt. Ordnen sie den folgenden Echtweltvorgängen ihre entsprechende Nachrichtensemantik zu:

Durchsage: „Nächster Halt...“	
„Geöffnet“-Schild an die Ladentür hängen	
Abschlussarbeit zur Bewertung in Briefkasten einwerfen	
Kuchen beim Konditor kaufen	
Frage stellen: „Wollen wir gleich ins Kino gehen?“	
Frage in einem Forum stellen	

Aufgabe 6: (30 Punkte)

Auf einem Rechensystem befinden sich diverse Dateien und Verzeichnisse in einer Verzeichnisstruktur. Aus diesen sollen regulären Dateien herausgesucht werden, deren Name `good` enthält. Für jede gefundene Datei soll das Programm `cornet` mit dem Dateinamen als Parameter aufgerufen werden. Alle anderen Dateien sollen ignoriert werden.

Aufrufsyntax: `wiesel DIRECTORY...`

Implementieren Sie das Programm `wiesel`, welches mit einer Liste von Verzeichnissen aufgerufen wird, die rekursiv nach Dateien durchsucht werden sollen. Zur Vereinfachung werden dem Programm `wiesel` stets absolute Pfade übergeben.

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

- Prüfen der Anzahl der Parameter
- Nutzungsausgabe, wenn die Anzahl der Parameter falsch ist
- Iterieren über die angegebenen Verzeichnisse. Für jedes Verzeichnis die Funktion `recurse(char *path)` aufrufen.

Funktion: `recurse(char *path)`

- Verzeichnisabstieg in das übergebene Verzeichnis
- Aufruf von `iterate_current_dir()`, um das betretene Verzeichnis zu durchlaufen.
- Rückkehr in das vorherige Verzeichnis

Funktion: `iterate_current_dir()`

- Iterieren über alle Einträge des aktuellen Verzeichnisses
- versteckte Dateien und Sicherungskopien (Dateiname beginnt mit `"."` oder `"~"`) sollen ignoriert werden
- reguläre Dateien an die Funktion `handle_file(char *filename)` übergeben
- in Verzeichnisse mittels `recurse(char *path)` absteigen

Funktion: `handle_file(char *filename)`

- Prüfen, ob der Dateiname `good` enthält
- Das Programm `cornet` mit dem Dateinamen aufrufen
- Auf das Beenden des `cornet`-Prozesses warten

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

Beispielfunktionsweise:

```
$ ls /foobar/
dir/ file  file1  file2  good  test
```

Aufruf und Standard-Ausgabe:

```
$ ./wiesel /foobar
```

Keine Ausgabe


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

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


// Zu implementieren
void recurse(const char *path);
void handle_file(const char *filename);
void iterate_current_dir(void);
```

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

M:

```
void recurse(const char *path) {
```

R:

C:


```
void iterate_current_dir(void) {
```

```
void handle_file(const char *filename) {
```

H:

opendir(3) chdir(2)

NAME

opendir, fdopendir — open a directory

SYNOPSIS

```
#include <unistd.h>
#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.

RETURN VALUE

The **opendir**() function returns a pointer to the directory stream. On error, NULL is returned, and *errno* is set appropriately.

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.

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 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, **open**() returns a new file descriptor; on error return −1, errno is set appropriately.

closedir(3) 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.

dup(2) dup(2)

opendir(3) opendir(3)

NAME

dup, dup2 − duplicate a file descriptor

fdopendir, fdopendir − open a directory

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 creates a duplicate of the file descriptor *oldfd*. 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.

pipe(3) pipe(3)

NAME

pipe, pipe2, execl, execlp, execv, execvp – execute a file

SYNOPSIS

```
#include <unistd.h>

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

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

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.

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.

RETURN VALUE

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.

readdir(3) readdir(3)

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

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.

fork(2) fork(2)

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.

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, **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.

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

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

search and manipulate strings

SYNOPSIS

#include <string.h>

char *strstr(const char **haystack*, **const char** **needle*);
size_t strlen(const char **s*);

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*.

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.