

1 C Standard Library Overview for Linux

1.1 Introduction

This collection provides a number of examples and cheatsheets for commonly used libc functions and system calls for the Linux operating system. It gives a short description of each function and provides examples for their usage.

Functions are grouped in several categories, which cover specific use cases, respectively. For each category an overview of the involved functions and files is given followed by an example for the typical usage of these functions. After the usage example, a more detailed description of each function is given.

However, cheatsheets **can not** replace the thorough study of the corresponding manpages for more detailed information. Manpages can be retrieved using the following command:

```
man [<section>] <function>
```

Please use the following compiler flags for your exercises. These are the flags used to compile your Linux submissions.

```
CFLAGS = -std=c11 -pedantic -D_XOPEN_SOURCE=700 -Wall -Werror -O3
```

1.2 Cheatsheets (aka SPiC-Zettel)

- [Errno Variable](#)

Explanation of the errno variable.

- [Memory](#)

List of functions: [malloc\(\)](#), [free\(\)](#)

Functions to (de-)allocate memory.

- [Strings](#)

List of functions: [strlen\(\)](#), [strcpy\(\)](#), [strcat\(\)](#)

Functions to analyze and manipulate C strings.

- [File System](#)

List of functions:

- **directory handling:** [opendir\(\)](#), [closedir\(\)](#), [readdir\(\)](#)
- **file handling:** [fopen\(\)](#), [fclose\(\)](#), [stat\(\)](#), [lstat\(\)](#)

Functions for file and directory handling (i.e., opening/closing files and retrieving meta-data).

- [Input/Output](#)

List of functions: [printf\(\)](#), [fprintf\(\)](#), [fgetc\(\)](#), [fgets\(\)](#), [fputc\(\)](#), [fputs\(\)](#), [perror\(\)](#), [feof\(\)](#), [ferror\(\)](#)

Functions for (formatted) input and output.

- [Processes](#)

List of functions: [exit\(\)](#), [wait\(\)](#), [waitpid\(\)](#), [fork\(\)](#), [execl\(\)](#), [execv\(\)](#), [execlp\(\)](#), [execvp\(\)](#), [strtok\(\)](#)

Functions to create new processes and wait for the termination of processes. Furthermore, functions to execute a new program in a process.

- [Signals](#)

List of functions: [kill\(\)](#), [sigemptyset\(\)](#), [sigfillset\(\)](#), [sigaddset\(\)](#), [sigdelset\(\)](#), [sigismember\(\)](#), [sigprocmask\(\)](#), [sigaction\(\)](#), [sigsuspend\(\)](#)

Functions to deliver, synchronize, and wait for POSIX signals in Linux.

- [Threads](#)

List of functions: [pthread_create\(\)](#), [pthread_exit\(\)](#), [pthread_join\(\)](#), [pthread_mutex_init\(\)](#), [pthread_mutex_lock\(\)](#), [pthread_mutex_unlock\(\)](#), [pthread_mutex_destroy\(\)](#)

Functions to create, synchronize, and wait for POSIX threads (pthreads).

2 Module Documentation

2.1 File System

Files

- file [dirent.h](#)
- file [stdio.h](#)
- file [stat.h](#)
- file [types.h](#)

Functions

- DIR * [opendir](#) (const char *name)
Open a directory.
- int [closedir](#) (DIR *dirp)
Close a directory.
- struct dirent * [readdir](#) (DIR *dirp)
Read an entry of a directory.
- FILE * [fopen](#) (const char *pathname, const char *mode)
Open a file.
- int [fclose](#) (FILE *fp)
Close a file.
- int [stat](#) (const char *path, struct stat *buf)
Retrieve metadata of a file.
- int [lstat](#) (const char *path, struct stat *buf)
Retrieve metadata of a file.

2.1.1 Detailed Description

Checking whether a regular file exists, open it, and close it:

```

char *file = "./testfile";

// get file metadata
struct stat sbuf;
if (lstat(file, &sbuf) == -1){
    perror("lstat");
    exit(EXIT_FAILURE);
}

// check file type
if (!S_ISREG(sbuf.st_mode)) {
    fprintf(stderr, "%s is not a regular file.", file);
    exit(EXIT_FAILURE);
}

// open file
FILE *fd = fopen(file, "r+");
if (fd == NULL) {
    perror("fopen");
    exit(EXIT_FAILURE);
}

// use file
// [...]

// close file, check for errors (like full disk)
if (fclose(fd) != 0) {
    perror("fclose");
    exit(EXIT_FAILURE);
}

```

Iterating over all entries of a directory. Be aware, that `readdir()` also returns hidden files (starting with a `.`) including the two entries pointing to the current directory (`.`) and the parent directory (`..`).

```

const char *path = "test/";

// open directory
DIR *dir = opendir(path);
if (dir == NULL) {
    perror("opendir");
    exit(EXIT_FAILURE);
}

// iterate over directory entries
struct dirent *dirent;
while (errno = 0, (dirent = readdir(dir)) != NULL) {
    printf("%s\n", dirent->d_name);
}
if (errno != 0) {
    perror("readdir");
    exit(EXIT_FAILURE);
}

```

```

}

// close directory
closedir(dir);

```

2.1.2 Function Documentation

2.1.2.1 opendir()

```

DIR* opendir (
    const char * name )

```

The `opendir()` function opens a directory stream according to `name`. The stream is positioned at the first entry of the directory. Opened directories must be closed by `closedir()`.

Parameters

<i>name</i>	name of the directory to be opened
-------------	------------------------------------

Return values

<i>DIR*</i>	on success
<i>N</i> ↔ <i>ULL</i>	on error, <code>errno</code> is set

2.1.2.2 closedir()

```

int closedir (
    DIR * dirp )

```

A directory opened by the `opendir()` function, can be closed by the `closedir()` function, which frees all allocated resources.

We do not expect error handling when closing directories, so simply do:

```

closedir(dir);

```

Parameters

<i>dirp</i>	directory stream to be closed
-------------	-------------------------------

2.1.2.3 readdir()

```

struct dirent* readdir (
    DIR * dirp )

```

The `readdir()` function reads the next entry from an opened directory stream pointed to by `dirp`. It allocates a `struct dirent` structure and returns a pointer to the allocated structure con-

taining the information about the next directory entry. The caller of `readdir()` must not provide (or free) memory for the `struct dirent` structure.

`readdir()` returns `NULL` if an error occurs or if the end of the directory stream is reached. To be able to distinguish these two events, a caller must set the `errno` variable to 0 before **each** call of `readdir()`. If `errno` is still 0 after `readdir()` returned `NULL` the end of the directory stream has been reached, otherwise an error has occurred.

The `struct dirent` contains information about a directory entry. The most important information are the inode number and the name of the entry:

```
struct dirent {
    ino_t      d_ino;      // Inode number
    [...]
    char      d_name[256]; // Null-terminated filename
};
```

Parameters

<i>dirp</i>	directory stream to read next entry from
-------------	--

Return values

<i>dirent*</i>	pointer to the next directory entry
<code>NULL</code>	on error (<code>errno</code> is set) or if the end of the directory stream is reached

2.1.2.4 fopen()

```
FILE* fopen (
    const char * pathname,
    const char * mode )
```

The `fopen()` (**file open**) function opens the file at `pathname` with the mode as specified in `mode`. Opened files must be closed by `fclose()`.

The path in `pathname` can specify a relative (based on the current working directory) or an absolute path.

Valid file modes are:

Mode	Description
<code>r</code>	read only
<code>r+</code>	read and write
<code>w</code>	write only, create file if it does not exist yet
<code>w+</code>	read and write file, create file if it does not exist yet
<code>a</code>	write only, append only, create file if it does not exist yet
<code>a+</code>	write append only, read from beginning only, create file if it does not exist yet

Parameters

<i>pathname</i>	path to file
<i>mode</i>	file mode

Return values

<i>FILE*</i>	on success
<i>N↔</i> <i>ULL</i>	on error, <code>errno</code> is set

2.1.2.5 fclose()

```
int fclose (
    FILE * fp )
```

A file opened by the `fopen()` function, can be closed with the `fclose()` (**file close**) function, which writes all remaining buffered operations to the file and frees all allocated resources.

Parameters

<i>fp</i>	file stream to be closed
-----------	--------------------------

Return values

<code>0</code>	on success
<i>EOF</i>	on error, <code>errno</code> is set

2.1.2.6 stat()

```
int stat (
    const char * path,
    struct stat * buf )
```

The `stat()` function retrieves information about the file pointed to by `path`. If `path` is a symbolic link, `stat()` returns information about the underlying file instead of the link itself. Be aware that the caller is responsible to provide the memory for the `struct stat` structure pointed to by `buf`!

The `struct stat` contains, amongst others, the following information:

```
struct stat {
    [...]
    ino_t      st_ino;      // Inode number
    mode_t     st_mode;     // File type and mode
    nlink_t    st_nlink;   // Number of hard links
    uid_t      st_uid;     // User ID of owner
    gid_t      st_gid;     // Group ID of owner
```

```
[...]
off_t    st_size;    // Total size, in bytes
[...]
```

The `st_mode` field encodes the file type and permissions. In order to check whether a file is regular file, a symbolic link, or a directory, some macros exist:

```
struct stat buf;
stat(pathname, &buf);
[...] // error handling
if (S_ISREG(buf.st_mode)) { printf("regular file"); }
if (S_ISDIR(buf.st_mode)) { printf("directory "); }
if (S_ISLNK(buf.st_mode)) { printf("link"); } // only with lstat()
```

Parameters

<code>path</code>	file to be analyzed
<code>buf</code>	pointer to a buffer storing the retrieved information

Return values

0	on success
-1	on error, <code>errno</code> is set

2.1.2.7 lstat()

```
int lstat (
    const char * path,
    struct stat * buf )
```

The `lstat()` function retrieves information about the file pointed to by `path`. If `path` is a symbolic link, `lstat()` returns information about the link itself instead of the underlying file. Be aware that the caller is responsible to provide the memory for the struct `stat` structure pointed to by `buf`!

For more details see the `stat()` function.

Parameters

<code>path</code>	file to be analyzed
<code>buf</code>	pointer to a buffer storing the retrieved information

Return values

0	on success
-1	on error, <code>errno</code> is set



2.2 Errno Variable

Files

- file [errno.h](#)

Variables

- int [errno](#)

Error code set by various library functions.

2.2.1 Detailed Description

The `errno` variable is an integer variable and set by system calls and library functions to indicate the source of an error. The `errno` is undefined, except when a system call or library function indicates an error (e.g., by a special return value) and the corresponding manpage states that in case of an error the `errno` variable is set. From this follows that the value of the `errno` is undefined after a successful call to a system call or library function. Furthermore, no system call or library functions sets the `errno` to 0.

There are rare cases, where the `errno` is not only used to indicate the source of an error, but is also used to detect an error (e.g., `readdir()`). If the `errno` is used to detect an error (and not, as usually, the return value) it is explicitly stated in the manpages. In this case the `errno` must be manually set to 0 before calling the function to be able to check if the function changed the value. Except for these rare cases setting the `errno` manually is never correct, unless one is writing a library function (e.g., `malloc()`).

The `errno` variable is a thread-local variable, which means every POSIX thread has a separate `errno`. Hence, the access of `errno` must not be synchronized against other POSIX threads.

Wrong usage of `errno`:

```
char *s = malloc(1024);
if (s == NULL) {
    fprintf(stderr, "malloc: ");

    // now the errno is undefined, because of a successful
    // or unsuccessful call to fprintf()

    // wrong: strerror() uses an undefined value to generate the string
    fprintf(stderr, "%s\n", strerror(errno));
    exit(EXIT_FAILURE);
}

char *s = malloc(1024);
if (s == NULL) {
    // wrong: original errno value is overwritten!
    errno = ENOMEM;
    perror("malloc");
    exit(EXIT_FAILURE);
}

errno = 0; // wrong: setting errno has no effect here
```



```
char *s = malloc(1024);
if (errno != 0) {
    // wrong: errno can be != 0 even if malloc() is successful
    // NOTE: There are rare exceptions (e.g., readdir()).
    perror("malloc");
    exit(EXIT_FAILURE);
}
```

Correct usage of errno:

```
char *s = malloc(1024);
if (s == NULL) {
    perror("malloc");
    exit(EXIT_FAILURE);
}

errno = 0;
struct dirent *entry = readdir(dirp);
if (entry != NULL) {
    // process entry
} else if (errno != 0) { // explicitly stated in man page
    perror("readdir");
    exit(EXIT_FAILURE);
}
```

2.3 Threads

Files

- file [pthread.h](#)

Functions

- int [pthread_create](#) (pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine)(void *), void *arg)
 - Create a thread.
- void [pthread_exit](#) (void *retval)
 - Exit a thread.
- int [pthread_join](#) (pthread_t thread, void **retval)
 - Wait for a thread.
- int [pthread_detach](#) (pthread_t thread)
 - Detach a thread.
- int [pthread_mutex_init](#) (pthread_mutex_t *mutex, const pthread_mutexattr_t *mutexattr)
 - Create a mutex.
- int [pthread_mutex_lock](#) (pthread_mutex_t *mutex)
 - Lock a mutex.

- int [pthread_mutex_unlock](#) (pthread_mutex_t *mutex)
 - Unlock a mutex.
- int [pthread_mutex_destroy](#) (pthread_mutex_t *mutex)
 - Destroy a mutex.

2.3.1 Detailed Description

This page shows a simplified interface for POSIX threads (pthreads). Threads are a more lightweight method to use the concurrency potential of modern multi-core processors, compared to the process concept of Linux.

Disclaimer: Some parts of the interface are simplified. This page does not replace a thorough study of the manpages for the respective functions!

The pthread_*() function family does not set the errno variable to indicate the error cause, but instead returns an error value (or 0 on success). Thus, the return value of pthread_*() functions can be usually assigned to errno (except otherwise stated) and in case of an error [perror\(\)](#) can be used to print a meaningful error message. Be aware, that the errno is not a global but a thread-local variable, hence each thread has its own errno.

If you intend to use this library, make sure to run your gcc with the appropriate flags.

-pthread -std=c11 -Werror -Wall -pedantic -D_XOPEN_SOURCE=700 -O3

Minimal pthread example:

```
static int counter = 0;

// Function the threads execute
void *thread_func(void *arg) {
    pthread_mutex_t *mutex = (pthread_mutex_t *) arg;

    // do stuff concurrently
    for (unsigned int i = 0; i < 1000; i++) {
        pthread_mutex_lock(mutex);
        counter++;
        pthread_mutex_unlock(mutex);
    }

    pthread_exit(NULL);
}

int main(int argc, char *argv[]) {
    // create mutex
    pthread_mutex_t mutex;
    errno = pthread_mutex_init(&mutex, NULL);
    if (errno != 0) {
        perror("pthread_mutex_init");
        exit(EXIT_FAILURE);
    }

    // create and start threads
    pthread_t threads[4];
```

```

for (unsigned int i = 0; i < 4; i++) {
    errno = pthread_create(&(threads[i]), NULL, thread_func,
                          (void *) &mutex);

    if (errno != 0) {
        perror("pthread_create");
        exit(EXIT_FAILURE);
    }
}

// wait until threads terminate
for (unsigned int i = 0; i < 4; i++) {
    errno = pthread_join(threads[i], NULL);
    if (errno != 0) {
        perror("pthread_join");
        exit(EXIT_FAILURE);
    }
}

pthread_mutex_destroy(&mutex);

printf("counter: %i\n", counter);
}

```

2.3.2 Function Documentation

2.3.2.1 pthread_create()

```

int pthread_create (
    pthread_t * thread,
    const pthread_attr_t * attr,
    void (*)(void *) start_routine,
    void * arg )

```

The `pthread_create()` function creates and starts a new thread within the calling process. The new thread will initially execute the function specified in `start_routine`, which receives a `void *` pointer as parameter and returns a `void *` pointer. The argument handed over to the new thread is specified in `arg`.

The `pthread_create()` function uses the pointer in `thread` to store the thread id in the underlying `pthread_t` variable, which can be used to identify a thread in further `pthread_*` function calls.

The attributes for the new thread are specified in `attr`. For default attributes `NULL` can be used.

A thread that has been created with `pthread_create()` must be joined with `pthread_join()` or marked as detached using `pthread_detach()` in order to free the resources associated with the thread. This is similar to `fork()` and `waitpid()` for processes.

Parameters

<code>thread</code>	pointer to the thread id
<code>attr</code>	thread attributes (<code>NULL</code> for default attributes)
<code>start_routine</code>	function the thread initially executes
<code>arg</code>	argument for <code>start_routine</code>

Return values

0	on success
!=0	on error

2.3.2.2 pthread_exit()

```

void pthread_exit (
    void * retval )

```

The `pthread_exit()` function terminates the calling thread with the return value in `retval`. This function never returns.

Parameters

<code>retval</code>	return value visible to a <code>pthread_join()</code> caller
---------------------	--

2.3.2.3 pthread_join()

```

int pthread_join (
    pthread_t thread,
    void ** retval )

```

The `pthread_join()` function waits for the thread in `thread` to terminate. If the thread has already been terminated, the `pthread_join()` function returns immediately. The return value of the terminated thread can be retrieved by `retval`.

Be aware that threads marked as detached (`pthread_detach()`) can not be joined anymore!

Example code

```

void *retval;

errno = pthread_join(thread, &retval);
if (errno != 0) {
    perror("pthread_join");
    exit(EXIT_FAILURE);
}

printf("Exit code of thread: %p\n", retval);

```

Parameters

<i>thread</i>	thread to wait for
<i>retval</i>	buffer for a pointer to the return value

Return values

0	on success
!=0	on error

2.3.2.4 pthread_detach()

```
int pthread_detach (
    pthread_t thread )
```

The `pthread_detach()` function marks the thread in `thread` detached. This automatically frees all resources, when the threads exits. Once a thread is marked as detached it can not be joined using `pthread_join()` anymore!

Parameters

<i>thread</i>	thread to detach
---------------	------------------

Return values

0	on success
!=0	on error

2.3.2.5 pthread_mutex_init()

```
int pthread_mutex_init (
    pthread_mutex_t * mutex,
    const pthread_mutexattr_t * mutexattr )
```

The `pthread_mutex_init()` function initializes a pthread mutex. It receives a pointer to a `pthread_mutex_t` type and a pointer to the attributes in `mutexattr`. For default attributes NULL can be used for `mutexattr`. If a mutex should be destroyed the `pthread_mutex_destroy()` function can be used.

```
pthread_mutex_t mutex;
errno = pthread_mutex_init(&mutex, NULL);
if (errno != 0) {
    perror("pthread_mutex_init");
    exit(EXIT_FAILURE);
}
```

Parameters

<i>mutex</i>	mutex to be initialized
<i>mutexattr</i>	attributes for mutex (NULL for default)

Return values

0	on success
!=	on error

2.3.2.6 pthread_mutex_lock()

```
int pthread_mutex_lock (
    pthread_mutex_t * mutex )
```

The `pthread_mutex_lock()` function blocks the current thread, until it successfully acquired the mutex in `mutex`. When this function returns, the thread can safely assume to be the only thread inside of the critical section guarded by `mutex`.

Parameters

<i>mutex</i>	mutex to lock
--------------	---------------

2.3.2.7 pthread_mutex_unlock()

```
int pthread_mutex_unlock (
    pthread_mutex_t * mutex )
```

The `pthread_mutex_unlock()` function releases the mutex in `mutex`.

Parameters

<i>mutex</i>	mutex to unlock
--------------	-----------------

2.3.2.8 pthread_mutex_destroy()

```
int pthread_mutex_destroy (
    pthread_mutex_t * mutex )
```

The `pthread_mutex_destroy()` function destroys the mutex in `mutex`. After this function returns, `pthread_mutex_lock()` and `pthread_mutex_unlock()` must **not** be called with this mutex as argument anymore.

Parameters

<i>mutex</i>	mutex to destroy
--------------	------------------

2.4 Signals

Files

- file [signal.h](#)

Data Structures

- struct [sigaction](#)

Functions

- int [kill](#) (pid_t pid, int sig)
 - Send signal to a process.
- int [sigemptyset](#) (sigset_t *set)
 - Empty signal set.
- int [sigfillset](#) (sigset_t *set)
 - Fill signal set.
- int [sigaddset](#) (sigset_t *set, int signum)
 - Add signal to set.
- int [sigdelset](#) (sigset_t *set, int signum)
 - Remove signal from set.
- int [sigismember](#) (const sigset_t *set, int signum)
 - Test signal's membership.
- int [sigprocmask](#) (int how, const sigset_t *set, sigset_t *oset)
 - Change signal mask of a process.
- int [sigaction](#) (int sig, const struct [sigaction](#) *act, struct [sigaction](#) *oact)
 - Set action for a signal.
- int [sigsuspend](#) (const sigset_t *mask)
 - Wait for a signal.

2.4.1 Detailed Description

This set of functions and system calls control the signal handling of processes in Linux. A signal asynchronously interrupts the execution of a process and has great similarities with interrupts as known from the microcontroller programming. This includes the arising problems of asymmetric program interruption, for example, lost wake-up and synchronization problems.

A complete list of all signals and further information can be found at `man 7 signal`. The action values mean, that the default action is either to terminate the process (**Term**), to terminate the process and generate a core dump (**Core**) or to ignore a signal (**Ign**). An excerpt of the available signals is shown here:

Signal	Default Action	Description
SIGINT	Term	interrupt from keyboard (Ctrl-C)

Signal	Default Action	Description
SIGQUIT	Core	quit from keyboard
SIGKILL	Term	kill signal (non blockable)
SIGSEGV	Core	invalid memory reference
SIGALRM	Term	timer signal
SIGTERM	Term	termination signal
SIGUSR1	Term	user-defined signal 1
SIGUSR2	Term	user-defined signal 2
SIGCLD/SIGCHLD	Ign	child stopped/terminated

The following examples show some typical use cases for the presented functions.

Install a new action for SIGINT

```
static void sigint_handler(int signum) { ... }

int main(int argc, char *argv[]) {
    struct sigaction act, oldact;

    // signal mask during handling of a signal
    // (handled signal itself is automatically blocked)
    sigemptyset(&act.sa_mask);

    // set signal handler (also possible: SIG_DFL (default action)
    // and SIG_IGN (ignoring))
    act.sa_handler = sigint_handler;

    // set flags
    act.sa_flags = SA_RESTART;

    if (sigaction(SIGINT, &act, &oldact) == -1) {
        perror("sigaction");
        exit(EXIT_FAILURE);
    }

    [...]
```

Block and unblock a signal:

```
sigset_t set, oldset;

// initialize set (first empty set, then add SIGINT)
sigemptyset(&set);
sigaddset(&set, SIGINT);

// block SIGINT and get previous signal mask
if (sigprocmask(SIG_BLOCK, &set, &oldset) == -1) {
    perror("sigprocmask");
}
```

```

    exit(EXIT_FAILURE);
}

// SIGINT is blocked
[...]

// unblock SIGINT
if (sigprocmask(SIG_UNBLOCK, &set, NULL) == -1) {
    perror("sigprocmask");
    exit(EXIT_FAILURE);
}

```

Only allow one signal

```

sigset_t set, oldset;

// initialize set (first add all signals, then remove SIGINT)
sigfillset(&set);
sigdelset(&set, SIGINT);

// install new signal mask and get previously installed signal mask
if (sigprocmask(SIG_SETMASK, &set, &oldset) == -1) {
    perror("sigprocmask");
    exit(EXIT_FAILURE);
}

// all signals are blocked except for SIGINT
[...]

```

2.4.2 Data Structure Documentation

2.4.2.1 struct sigaction

Data Fields

- `void(* sa_handler)(int)`
- `sigset_t sa_mask`
- `int sa_flags`

Field Documentation

2.4.2.1.1 sa_handler

```
void(* sa_handler) (int)
```

Pointer to the function, which will be installed for the associated signal. The installed function must have one parameter, where the incoming signal is encoded, and no return value. Instead of a pointer to a handler function the two special values `SIG_IGN` (ignore occurrences of this signal) or `SIG_DFL` (restore the default action for this signal) can be used.

2.4.2.1.2 sa_mask

```
sigset_t sa_mask
```

Specifies a signal mask with signals, which are blocked during the handling of the associated signal. The signal itself will be implicitly added to the signal mask (except `SA_NODEFER` is used in `sa_flags`). Usually, an empty signal mask can be used.

2.4.2.1.3 sa_flags

```
int sa_flags
```

Specifies further options for the signal handling process. It is formed by a bitwise OR of zero or more options. Usually, it is set to `SA_RESTART`.

2.4.3 Function Documentation

2.4.3.1 kill()

```
int kill (
    pid_t pid,
    int sig )
```

The `kill()` system call can be used to send the signal specified in `sig` to the process specified in `pid`. The `kill()` function can also be used to send the signal to multiple processes, see man 2 `kill` for more details.

Parameters

<code>pid</code>	pid of the receiving process
<code>sig</code>	signal to be sent

Return values

0	on success
-1	on error, <code>errno</code> is set

2.4.3.2 sigemptyset()

```
int sigemptyset (
    sigset_t * set )
```

The `sigemptyset()` function empties a given signal set.

We do not expect error handling when setting signal masks.

Parameters

<code>set</code>	pointer to the signal set
------------------	---------------------------

2.4.3.3 sigfillset()

```
int sigfillset (
```

```
sigset_t * set )
```

The `sigfillset()` function fills a signal set, that is, all signals are included. We do not expect error handling when setting signal masks.

Parameters

<i>set</i>	pointer to the signal set
------------	---------------------------

2.4.3.4 sigaddset()

```
int sigaddset (
    sigset_t * set,
    int signum )
```

The `sigaddset()` function adds the signal `signum` to the signal set in `set`. We do not expect error handling when setting signal masks.

Parameters

<i>set</i>	pointer to the signal set
<i>signum</i>	signal to be added

2.4.3.5 sigdelset()

```
int sigdelset (
    sigset_t * set,
    int signum )
```

The `sigdelset()` function removes the signal `signum` from the signal set `set`. We do not expect error handling when setting signal masks.

Parameters

<i>set</i>	pointer to the signal set
<i>signum</i>	signal to be removed

2.4.3.6 sigismember()

```
int sigismember (
    const sigset_t * set,
    int signum )
```

The `sigismember()` function determines whether the signal `signum` is a member of the signal set `set`.

Parameters

<i>set</i>	pointer to the signal set
<i>signum</i>	signal to be tested

Return values

1	signal is a member
0	signal is not a member

2.4.3.7 sigprocmask()

```
int sigprocmask (
    int how,
    const sigset_t * set,
    sigset_t * oset )
```

The `sigprocmask()` function is used to manipulate or get the currently installed signal mask. The signal mask is the set of signals that are currently blocked.

The new installed signal mask is specified in the struct pointed to by `act` (`act` can be NULL if no new signal mask should be installed). If `oact` is not NULL the previously installed signal mask is saved.

Instead of setting a new signal mask, the current set can be manipulated by adding or removing the signals specified in `act` depending on the value of `how`. The possible values for `how` are:

Value	Description
SIG_BLOCK	add signals in <code>set</code> to the set of currently blocked signals
SIG_UNBLOCK	remove signals in <code>set</code> from the set of currently blocked signals
SIG_SETMASK	set the set of currently block signals to the signals in <code>set</code>

Parameters

<i>how</i>	determines how the signal mask is changed
<i>set</i>	pointer to the signal set
<i>oset</i>	copy of previous signal set

Return values

0	on success
-1	on error, <code>errno</code> is set

2.4.3.8 sigaction()

```
int sigaction (
    int sig,
    const struct sigaction * act,
    struct sigaction * oact )
```

The `sigaction()` function is used to change the action taken by a process, when receiving a specific signal. For each signal a default action is specified, which can be overwritten by `sigaction()` (except for SIGKILL and SIGSTOP).

The new installed action for the signal `sig` is specified in the struct pointed to by `act` (`act` can be NULL if no new action should be installed). If `oact` is not NULL the previous action is saved. For further information about the content of `act` and `oact` see the documentation of struct `sigaction`.

Parameters

<code>sig</code>	signal to change action for
<code>act</code>	action to take
<code>oact</code>	copy of previous action

Return values

0	on success
-1	on error, <code>errno</code> is set

2.4.3.9 sigsuspend()

```
int sigsuspend (
    const sigset_t * mask )
```

The `sigsuspend()` function temporarily replaces the signal mask of the process with `mask` and then suspends the execution of the process until it receives a signal in an atomic way.

If the signal terminates the process, this function does not return. If the signal is caught, this function returns after the execution of the signal handler and the old signal mask is restored.

The return value of `sigsuspend()` is always -1 and can be ignored.

Parameters

<code>mask</code>	temporary signal mask
-------------------	-----------------------

2.5 Input/Output

Files

- file `stdio.h`

Functions

- int `printf` (const char *format,...)
 - Print formatted data to `stdout`
- int `fprintf` (FILE *stream, const char *format,...)
 - Print formatted data to `stream`.
- int `fgetc` (FILE *stream)
 - Read a character.
- char * `fgets` (char *s, int size, FILE *stream)
 - Read a string.
- int `fputc` (int c, FILE *stream)
 - Write a character.
- int `fputs` (const char *s, FILE *stream)
 - Write a string.
- void `perror` (const char *s)
 - Print an error message.
- int `feof` (FILE *stream)
 - Test end-of-file indicator of a file stream.
- int `ferror` (FILE *stream)
 - Test error indicator of a file stream.

2.5.1 Detailed Description

This code snippet illustrates the usage of `fgetc()` and `fputc()`:

```
int c;
while ((c = fgetc(stdin)) != EOF) {
    if (fputc((unsigned char) c, stdout) == EOF) {
        perror("fputc");
        exit(EXIT_FAILURE);
    }
}
if (ferror(stdin)) {
    // error
}
```

// no error; end of file reached

This code snippet illustrates the usage of `fgets()` and `fputs()`:

```
char buffer[1024];
char *check;

// reads at most 1023 characters per iteration
while ((check = fgets(buffer, 1024, stdin)) != NULL) {
    if (fputs(buffer, stdout) == EOF) {
```

```

    perror("fputs");
    exit(EXIT_FAILURE);
}
if (ferror(stdin)) {
    // error
}

// alternative to ferror():
// if (feof(stdin)) {
//     // no error; end of file reached
// } else {
//     // error
// }

```

2.5.2 Function Documentation

2.5.2.1 printf()

```

int printf (
    const char * format,
    ... )

```

The `printf()` (**print** formatted) function produces output according to a format string as specified in `format` and writes it `stdout`. The format string defines the structure of the output (i.e., how many and which additional arguments) and the additional parameter of the `printf()` function are used to replace the arguments in the format string with actual data. It is important that the number of additional parameter matches the number of arguments of the format string.

An argument in the format string starts with a `%`. The following table shows some important arguments, however, there are several more options described in the manpage of `printf()` (man 3 printf):

Type	Description
i, d	integer
u	unsigned integer
f	floating point
p	pointer
s	string
c	character

Example:

```

int i = 5;
char c = 'a';
void *p = &i;

```

```

printf("Hello World!\n");
printf("$> i: %i\n$> c: %c\n$> f: %f\n \
    $> p: %p\n", i, c, 3.14, p);
// no arguments
// including arguments

// produces:
Hello world!
$> i: 5
$> c: a
$> f: 3.140000
$> p: 0x7ffebe5dbe34

```

We do not expect error handling for `printf()`.

Parameters

<i>format</i>	format string
...	arguments

Return values

>=0	number of characters printed
<0	on error, <code>errno</code> is set

2.5.2.2 fprintf()

```

int fprintf (
    FILE * stream,
    const char * format,
    ... )

```

The `fprintf()` (file stream **print** formatted) function is very similar to `printf()`, except that it does not write to `stdout`, but to `stream`.

For more details see `printf()`.

```

// both lines write to stdout
fprintf(stdout, "Hello world!\n");
printf("Hello world!\n");

// write to stderr
fprintf(stderr, "Error in file '%s' at line %u\n", __FILE__, __LINE__);

```

We do not expect error handling for `fprintf()`.

Parameters

<i>stream</i>	file stream to write
<i>format</i>	format string, see <code>printf()</code>
...	arguments

Return values

<code>>=0</code>	number of characters printed
<code><0</code>	on error, <code>errno</code> is set

2.5.2.3 `fgetc()`

```
int fgetc (
    FILE * stream )
```

The `fgetc()` (file stream **get** character) function returns an unsigned `char` casted to an `int` read from a stream or returns `EOF`.

`fgetc()` returns `EOF` on error or when the end of the file is reached. To distinguish these two events the `feof()` or `ferror()` function can be used. Be aware, that the return value of `fgetc()` must be saved in an `int` (and **not** in an unsigned `char`) in order to distinguish `EOF` and `0xFF` (which is the character `ÿ` when interpreted as ISO-8859-1).

Parameters

<code>stream</code>	file stream to read
---------------------	---------------------

Return values

<code>EOF</code>	on error or end of file, <code>errno</code> is set on error
<code>!=EOF</code>	unsigned <code>char</code> read from stream

2.5.2.4 `fgets()`

```
char* fgets (
    char * s,
    int size,
    FILE * stream )
```

The `fgets()` (file stream **get** string) function reads at most `size-1` characters from `stream` and saves them in the buffer pointed to by `s`. It terminates the string in `s` with a null byte (`\0`).

`fgets()` reads in characters until it finds a newline character (`\n`) or it has read `size-1` characters. If an error occurred or the end of file was reached it returns `NULL`. The `ferror()` and `feof()` functions can be used to distinguish these two events.

Parameters

<code>s</code>	buffer to write to
<code>size</code>	size of the buffer
<code>stream</code>	file stream to read

Return values

<code>NULL</code>	on error or end of file, <code>errno</code> is set on error
<code>!=NULL</code>	pointer to <code>s</code>

2.5.2.5 `fputc()`

```
int fputc (
    int c,
    FILE * stream )
```

The `fputc()` (file stream **put** character) function writes the character `c` to `stream`. We do not expect error handling when using `fputc()`.

Parameters

<code>c</code>	char to print (casted to an <code>int</code>)
<code>stream</code>	file stream to write

Return values

<code>!=EOF</code>	printed char
<code>EOF</code>	on error, <code>errno</code> is set

2.5.2.6 `fputs()`

```
int fputs (
    const char * s,
    FILE * stream )
```

The `fputs()` (file stream **put** string) function writes the string pointed to by `s` to `stream`. We do not expect error handling when using `fputs()`.

Parameters

<code>s</code>	string to be printed
<code>stream</code>	file stream to write

Return values

<code>>=0</code>	on success
<code>EOF</code>	on error, <code>errno</code> is set

2.5.2.7 perror()

```
void perror (
    const char * s )
```

The `perror()` (print **error**) function produces an error message on `stderr` printing the string in `s` followed by a human-readable description of the value in `errno`. This is especially helpful after a failed call to a system or library function, which sets the `errno` variable (e.g., `malloc()`).

Parameters

<code>s</code>	string printed before the actual error message
----------------	--

2.5.2.8 feof()

```
int feof (
    FILE * stream )
```

The `feof()` (file stream **end of file**) function tests the EOF indicator of `stream`.

Parameters

<code>stream</code>	file stream to test
---------------------	---------------------

Return values

0	end-of-file indicator is not set
!=0	end-of-file indicator is set

2.5.2.9 ferror()

```
int ferror (
    FILE * stream )
```

The `ferror()` (file stream **error**) function tests the error indicator of `stream`.

Parameters

<code>stream</code>	file stream to test
---------------------	---------------------

Return values

0	error indicator is not set
!=0	error indicator is set

2.6 Memory

Files

- file `stdlib.h`

Functions

- void * `malloc` (size_t size)
Allocate memory.
- void `free` (void *ptr)
Free allocated memory.

2.6.1 Detailed Description

With `malloc()` (memory **allocation**) a program can request memory from the operating system. The allocated memory is usable until it is `free()`'d again. It is important, that programs always check if a `malloc()` call was successful and free their memory after usage, otherwise a so-called memory leak exists.

```
// allocate memory
char *s = malloc(strlen("Hello World\n") + 1);
if (s == NULL) {
    perror("malloc");
    exit(EXIT_FAILURE);
}
```

```
// use allocated memory
strcpy(s, "Hello World\n");
printf("%s", s);
```

```
// free allocated memory
free(s);
```

2.6.2 Function Documentation

2.6.2.1 malloc()

```
void* malloc (
    size_t size )
```

The `malloc()` function allocates `size` bytes at the heap and returns a pointer to the allocated memory or `NULL` if an error has been occurred. Be aware, that the memory is not initialized after a successful call to `malloc()`.

Parameters

<code>size</code>	number of bytes to be allocated
-------------------	---------------------------------

Return values

<code>NULL</code>	on error, <code>errno</code> is set
<code>!=NULL</code>	pointer to allocated memory

2.6.2.2 free()

```
void free (
    void * ptr )
```

The `free()` function frees the memory pointed to by `ptr`, which must have been returned by a previous call to `malloc()`. Otherwise, or if `free(ptr)` has already been called before, undefined behavior occurs. If `ptr` is `NULL`, no operation is performed.

Parameters

<code>ptr</code>	pointer to buffer
------------------	-------------------

2.7 Processes

Files

- file `stdlib.h`
- file `string.h`
- file `types.h`
- file `wait.h`
- file `unistd.h`

Functions

- void `exit` (int status)
 - Terminate process.*
- char * `strtok` (char *str, const char *delim)
 - Tokenize string.*
- pid_t `wait` (int *wstatus)
 - Wait for a child process.*
- pid_t `waitpid` (pid_t pid, int *wstatus, int options)
 - Wait for a child process.*
- pid_t `fork` (void)
 - Fork new process.*
- int `execl` (const char *path, const char *argo, ..., NULL)
 - Execute a program.*
- int `execv` (const char *path, char *const argv[])
 - Execute a program.*
- int `execlp` (const char *file, const char *argo, ..., NULL)
 - Execute a program.*
- int `execvp` (const char *file, char *const argv[])
 - Execute a program.*

2.7.1 Detailed Description

Example of how to create a new process, which executes a program, and wait for the new process to terminate.

```
// create new process
pid_t pid = fork();
if (pid == 0) {
    // child executes new program
    execlp("ls", "ls", "-A", NULL);
    // execlp() only returns on error
    perror("exec");
    exit(EXIT_FAILURE);
} else if (pid < 0) {
    // fork had an error
    perror("fork");
    exit(EXIT_FAILURE);
}

// parent waits for child to terminate
int status;
if (waitpid(pid, &status, 0) < 0) {
    perror("waitpid");
    exit(EXIT_FAILURE);
}
```

The `exec*()` function family allows for the execution of a new executable within a process. The differences are:

Function	Searches PATH	Array of Arguments	List of Arguments
<code>execl()</code>	X	X	✓
<code>execlp()</code>	✓	X	✓
<code>execv()</code>	X	✓	X
<code>execvp()</code>	✓	✓	X

2.7.2 Function Documentation

2.7.2.1 exit()

```
void exit (
    int status )
```

The `exit()` function terminates the current process with the exit code as specified in `status`. This function does **not** return.

Examples for typically used exit codes: `EXIT_SUCCESS` (0), `EXIT_FAILURE`.

Parameters

<code>status</code>	exit code
---------------------	-----------

2.7.2.2 strtok()

```
char* strtok (
    char * str,
    const char * delim )
```

The `strtok()` (**string tokenize**) function tokenizes the string pointed to by `str`. For the first call the pointer to the string to be tokenized must be provided. For all subsequent tokens for the same string `NULL` must be used for `str`. Be aware, that `strtok()` manipulates the parsed string (e.g., inserts `\0`). Each call of `strtok()` returns a pointer to the next token or `NULL` if no more tokens are available.

The `strtok()` function is especially useful to prepare the arguments for a `execv()` or `execvp()` system call.

Example:

```
const char *delim = "|-";
strcpy(buffer, "This|is-an|example!");
printf("%s\n", buffer);

char *tok = strtok(buffer, delim);
while (tok != NULL) {
    printf("%s ", tok);
    tok = strtok(NULL, delim);
}
printf("\n");

// produces
$> This|is-an|example!
$> This is an example!
```

Parameters

<code>str</code>	string to be tokenized
<code>delim</code>	delimiter

Return values

<code>NULL</code>	no more token
<code>!=NULL</code> <code>ULL</code>	pointer to next token

2.7.2.3 wait()

```
pid_t wait (
    int * wstatus )
```

The `wait()` function blocks, until at least one child process has been terminated. The call of `wait(&status)` is equivalent to `waitpid(-1, &status, 0)`. For more information see `waitpid()`.

Parameters

<code>wstatus</code>	pointer to an integer value, where <code>wait()</code> will store further information
----------------------	---

Return values

<code>pid</code>	of the terminated child process
<code>-1</code>	on error, <code>errno</code> is set

2.7.2.4 waitpid()

```
pid_t waitpid (
    pid_t pid,
    int * wstatus,
    int options )
```

The `waitpid()` function blocks until the child process defined by `pid` terminates. The value of `pid` can be one of the following values:

Value	Description
<code>< -1</code>	wait for any child process where the process group ID equals the absolute value of <code>pid</code>
<code>-1</code>	wait for any child process
<code>0</code>	wait for any child process where the process group ID equals the ID of the calling process
<code>> 0</code>	wait for the child process with the defined <code>pid</code>

The value in `options` can be used to further manipulate the behavior of `waitpid()`.

Value	Description
<code>WNOHANG</code>	do not block if no child has been terminated (immediately return)
<code>WUNTRACED</code>	also return if a child has been stopped
<code>WCONTINUED</code>	also return if a child has been resumed (<code>SIGCONT</code>)

If `wstatus` is not `NULL`, `waitpid()` stores further information about the termination of the child process in the underlying `int`. Be aware that the caller must provide the memory for the `int` and only hands over a pointer! Some macros can be used to extract these information.

Macro	Description
<code>WIFEXITED(wstatus)</code>	True if child terminated by calling <code>exit()</code>
<code>WEXITSTATUS(wstatus)</code>	Returns the exit code if <code>WIFEXITED</code> is true
<code>WIFSIGNALED(wstatus)</code>	True if child terminated because of a signal
<code>WTERMSIG(wstatus)</code>	Returns the signal number if <code>WIFSIGNALED</code> is true

Example:

```
int status;
if (waitpid(child, &status, WNOHANG) < 0) {
    perror("waitpid");
    exit(EXIT_FAILURE);
}
```

Parameters

<code>pid</code>	defines child process to wait for
<code>wstatus</code>	pointer to an integer value, where <code>waitpid()</code> will store further information
<code>options</code>	further options

Return values

0	on success
-1	on error, <code>errno</code> is set

2.7.2.5 fork()

```
pid_t fork (
    void )
```

`fork()` creates a new process by duplicating the calling process. Duplicating means it executes the same program and has the same state (variable values, opened files, ...). The child and parent process can be distinguished by the return value of `fork()`. For detailed information about differences between the child and the parent see the manpage of `fork()` (`man 2 fork`).

If a child process terminates it must be collected by using `wait()` or `waitpid()`, otherwise it remains in a *zombie* state and consumes system resources.

```
pid_t pid = fork();
if (pid == 0) printf("child\n");
```



```
if (pid < 0) printf("error\n");
if (pid > 0) printf("parent\n");
```

Return values

0	child process
>0	child's pid
<0	on error, <code>errno</code> is set

2.7.2.6 execl()

```
int execl (
    const char * path,
    const char * arg0,
    ...,
    NULL )
```

The `execl()` function (**exec** arg list) replaces the currently executed program with the program as specified in `path`. It hands over all parameters after the `path` parameter (i.e., `arg0`, `arg1`, ...) as arguments for the newly executed program. By convention the first argument is the name of the program itself and the last parameter **must** be a `NULL` pointer. Because all arguments are handed over in a list, the number of arguments is fixed at compile time. This is the most important difference to the `execv()` and `execvp()` function.

Any `exec*()` function only returns, if an error occurs. The `errno` is set appropriately, so call `perror("exec")`; after the call to a `exec*()` function.

Example to execute the program `ls -lA`:

```
execl("/bin/ls", "/bin/ls", "-lA", NULL);
perror("exec");
```

Parameters

<code>path</code>	path to the executable
<code>arg0</code>	first argument (by convention: executable file name)
...	all further arguments (terminated by a <code>NULL</code> pointer)

Returns

-1 on error, `errno` is set

2.7.2.7 execv()

```
int execv (
```



```
const char * path,
char *const argv[] )
```

The `execv()` function (**exec** arg **vector**) replaces the currently executed program with the program as specified in `path`. It hands over the `argv` parameter as arguments to the newly executed program. By convention the first argument (`argv[0]`) is the name of the program itself and the last parameter **must** be a NULL pointer. Because of the usage of an array for the arguments, the number of arguments is not fixed at compile time, but can be determined at run time. This is the most important difference to the `execl()` and `execlp()` function.

Any `exec*()` function only returns, if an error occurs. The `errno` is set appropriately, so call `perror("exec")`; after the call to a `exec*()` function.

Example to execute the program `ls -lA`:

```
char *args[3];
args[0] = "/bin/ls";
args[1] = "-lA";
args[2] = NULL;
```

```
execv(args[0], args);
perror("exec");
```

Parameters

<i>path</i>	path to the executable
<i>argv</i>	array of arguments (terminated by NULL pointer)

Returns

-1 on error, `errno` is set

2.7.2.8 execlp()

```
int execlp (
    const char * file,
    const char * arg0,
    ...,
    NULL )
```

Same as `execl()`, but also searches the `PATH` environment variable if `file` does not contain a slash `/`. This means regular shell commands like `ls` are available.

See `execl()` for further information.

Returns

-1 on error, `errno` is set

2.7.2.9 execvp()

```
int execvp (
    const char * file,
    char *const argv[] )
```

Same as `execv()`, but also searches the `PATH` environment variable if `file` does not contain a slash `/`. This means regular shell commands like `ls` are available.

See `execv()` for further information.

Returns

-1 on error, `errno` is set

2.8 Strings

Files

- file [string.h](#)

Functions

- `size_t strlen` (const char *s)
 - Calculate the length of a string.*
- `char * strcpy` (char *dest, const char *src)
 - Copy a string.*
- `char * strcat` (char *dest, const char *src)
 - Append a string to another string.*

2.8.1 Detailed Description

The functions in [string.h](#) allow for an easy manipulation of C strings. Always remember to allocate the memory for the trailing `\0` after a C string and be aware, that `strlen()` does **not** include the terminating `\0` in the string length.

```
// allocate some memory
char string_a[5+1], string_b[7+1];
char string[5+7+1];
```

```
// copy substrings into memory
strcpy(string_a, "Hello");
strcpy(string_b, " World!");
```

```
// concatenate the two strings
strcpy(string, string_a);
strcat(string, string_b);
printf("%s", string); // $> Hello World!
```

```
// determine the length of the concatenated string
size_t siz = strlen(string); // siz = 12 (5+7)
```

2.8.2 Function Documentation

2.8.2.1 strlen()

```
size_t strlen (
    const char * s )
```

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

Parameters

<code>s</code>	string under test
----------------	-------------------

Returns

number of chars in `s`

2.8.2.2 strcpy()

```
char* strcpy (
    char * dest,
    const char * src )
```

The `strcpy()` (**string copy**) function copies the string pointed to by `src`, including the terminating `\0`, to the buffer pointed to by `dest`.

The strings may not overlap, and the destination string `dest` must be large enough to receive the copy.

Parameters

<code>dest</code>	buffer, where to copy to
<code>src</code>	buffer to be copied

Returns

pointer to `dest`

2.8.2.3 strcat()

```
char* strcat (
    char * dest,
    const char * src )
```

The `strcat()` (**string concatenate**) function appends the string pointed to by `src` to the string pointed to by `dest`. Thereby, the terminating `\0` of `dest` is overwritten. The concatenated string in `dest` is terminated by a `\0` again.

If `dest` is not large enough to include both strings, the program behavior is unpredictable.

Parameters

<code>dest</code>	buffer, where to append to
<code>src</code>	buffer to be appended

Returns

pointer to `dest`