A rundown of POSIX syscalls

and other useful system programming functions
AY 2020-2021, Operating Systems, Università di Verona

Luca Bramè

Contents

1 Filesystem **3**
1.1 open . . . 3
1.2 read . . . 4
1.3 write . . . 4
1.4 lseek . . . 4
1.5 close . . . 4
1.6 unlink . . . 5
1.7 stat, lstat, fstat . . . 5
1.8 access . . . 6
1.9 chmod and fchmod . . . 6
1.10 mkdir . . . 7
1.11 rmdir . . . 7
1.12 opendir . . . 7
1.13 closedir . . . 7
1.14 readdir . . . 7

2 Processes **8**
2.1 getpid . . . 8
2.2 getuid and geteuid . . . 8
2.3 getgid and getegid . . . 8
2.4 getenv . . . 8
2.5 setenv . . . 9
2.6 unsetenv . . . 9
2.7 getcwd . . . 9
2.8 chdir . . . 9
2.9 fchdir . . . 9
2.10 dup . . . 9
2.11 _exit . . . 10
2.12 atexit . . . 10
2.13 fork . . . 10
2.14 getppid . . . 10
2.15 wait . . . 11

2.16 waitpid . . . 11
2.17 exec . . . 12

3 System V IPCs **12**
3.1 IPC operations in short . . . 12
3.2 Keys . . . 13
3.2.1 ftok . . . 13
3.3 Data structures . . . 13
3.3.1 ipc_perm . . . 13
3.4 Semaphores . . . 13
3.4.1 semget . . . 14
3.4.2 semctl . . . 14
3.4.3 semop . . . 15
3.5 Shared memory . . . 16
3.5.1 shmget . . . 16
3.5.2 shmat . . . 16
3.5.3 shmdt . . . 16
3.5.4 shmctl . . . 17
3.6 Message queue . . . 17
3.6.1 msgget . . . 17
3.6.2 Message structure . . . 18
3.6.3 msgsnd . . . 18
3.6.4 msgrcv . . . 18
3.6.5 msgctl . . . 19

4 Signals **19**
4.1 Signal handler . . . 20
4.2 signal . . . 20
4.3 pause and sleep . . . 21
4.4 kill . . . 21
4.5 alarm . . . 22
4.6 sigset_t, sigemptyset, sigfillset . . . 22
4.6.1 siagaddset and sigdelset . . . 22
4.6.2 sigismember . . . 22
4.7 sigprocmask . . . 23

5 PIPEs **23**
5.1 pipe . . . 24

6 FIFOs **24**
6.1 mkfifo . . . 24
6.2 Reading from a FIFO . . . 24

7 Contact me **25**

1 Filesystem

1.1 open

```c
int open(const char *pathname, int flags, .../*mode_t mode */);
```

Either opens, or creates and then opens, a file.

On success, it returns the *file descriptor* of the file. On failure, it returns -1.

`flags` is a bit mask for one of more of the following constants (that should be XORED) and it specifies the **access mode** for the file:

Flag	Description
`O_RDONLY`	Opens for reading only
`O_WRONLY`	Opens for writing only
`O_RDWR`	Opens for reading and writing
`O_TRUNC`	Truncate existing file to zero length
`O_APPEND`	Writes are always appended to the end of the file
`O_CREAT`	Create file if it doesn't already exist
`O_EXCL`	With `O_CREAT`, ensure this call creates the file

Memo: `O_EXCL` stands for "Exclusive use flag".

`mode` is a bit mask for one or more of the following constants and it specifies **permissions** for the new file:

Flag	Description
`S_IRWXU`	user has read, write, and execute permission
`S_IRUSR`	user has read permission
`S_IWUSR`	user has write permission
`S_IXUSR`	user has execute permission
`S_IRWXG`	group has read, write, and execute permission
`S_IRGRP`	group has read permission
`S_IWGRP`	group has write permission
`S_IXGRP`	group has execute permission
`S_IRWXO`	others have read, write, and execute permission
`S_IROTH`	others have read permission
`S_IWOTH`	others have write permission
`S_IXOTH`	others have execute permission

Useful things to memorize:

- `R` = Read permission
- `W` = Write permission
- `X` = Execute permission
- Leading `U` = User
- `USR` = User

- Leading `G` = Group
- `GRP` = Group
- Leading `O` = Other
- `OTH` = Other

1.2 read

```c
ssize_t read(int fd, void *buf, size_t count);
```

Reads data from a file descriptor.
Returns number of bytes read on success, -1 on failure.
`count` = max number of bytes to read from file descriptor `fd`.
`buf` = Address of memory buffer into which read input data is stored.

1.3 write

```c
ssize_t write(int fd, void *buf, size_t count);
```

Write data to a file descriptor.
Returns number of bytes written on success, -1 on failure.
`count` = Number of bytes of buffer pointed by `buf` that has to be written to file desciptor referred by `fd`

1.4 lseek

```c
off_t lseek(int fd, off_t offset, int whence);
```

Adjusts offset location of an open file.
Returns offset location, or -1 on failure.
`fd` = File desciptor of open file
`offset` = Value in bytes
`whence` = Base point from which `offset` is interpreted

```c
// first byte of the file.
off_t current = lseek(fd1, 0, SEEK_SET);
// last byte of the file.
off_t current = lseek(fd2, -1, SEEK_END);
// 10th byte past the current offset location of the file.
off_t current = lseek(fd3, -10, SEEK_CUR);
// 10th byte after the current offset location of the file.
off_t current = lseek(fd4, 10, SEEK_CUR);
```

1.5 close

```c
int close(int fd);
```

Closes an open file descriptor.

1.6 unlink

```
int unlink(const char *pathname);
```

Removes a link to a file. It also removes the file itself if that is the last link to a file. (There can be more than one link to a file because of hard links: every hard link pointing to that file needs to be unlinked from the filesystem for that file to be removed.)

1.7 stat, lstat, fstat

```
int stat(const char *pathname, struct stat *statbuf);
int lstat(const char *pathname, struct stat *statbuf);
int fstat(int fd, struct stat *statbuf);
```

Retrieve information about a file.

- `stat` returns information about a named file
- `lstat` returns information about a symbolic link
- `fstat` like `stat`, but uses fd instead of pathname

These syscalls return a stat structure in the buffer pointed to by `statbuf`:

```
struct stat {
    dev_t st_dev; // IDs of device on which file resides.
    ino_t st_ino; // I-node number of file.
    mode_t st_mode; // File type and permissions.
    nlink_t st_nlink; // Number of (hard) links to file.
    uid_t st_uid; // User ID of file owner.
    gid_t st_gid; // Group ID of file owner.
    dev_t st_rdev; // IDs for device special files.
    off_t st_size; // Total file size (bytes).
    blksize_t st_blksize; // Optimal block size for I/O (bytes).
    blkcnt_t st_blocks; // Number of (512B) blocks allocated.
    time_t st_atime; // Time of last file access.
    time_t st_mtime; // Time of last file modification.
    time_t st_ctime; // Time of last status change.
};
```

`st_mode` is a bit mask that identifies the file type and specifies the file permissions. The file type can be extracted by applying a logical AND with the constant `S_IFMT` and comparing the result with one of the following constants. Alternatively, the corresponding "Test macro" can be ran on the `statbuf.st_mode` entry to get an equivalent result.

Constant	Test macro	File type
S_ISREG	S_ISREG	Regular file
S_IFDIR	S_ISDIR()	Directory
S_IFCHR	S_ISCHR()	Character device
S_ISBLK	S_IFBLK()	Block device
S_IFIFO	S_ISIFO()	FIFO or pipe
S_IFSOCK	S_ISSOCK()	Socket
S_IFLNK	S_ISLNK()	Symbolic link

1.8 access

```
int access(const char *pathname, int mode):
```

Checks accessibility of a file based on a process's real user ID and group ID
Returns 0 on success, -1 on failure
If `pathname` is a symlink, `access` dereferences it.
`mode` is a bit mask that may contain the following constants:

Constant	Description
F_OK	Does the file exist?
R_OK	Can the file be read?
W_OK	Can the file be written?
X_OK	Can the file be executed?

1.9 chmod and fchmod

```
#include <sys/stat.h>

// Example of mode with owner-write on and others-read off
mode_t mode = (sb.st_mode | S_IWUSR) & ~S_IROTH;

int chmod(const char *pathname, mode_t mode);
```

```
#define _BSD_SOURCE
#include <sys/stat.h>

// Example of mode with owner-write on and others-read off
mode_t mode = (sb.st_mode | S_IWUSR) & ~S_IROTH;

int fchmod(int fd, mode_t mode);
```

Change the permission of a file.
Both return 0 on success, -1 on failure.
`chmod` changes the permissions of file referenced by `pathname`, while `fchmod` accepts the file descriptor of an open file.
`mode` accepts the same constants as `open()`.

1.10 mkdir

```c
int mkdir(const char *pathname, mode_t mode);
```

Creates a new directory. If this pathname already exists, it fails with error `EEXIST`.

1.11 rmdir

```c
int rmdir(const char *pathname);
```

Removes a directory. The target directory must be empty, if it isn't, `rmdir` fails. If the path links to a symbolic link, it fails with `ENOTDIR`.

1.12 opendir

```c
DIR *opendir(const char *dirpath);
```

Opens a directory. If it succeds, it returns the directory stream handler. It returns `NULL` on failure.

After `opendir` returns, the directory stream `DIR *` is positioned at the first entry in the directory list.

1.13 closedir

```c
int closedir(DIR *dirp);
```

Closes the open directory stream referred to by `dirp` and it frees the resources used by the stream. It returns 0 on success and -1 on failure.

1.14 readdir

```c
struct dirent *readdir(DIR *dirp);
```

Reads the content of a directory.

It returns a pointer to an allocated structure describing the next directory entry, or NULL on end-of-directory error.

Every time it's called, it reads the next entry from the directory stream referred to by `dirp`. Each entry is a struct defined as follows:

```c
struct dirent {
    ino_t d_ino;    // File i-node number.
    unsigned char d_type;   // Type of file.
    char d_name[256];   // Null-terminated name of file.
    //...
```

The value returned in `d_type` can be one of the following macros:

Constant	File type
`DT_BLK`	block device
`DT_CHR`	character device
`DT_DIR`	directory
`DT_FIFO`	named pipe (FIFO)
`DT_LNK`	symbolic link
`DT_REG`	regular file
`DT_SOCK`	UNIX socket

2 Processes

2.1 getpid

```c
pid_t getpid(void);
```

Returns PID of calling process. It's always successful.
`pid_t` is an integer type designated to store process id's.

2.2 getuid and geteuid

```c
uid_t getuid(void); // Real user ID
uid_t geteuid(void); // Effective user ID
```

- `getuid` returns the *real* user ID of the calling process
- `geteuid` returns the *effective* user ID of the calling process

They are always successful.

2.3 getgid and getegid

```c
uid_t getgid(void); // Real group ID
uid_t getegid(void); // Effective group ID
```

- `getgid` returns the *real* group ID of the calling process
- `getegid` returns the *effective* group ID of the calling process

They are always successful.

2.4 getenv

```c
char *getenv(const char *name);
```

Given a variable name, returns a pointer to string of the associated value in the process's environment. It returns `NULL` if there is no environment variable with that specified name.

2.5 setenv

```
int setenv(const char *name, const char *value, int overwrite);
```

Adds `NAME=VALUE` to the environment, unless that same pair already exists and `overwrite` is 0. If a pair with the same key already exists, the pair gets overwritten. If `overwrite` is nonzero, the environment gets changed no matter what.

2.6 unsetenv

```
int unsetenv(const char *name);
```

Removes the variable identified by the string `name` from the environment.

2.7 getcwd

```
char *getcwd(char *cwdbuf, size_t size);
```

Used to retrieve a process's current working directory.

On success, it returns a pointer to `cwdbuf`. If the pathname for che cwd exceeds `size` bytes, it returns 0.

2.8 chdir

```
int chdir(const char *pathname);
```

Changes the process's current working directory to the relative or absolute pathname in `pathname`.

Returns 0 on success, -1 on error.

2.9 fchdir

```
int fchdir(int fd);
```

Same as `chdir`, but the directory is specified via a file descriptor.

Returns 0 on success, -1 on error.

2.10 dup

```
int dup(int oldfd);
```

Takes an existing file descriptor and returns a new fd that refers to the same open fd. The new fd will be the lowest unused fd.

Returns -1 on error.

2.11 `_exit`

```
void _exit(int status);
```

Terminates calling process successfully. Cannot fail.
First bit of `status` defines *termination* status of the process.

- 0 → Process terminated **successfully**
- nonzero → Process terminated **unsuccessfully**

It should be noted that using library function `exit()` is more common than calling syscall `_exit()`, since the former calls exit handlers and flushes `stdio` stream buffers before calling `_exit()`.

Additionally, calling `return STATUS` works the same way, and falling off the end of `main()` is equivalent to `return 0` in C99, but it's undefined behaviour otherwise.

2.12 `atexit`

```
int atexit(void (*func)(void));
```

Defines exit handlers for current process.
`atexit` adds the function pointer `func` to a list of functions that are called during process termination, **as long as the process is terminated gracefully**.
Multiple exit handlers are called in *reverse order* of registration.

2.13 `fork`

```
pid_t fork(void);
```

Creates a new process.
Child process is an almost exact replica of calling (parent) process. Child receives parent's fds and attached shmem segments.
In parent: returns PID of child process on success or -1 on error.
In child: always returns 0.

2.14 `getppid`

```
pid_t getppid(void);
```

Always successfully returns PID of parent process.

2.15 wait

```
pid_t wait(int *status);
```

Waits for one of the children of the calling process to terminate.

- If calling process have no unwaited-for children, `wait` returns -1 and sets errno to `ECHILD` (No child processes).
- If no child has yet terminated, `wait` blocks the calling process until a child terminates. If a child has already terminated, `wait` returns.
- If `status` is not `NULL`, information about child is stored in the int pointed by `status`.

2.16 waitpid

```
pid_t waitpid(pid_t pid, int *status, int options);
```

Blocks the calling process until child specified by `pid` has changed state.
`status` is same as `wait`.
Value of `pid` depends on what child process we want to wait for:

- `pid` ≥ 0, wait for the child whose PID equals to `pid`
- `pid` $= 0$, wait for any child in the same calling process's group
- `pid` $<$ -1, wait for any child in the process group `|pid|`
- `pid` $=$ -1, wait for any child

`options` is an OR of:

- `WUNTRACED`, Return when child stopped by signal or terminates
- `WCONTINUED`, Return when a `SIGCONT` signal has resumed execution of a child
- `WNOHANG`, Non-blocking: return immediately if no child has changed state, do not wait. Returns 0.
- `0`, Wait only for terminated children.

To gather more information about the termination status of a process, various macros can be ran on `status`.

Macro	Description
`WIFEXITED`	Returns true if child exited normally
`WEXITSTATUS`	Returns exit status of child
`WIFSIGNALED`	Returns true if the child was killed by a signal
`WTERMSIG`	Returns number of signal that caused the process to terminate
`WIFSTOPPED`	Returns true if child process was stopped
`WSTOPSIG`	Returns number of signal that stopped the process
`WIFCONTINUED`	Returns true if child was resumed by `SIGCONT`

2.17 exec

```c
#include <unistd.h>
// None of the following returns on success, all return -1 on error.
int execl (const char *path, const char *arg, ... ); // ... variadic functions
int execlp(const char *path, const char *arg, ... );
int execle(const char *path, const char *arg, ... , char *const envp[]);
int execv (const char *path, char *const argv[]);
int execvp(const char *path, char *const argv[]);
int execve(const char *path, char *const argv[], char *const envp[]);
```

Replaces the current process image with a new process image.
Here's how to distinguish what various `exec` family functions do:

- Takes `pathname` as path by default. If it contains a **p**, it accepts the **filename** instead.
- If it contains a **l** (list), `argv[]` is a **list**.
- If it contains a **v** (vector), `argv[]` is an **array**.
- Inherits the caller's environment by default. If it contains a **e** (environment), it expects the new environment to be defined in an **array**.

NB: Both `argv` and `envp` are NULL-terminated! These lists and arrays must always be terminated with `(char *)NULL`.

The first item of `argv` is the new program.

These functions only return anything on failure. They do not return anything on success.

3 System V IPCs

There are two implementations of Inter-Process Communication: System V and POSIX. We shall focus on System V's implementation.

3.1 IPC operations in short

Interface	**msq**	**sem**	**shmem**
Header file	`<sys/msg.h>`	`<sys/sem.h>`	`<sys/shm.h>`
Data structure	`msqid_ds`	`semid_ds`	`shimd_ds`
Create/Open	`msgget(...)`	`semget(...)`	`shmget(...)`
Close	(none)	(none)	`shmdt(...)`
Control Ops.	`msgctl(...)`	`semctl(...)`	`shmctl(...)`
Performing IPC	`msgsnd(...)` `msgrcv(...)`	`semop(...)` To test/adjust	access memory in shared region

- **IPC Key**: Integer value, analogous to a filename, used to uniquely identify an instance of an IPC object on a running system. Used in `get` system calls.
- **IPC Identifier**: Integer value, analogous to a file descriptor. Used in all subsequent system calls.
- **`get` system calls**: Translate an IPC Key into an IPC Identifier. It creates a new IPC object if no IPC object with the provided key is present, while it simply returns the relevant identifier if an IPC object with such key is found.

3.2 Keys

3.2.1 `ftok`

```
key_t ftok(char *pathname, int proj_id);
```

Converts a `pathname` and a project ID into a System V IPC key. Only the last 8 bits of `proj_id` are actually used.

3.3 Data structures

3.3.1 `ipc_perm`

The kernel maintains specific data structures for each type of IPCs. Every IPC has one instance of its kind of data structure associated to it. This data structure contains specific information about that particular IPC object. Each of these different data structures, however, contains the common data structure `ipc_perm`, whose role is to hold relevant permissions:

```
struct ipc_perm {
    key_t __key; /* Key, as supplied to 'get' call */
    uid_t uid; /* Owner's user ID */
    gid_t gid; /* Owner's group ID */
    uid_t cuid; /* Creator's user ID */
    gid_t cgid; /* Creator's group ID */
    unsigned short mode; /* Permissions */
    unsigned short __seq; /* Sequence number */
};
```

3.4 Semaphores

Semaphores are a construct to implement mutual exclusion in order to let processes synchronize their actions to ensure a critical section only ever gets accessed by one process at a time. Some resources must never be accessed or modified concurrently, as that would cause corruption: this is the reason why semaphores (or other forms of mutual exclusion) often need to be included in programs that rely on multiple concurrent processes.

Semaphores are kernel-maintained values that get modified by OS-provided primitives before performing critical actions.

3.4.1 `semget`

```
int semget(key_t key, int nsems, int semflg);
```

Creates a new semaphore set or obtains the identifier of an existing one.

Returns semaphore set identifier on success, -1 on failure.

`key`: IPC key,

`nsems`: Number of semaphores in set

`semflg`: Bit mask that specifies the permissions (see `open()`). The following constants can also be ORed in:

- `IPC_CREAT`: If no semaphore set with `key` exists, create one.
- `IPC_EXCL`: Fail if `IPC_CREAT` is being used and a semaphore set with `key` already exists.

3.4.2 `semctl`

```
int semctl(int semid, int semnum, int cmd, ... /* union semun arg */);
```

Performs operations on a whole semaphore set or on a single semaphore contained in it.

`semid`: ID of semaphore set to run command on.

`semnum`: Number of semaphore in set to run command on. `0` for whole set.

`cmd`: Control operation to run.

`union semun arg`: Required by some operations. Relies on the union `semun`.

```
union semun {
    int val;
    struct semid_ds * buf;
    unsigned short * array;
};
```

`cmd` can accept one of these constants:

- `IPC_RMID`: Removes a semaphore set. Awakens blocked processes and sets `errno` to `EIDRM` (Identifier removed).
- `IPC_STAT`: Copy `semid_ds` to `arg.buf`
- `IPC_SET`: Update fields in `semid_ds` using values from buffer pointed by `arg.buf`
- `SETVAL`: Value of `semnum`-th semaphore in set initialized to `arg.val`

- GETVAL: Make semctl return value of the semnum-th semaphore in the set
- SETALL: Initialize all semaphores in the set to values in arg.array
- GETALL: Copy values of all semaphores in set to arg.array
- GETPID: Return PID of last process to perform a semop on semnum-th semaphore
- GETNCNT: Return number of processes waiting for value of semnum-th semaphore to increase
- GETZCNT: Return number of processes waiting for value of semnum-th semaphore to become 0.

3.4.3 semop

```
int semop(int semid, struct sembuf *sops, unsigned int nsops);
```

Performs one or more operations (wait (P) and signal (V)) on semaphores.
Returns 0 on success, -1 on failure.
sops: Pointer to array that contains sorted sequence of operations to be performed atomically
nsops: Size of array pointed to by sops
The sops array contains structures that are structured like the following:

```
struct sembuf {
    unsigned short sem_num; /* Semaphore number */
    short sem_op; /* Operation to be performed */
    short sem_flg; /* Operation flags */
};
```

sem_op refers to the operation to be performed and can be:

- sem_op > 0: value(semnum) += value(sem_op)
- sem_op $= 0$: if value(semnum) != 0; calling process blocked until semaphore is 0
- sem_op < 0: value(semnum) -= value(sem_op)
  - Blocks calling process until sem value has increased to the point operation can be performed without resulting in negative value

semop can be made non-blocking if sops[i] is set to IPC_NOWAIT. In this case, when semop would have blocked, it fails with error EAGAIN (Resource temporarily unavailable [try again later]) instead.

3.5 Shared memory

A shared memory segment is a kernel-managed memory segment that can be used to exchange (larger amounts of) data between attached processes. Kernel intervention is not required to use a shmem segment once attached, since it becomes part of the process's virtual address space.

Shared memory syscalls implement no mutual exclusion of their own, so it is advisable to use semaphores or equivalent solutions to protect the shared memory from concurrent accesses (which can cause data corruption).

3.5.1 `shmget`

```c
int shmget(key_t key, size_t size, int shmflg);
```

Creates a new shared memory segment or obtains the identifier of an existing one. If a new segment is created, it gets initialized to 0.

Returns a shared memory identifier on success, -1 on failure.

`key`: IPC key

`size`: Desired segment size in bytes. If we just want to get the identifier of an existing shared memory segment, this field must be */leq* size of the segment.

`shmflg`: Bit mask specifying the permissions (Like `mode` in `open()`). The following flags can also be ORed in:

- `IPC_CREAT`: If no segment with `key` exists, create one.
- `IPC_EXCL`: Fail if `IPC_CREAT` is being used and a shared memory segment with `key` already exists.

3.5.2 `shmat`

```c
void *shmat(int shmid, const void *shmaddr, int shmflg);
```

Attaches shared memory segment to calling process's virtual address space.

If `shmaddr` is `NULL`, the *kernel* selects the suitable address to attach the segment at. `shmflg` also gets ignored.

If `shmaddr` is **not** `NULL`, the segment is attached at `shmaddr` address. On top of this, is `shmflg` is `SHM_RND`, `shmaddr` is rounded down to the nearest multiple of the constant `SHMLBA` (shared memory low boundary address).

If `shmflg` is `SHM_RDONLY`, the shared memory is attached in **read-only** mode.

If `shmflg` has value 0, the shared memory is attached in **read-write** mode.

3.5.3 `shmdt`

```c
int shmdt(const void *shmaddr);
```

Detaches a shared memory segment from the calling process's virtual address space.

Returns 0 on success, -1 on error.

All shared memory segments are automatically detached during an `exec` and on process termination.

3.5.4 shmctl

```c
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

Performs control operations on a shared memory segment.
Returns 0 on success, -1 on failure.
`cmd` specifies the operation to perform on shared memory segment:

- `IPC_RMID`: Mark selected segment for deletion. The segment will be deleted as soon as all processes using it have detached from it.
- `IPC_STAT`: Copy the `shmid_ds` data structure associated with this shared memory segment to the buffer pointed by `buf`.
- `IPC_SET`: Update selected fields of the `shmid_ds` data structure associated with this shared memory segment using values provided in the buffer pointed to by `buf`.

The kernel associates every shared memory segment with a `shmid_ds` data structure as follows:

```c
struct shmid_ds {
    struct ipc_perm shm_perm; /* Ownership and permissions */
    size_t shm_segsz; /* Size of segment in bytes */
    time_t shm_atime; /* Time of last shmat() */
    time_t shm_dtime; /* Time of last shmdt() */
    time_t shm_ctime; /* Time of last change */
    pid_t shm_cpid; /* PID of creator */
    pid_t shm_lpid; /* PID of last shmat() / shmdt() */
    shmatt_t shm_nattch; // Number of currently attached processes
};
```

3.6 Message queue

3.6.1 msgget

```c
int msgget(key_t key, int msgflg);
```

Creates a new message queue, or obtains the identifier of an existing queue.
Returns message queue identifier on success, -1 on failure.
`msgflg` is a bit mask containing the associated permissions that also accepts:

- `IPC_CREAT`: If no message queue with `key` exists, create one.
- `IPC_EXCL`: Fail if `IPC_CREAT` is being used and a message queue with `key` already exists.

3.6.2 Message structure

A message in a message queue always follows this structure:

```
struct mymsg {
    long mtype; /* Message type */
    char mtext[]; /* Message body */
};
```

`mtype` > 0.

`mtext[]` is an arbitrary structure that can be whatever type the programmer decides, or omitted.

3.6.3 `msgsnd`

```
int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);
```

Writes a message to the message queue.
Returns 0 on success, -1 on failure.
`msqid`: IPC id of message queue
`msgp`: Address pointing to the message structure
`msgsz`: Size of the message expressed by number of bytes contained in the `mtext[]` field of the message.
`msgflg` can be

- 0 (make operating blocking) → Block the calling process until enough space has become available to place the message on the queue
- `IPC_NOWAIT` (Return immediately if no message of the requested type is in the queue. The system call fails with `errno` set to `ENOMSG` (No message of the desired type).). In the latter case, the call fails with `EAGAIN` (Resource temporarily unavailable).

3.6.4 `msgrcv`

```
ssize_t msgrcv(int msqid, void *msgp, size_t msgsz, long msgtype, int msgflg);
```

Reads **and removes** a message from the message queue.
Returns number of bytes copied into `msgp` on success or -1 on failure.
In this case, `msgzg` expresses the maximum space available in the `mtext` field of the `msgp` buffer.

- `msgtype` $== 0$ → The first message from the queue is removed and returned to calling process.
- `msgtype` > 0 → The first message of the lowest `mtype` $==$ modulo `msgtype` is removed and returned to the calling process.
- `msgtype` < 0 → The first message of the lowest `mtype` $\leq$ modulo `msgtype` is removed and returned to the calling process.

msgflg is a bit mask for zero or more of these flags:

- IPC_NOWAIT: Fails with ENOMSG if no message matching msgtype is in the queue rather than blocking the process and waiting for it.
- MSG_NOERROR: By default, msgrcv fails if the size of mtext exceeds the available space. When this flag is specified, instead, the message is removed from the queue, its mtext field is truncated to msgsz.

3.6.5 msgctl

```
int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

Performs control operations on system call.
Returns 0 on success, -1 on error.
cmd specifies operation to be performed on the queue:

- IPC_RMID: Remove message queue. All blocked reader/writer processes are awakened and errno is set to EIDRM (Identifier removed).
- IPC_STAT: Copy msqid_ds associated to queue to buf
- IPC_SET: Update selected fields of msqid_ds using values from buf.

The kernel associates each message queue with a corresponding msqid_ds data structure:

```
struct msqid_ds {
    struct ipc_perm msg_perm; /* Ownership and permissions */
    time_t msg_stime; /* Time of last msgsnd() */
    time_t msg_rtime; /* Time of last msgrcv() */
    time_t msg_ctime; /* Time of last change */
    unsigned long __msg_cbytes; /* Number of bytes in queue */
    msgqnum_t msg_qnum; /* Number of messages in queue */
    msglen_t msg_qbytes; /* Maximum bytes in queue */
    pid_t msg_lspid; /* PID of last msgsnd() */
    pid_t msg_lrpid; /* PID of last msgrcv() */
};
```

4 Signals

A signal is a notification to a process that an event has occurred. They break normal execution flow.

A signal is *generated* by some event and then *delivered* to some process. Until a generated signal has not been delivered yet, it's said to be *pending*.

Signals can terminate, stop, resume a process or have custom actions. Additionally, signals that can be caught should be handled by signal handlers, functions that run as soon as their assigned signal is caught.

- Signals to **terminate** a process:
  - `SIGTERM` (15) - Can be caught, handler expected. Gracefully terminates a process.
  - `SIGINT` (2) - Can be caught. Interrupts a process when the user sends the Contr-C character.
  - `SIGQUIT` (3) - Can be caught. Always terminates a process producing a core dump.
  - `SIGKILL` (9) - **Cannot be caught**. Terminates a process abruptly, without the help of a signal handler. **Use should be avoided whenever possible**: the way `SIGKILL` terminates processes is not clean, and problems may or may not arise depending on how the process is programmed. It is, however, useful to terminate non-responding processes.
- Signals to **stop and resume** a process:
  - `SIGSTOP` (17) - **Cannot be caught**. Always stops a process.
  - `SIGCONT` (19) - Can be caught. Resumes a stopped process.
- Other **import** signals:
  - `SIGPIPE` (13) - Can be caught. Generated when a process tries to write a PIPE.
  - `SIGALARM` (14) - Can be caught. Terminates a process upon expiration of a real-time timer set by a call to *alarm*.
  - `SIGUSR1` (30) and `SIGUSR2` (31) - Can be caught. No predefined action, available for programmer-defined purposes.

4.1 Signal handler

A signal handler is a function that gets called when a signal associated to it is delivered to the process.

```c
void sigHandler(int sig) {
    /* Code for the handler */
}
```

`sig`: Number of signal delivered to process.

4.2 `signal`

```c
#include <signal.h>

typedef void (*sighandler_t)(int);
// Returns previous signal disposition on success, or SIG_ERR on error
sighandler_t signal(int signum, sighandler_t handler);
```

Changes the default signal handler for a signal type (number) for the calling process.

Returns the previous signal disposition on success, or the constant `SIG_ERR` on failure.

`signum`: Signal number for which we want to set a signal handler.

- `handler` can be:
  - Address of a user-defined signal
  - `SIG_DFL` → the default action associated with the signal occurs
  - `SIG_IGN` → The signal `signum` is ignored

4.3 pause and sleep

Stopping the execution flow of a program is necessary to wait for incoming signals without busy waiting.

```
int pause();
```

`pause` suspends execution of the process until the call is interrupted by a signal handler or an unhandled signal terminates the process.

It always returns -1 and sets `errno` to `EINTR` (Interrupted function call).

```
unsigned int sleep(unsigned int seconds);
```

`sleep` suspends execution of the calling process for `seconds` seconds or until a signal is caught.

It returns 0 if it completes correctly (taking the expected amount of time), or the number of seconds it should have slept but didn't if it prematurely terminated.

4.4 kill

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

Lets a process signal another process.

Despite its name, using `kill` does not necessarily *terminate* the process `pid`: it just sends that process a signal of number `sig`.

It returns 0 on success, or -1 on error.

- `pid` specifies one or more processes that should be signaled with `sig`:
  - `pid` > 0 → Process with PID == `pid` is signalled
  - `pid` $= 0$ → Every process in the same process group as the calling process, including the calling process itself, is signaled.
  - `pid` < 0 → All processes in the process group whose ID equals the *absolute* value of `pid` get signaled.
  - `pid` = -1 → Every process for which calling process has permission to send a signal gets signaled, *except* `init` and the calling process itself.

4.5 alarm

```
unsigned int alarm(unsigned int seconds);
```

Arranges for a `SIGALARM` signal to be delivered to the calling process after a fixed delay of `seconds` seconds.

After the time expires, a `SIGALARM` signal is sent to the calling process.

Setting a timer with `alarm` overrides any previously set timers.

It always succeeds. It returns the number of seconds remaining on any previously set timer, or 0 if no timer had previously been set.

4.6 sigset_t, sigemptyset, sigfillset

The `sigset_t` data type represents a signal set. `sigemptyset` and `sigfillset` must be used to initialize a signal set immediately after creation.

```
#include <signal.h>

typedef unsigned long sigset_t;

// Both return 0 on success, or -1 on error.
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
```

`sigemptyset` initializes a signal set to contain **no signal**. `sigfillset` initializes a signal set to contain **all signals**.

4.6.1 siagaddset and sigdelset

Individual signals can be added to or removed from an *initialized* signal set as follows. The names are self-explanatory.

```
#include <signal.h>

// Both return 0 on success, or -1 on error
int sigaddset(sigset_t *set, int sig);
int sigdelset(sigset_t *set, int sig);
```

4.6.2 sigismember

This function is useful to test a signal for membership of a set.

```
// Returns 1 if sig is a member of set, otherwise 0
int sigismember(const sigset_t *set, int sig);
```

4.7 sigprocmask

The kernel maintains for each process a **signal mask**, a set of *masked* signals; in other words, a set of blocked signals that will not be delivered to the process until they are removed from this mask.

```c
// Returns 0 on success, or -1 on error
int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
```

- `how` determines *how* `sigprocmask` changes the signal mask:
  - `SIG_BLOCK` → Set of blocked signals is the **union** of the current set and `set`
  - `SIG_UNBLOCK` → Signals in `set` get **removed** from the signal mask
  - `SIG_SETMASK` → Set of blocked signals is *set to* `set`

If `oldset` is not null, it points to a buffer to return the previous signal mask to.

If `set` is `NULL`, we can retrieve the signal mask copying it to `oldset` without making any changes.

5 PIPEs

A pipe is a unidirectional, sequential byte stream that resides in kernel memory and has a write end and a read end that two processes can attach to respectively.

- **Unidirectional**: Data only travels from the write end, to the read end.
- **Sequential**: Bytes are read from the PIPE in the exact order as they were written.
- **No** concept of **message** or message **boundaries**: it's a raw data stream, blocks of data of arbitrary size can be read from the read end.
- A reader attempting to read from an empty pipe will get blocked until either at least 1 byte has been written to the pipe or a no-termination signal occurs, setting `errno` to `EINTR` (Interrupted function call)
- If the write-end of a pipe is closed, the reader process will see an EOF when it's done reading the remaining bytes in the PIPE.
- A write is blocked until sufficient space is available in the pipe to perform the operation atomically (65536 bytes is the capacity on Linux), or a no-terminating signal occurs, setting `errno` to `EINTR`.
- Writes of larger data blocks than `PIPE_BUF` (4096 bytes large on Linux) may be broken into segments of arbitrary size smaller then `PIPE_BUF`.

5.1 pipe

```
int pipe(int filedes[2]);
```

Creates a new PIPE.
Returns 0 on success, -1 on failure.
If successful, it returns two open file descriptors in the array `filedes`:

- `filedes[0]` - read-end
- `filedes[1]` - write-end

PIPEs are used to allow communication between **related** processes, as children inherit the parent's file descriptor table. FIFOs, also known as "named pipes", are a better match if opening a byte stream between unrelated processes is necessary.

Reader and writer processes will have to perform I/O on the file descriptors as usual, using `read` and `write` system calls respectively.

6 FIFOs

A FIFO, much like a PIPE, is a byte stream that allows two processes to exchange bytes. FIFOs are also known as **named pipes**: in fact, the main feature that sets FIFOs aside from PIPEs is that a FIFO has a name within the filesystem and, since it *is* a file, it gets treated as one. This allows communication between unrelated processes since a child does not have to inherit the file descriptor from the parent's file descriptor table, as it can simply `open` the file.

6.1 mkfifo

```
int mkfifo(const char *pathname, mode_t mode);
```

Creates a new FIFO.
`pathname`: filesystem location where FIFO is created.
`mode`: Permissions for the FIFO, same as in `open`.

6.2 Reading from a FIFO

A FIFO is read using the `open` syscall.

As `FLAGS`, it only accepts `O_RDONLY` and `O_WRONLY`. In fact, the only sensible use for a FIFO is to have a read end and a write end, and to attach a process to each.

Opening the FIFO for reading is **blocking** until a process opens it for writing, and the other way around as well.

7 Contact me

If you have found any error in this document, please contact me at email address luca.brame@studenti.univr.it