CS 111 OS Cheat Sheet

UCLA · C System Calls & POSIX API

PROCESS MANAGEMENT

Creating Processes

```
pid_t fork(void)
```

↪ Creates an exact copy of the calling process. The child gets a duplicate of the parent's address space, file descriptors, and signal handlers. Returns 0 to the child, the child's PID to the parent, -1 on error. Memory is copy-on-write; fds share the same underlying file descriptions (same offset).

```
pid_t vfork(void)
```

↪ Like `fork` but child shares the parent's address space until `exec` or `_exit`. Parent is suspended. Dangerous; prefer `fork`.

Replacing Process Image (exec family)

```
int execve(const char *path, char *const argv[], char *const envp[])
```

↪ The fundamental exec syscall. Replaces the current process image with a new program loaded from `path`. `argv[0]` is conventionally the program name; both `argv` and `envp` must be NULL-terminated. Never returns on success; returns -1 on error.

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

↪ Like `execve` but searches `PATH` env var for `file`, and inherits the current environment (`environ`).

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

↪ Variadic form: arguments listed individually, terminated by `(char*)NULL`. Also searches `PATH`.

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

↪ Like `execlp` but uses exact `path`; does not search `PATH`.

```
int execv(const char *path, char *const argv[])
```

↪ Like `execve` but inherits the current environment. Uses exact `path`.

```
// exec mnemonic:
// l = list args     v = vector (array) args
// p = search PATH   e = explicit envp
```

Waiting for Children

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

↪ Blocks until any child terminates, stops, or is continued. Fills `*wstatus`. Returns child PID, or -1 if no children exist.

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

↪ Waits for a specific child (or group). `pid=-1` means any child (same as `wait`). Options: `WNOHANG` returns immediately if nothing ready; `WUNTRACED` also reports stopped children; `WCONTINUED` reports resumed children.

```
// Status macros (apply to wstatus):
WIFEXITED(s)     // true if exited normally
WEXITSTATUS(s)   // exit code 0-255
WIFSIGNALED(s)   // true if killed by signal
WTERMSIG(s)      // which signal killed it
WIFSTOPPED(s)    // true if stopped
WSTOPSIG(s)      // which signal stopped it
WIFCONTINUED(s)  // resumed by SIGCONT
```

Process Identity and Control

```
pid_t getpid(void)
```

↪ Returns PID of the calling process.

```
pid_t getppid(void)
```

↪ Returns PID of the parent process. If parent exited, returns 1 (init) or the subreaper.

```
int setpgid(pid_t pid, pid_t pgid)
```

↪ Sets process group of `pid` to `pgid`. Used to put a process into its own group for shell job control.

```
pid_t setsid(void)
```

↪ Creates a new session and process group; the caller becomes session leader. Used by daemons to detach from the controlling terminal.

```
int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options)
```

↪ More expressive than `waitpid`. `idtype`: `P_PID` (specific pid), `P_PGID` (process group), `P_ALL` (any child). Fills `infop` with detailed info including `si_status`, `si_code` (`CLD_EXITED`, `CLD_KILLED`, `CLD_STOPPED`, etc.), and `si_pid`. With `WNOWAIT`, leaves the child waitable.

```
pid_t gettid(void)
```

↪ Returns the thread ID (TID) of the calling thread. In a single-threaded process equals the PID; in a multi-threaded process each thread has a unique TID. Linux-specific; declared in `<unistd.h>` on glibc $\geq$ 2.30, otherwise use `syscall(SYS_gettid)`.

```
int nice(int inc)
```

↪ Adds `inc` to the process's nice value (scheduling priority). Nice values range from -20 (highest priority) to +19 (lowest). Only root can lower the nice value (increase priority). Returns new nice value, or -1 on error (check `errno` since -1 is also a valid nice value).

```
int getpriority(int which, id_t who)
```

↪ Gets the scheduling priority. `which`: `PRIO_PROCESS`, `PRIO_PGRP`, `PRIO_USER`. `who=0` means the calling process/group/user.

```
int setpriority(int which, id_t who, int prio)
```

↪ Sets scheduling priority. Only root can set to negative values (higher priority than normal).

```
int prctl(int option, unsigned long arg2, ...)
```

↪ Linux-specific per-process control. Common options: `PR_SET_NAME`/`PR_GET_NAME` (set/get thread name, visible in `ps`); `PR_SET_DUMPABLE` (allow/deny core dumps); `PR_SET_PDEATHSIG` (signal to send when parent dies — `arg2` is the signal); `PR_SET_NO_NEW_PRIVS` (irreversibly disallow privilege escalation, e.g., setuid); `PR_GET_SECCOMP`/`PR_SET_SECCOMP` (seccomp sandboxing).

```
int getgroups(int size, gid_t list[])
```

↪ Fills `list` with the supplementary group IDs of the calling process. Returns count. Pass `size=0` to query count without filling.

```
int setgroups(size_t size, const gid_t *list)
```

↪ Sets supplementary groups. Root only.

```
long clone(unsigned long flags, void *stack, int *ptid, int *ctid, unsigned long tls)
```

↪ The Linux primitive underlying both `fork()` and `pthread_create()`. `flags` controls what is shared: `CLONE_VM` (share address space), `CLONE_FS` (share filesystem root/cwd), `CLONE_FILES` (share fd table), `CLONE_SIGHAND` (share signal handlers), `CLONE_THREAD` (same thread group as parent, needed for POSIX threads). Combined: `fork=SIGCHLD`, `pthread_create=CLONE_VM|CLONE_FS|CLONE_FILES|CLONE`

Exiting

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

↪ Immediately terminates the process. Does NOT flush stdio buffers or call `atexit` handlers. Use in the child after `fork` when not exec-ing, to avoid double-flushing the parent's buffers.

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

↪ Flushes all stdio buffers, calls `atexit`/`on_exit` handlers in LIFO order, then calls `_exit`. The normal way to terminate a program.

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

↪ Registers `func` to be called at normal exit (up to 32 guaranteed). Returns 0 on success.

Environment Variables

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

↪ Returns pointer to the value of env variable `name`, or `NULL` if not found. Do not modify the returned string.

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

↪ Sets `name=value`. If `overwrite==0` and `name` already exists, does nothing.

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

↪ Removes `name` from the environment.

FILE I/O — SYSTEM CALLS

Opening and Closing

```
int open(const char *path, int flags, [mode_t mode])
```

↪ Opens or creates a file, returning a new file descriptor. `flags` must include one of `O_RDONLY`, `O_WRONLY`, `O_RDWR`. Additional flags: `O_CREAT` (create if absent, needs `mode`), `O_TRUNC` (truncate on open), `O_APPEND` (writes always go to end), `O_EXCL` (fail if exists; atomic creation), `O_NONBLOCK` (non-blocking I/O), `O_CLOEXEC` (close on exec; avoids fd leaks). Returns fd $\geq$ 0 on success, -1 on error.

```
int close(int fd)
```

↪ Releases the file descriptor. The underlying file description is closed when its last fd is closed. Always check the return value — errors can mean data was not flushed to disk (e.g., NFS).

Reading and Writing

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

↪ Reads up to `count` bytes from `fd` into `buf`. Returns bytes read ($\geq$ 0); 0 means EOF; -1 means error. Short reads are normal on pipes/terminals/signals — always loop until you have all data or EOF.

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

↪ Writes up to `count` bytes. Returns bytes written; `-1` on error. Short writes can occur; loop to write all. Writing to a pipe with no readers gives `EPIPE` error and `SIGPIPE`.

```
ssize_t readv(int fd, const struct iovec *iov, int iovcnt)
```

↪ Scatter read: fills multiple non-contiguous buffers in one syscall. Atomic for pipes/sockets. Each `iovec` has `iov_base` (pointer) and `iov_len` (bytes).

```
ssize_t writev(int fd, const struct iovec *iov, int iovcnt)
```

↪ Gather write: sends multiple buffers atomically. Critical for sending a header + body without risk of them being split into separate packets.

Positional and Atomic I/O

```
ssize_t pread(int fd, void *buf, size_t count, off_t offset)
```

↪ Like `read` but reads from `offset` without changing the file's current position. Atomic with respect to the offset — safe to call from multiple threads on the same fd without a mutex around lseek+read.

```
ssize_t pwrite(int fd, const void *buf, size_t count, off_t offset)
```

↪ Like `write` but writes at `offset` without changing the file's current position. Thread-safe for concurrent writers to different regions of the same file.

Seeking

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

↪ Moves the file offset. `SEEK_SET`: absolute; `SEEK_CUR`: relative to current; `SEEK_END`: relative to end (use `offset=0` to find file size). Returns new offset. Not valid on pipes, sockets, or FIFOs.

File Control

```
int fcntl(int fd, int cmd, ...)
```

↪ General-purpose fd control. `F_GETFL`/`F_SETFL`: get/set file status flags (e.g., add `O_NONBLOCK` after open); `F_GETFD`/`F_SETFD`: get/set fd flags (`FD_CLOEXEC`); `F_DUPFD`: dup to lowest available fd; `F_SETLK`/`F_SETLKW`: POSIX advisory record locks (cooperative only; no enforcement).

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

↪ Duplicates `oldfd` to the lowest available fd. Both share the same file description (same offset, flags).

```
int dup2(int oldfd, int newfd)
```

↪ Duplicates `oldfd` to exactly `newfd`, atomically closing `newfd` first if open. Classic use: `dup2(fd,1)` makes `fd` the new stdout before `exec`.

```
int dup3(int oldfd, int newfd, int flags)
```

↪ Like `dup2` but atomically sets flags on the new fd. Only flag currently supported: `O_CLOEXEC`. Also fails if `oldfd==newfd` (unlike `dup2`).

Permissions and Access

```
int access(const char *path, int mode)
```

↪ Checks whether the calling process can access `path` using the *real* UID/GID (not effective). `mode`: `F_OK` (exists?), `R_OK`, `W_OK`, `X_OK`. **TOCTOU hazard**: between `access` and the actual `open`, permissions can change. Prefer just `open` and handling `EACCES`.

```
int faccessat(int dirfd, const char *path, int mode, int flags)
```

↪ Like `access` but relative to `dirfd` (or `AT_FDCWD` for CWD). `flags`: `AT_EACCESS` (use effective UID/GID instead of real), `AT_SYMLINK_NOFOLLOW`.

Directory-relative (`*at`) Syscalls

```
int openat(int dirfd, const char *path, int flags, ...)
```

↪ Like `open` but resolves relative paths starting from `dirfd` instead of CWD. Use `AT_FDCWD` as `dirfd` for normal CWD behavior. Key for race-free directory traversal and chroot-less sandboxing: open a directory fd, then `openat` everything relative to it.

```
int fstatat(int dirfd, const char *path, struct stat *sb, int flags)
```

↪ Like `stat`/`lstat` but relative to `dirfd`. `flags=AT_SYMLINK_NOFOLLOW` gives `lstat` behavior. `flags=AT_EMPTY_PATH` with `path=""` stats the `dirfd` itself.

```
int unlinkat(int dirfd, const char *path, int flags)
```

↪ Like `unlink` or `rmdir` (with `AT_REMOVEDIR` flag) but relative to `dirfd`.

```
int renameat(int olddirfd, const char *old, int newdirfd, const char *new)
```

↪ Like `rename` but both paths are resolved relative to their respective directory fds.

```
int mkdirat(int dirfd, const char *path, mode_t mode)
```

↪ Like `mkdir` but relative to `dirfd`.

Zero-copy I/O (Linux)

```
ssize_t sendfile(int out_fd, int in_fd, off_t *offset, size_t count)
```

↪ Copies `count` bytes from `in_fd` to `out_fd` entirely in kernel space — no data is copied to userspace. `in_fd` must be `mmap`-able (a regular file); `out_fd` is typically a socket. `offset`: if non-NULL, reads from that offset (and updates it) without moving `in_fd`'s position. The classic way to serve static files efficiently (nginx uses this).

```
ssize_t splice(int fd_in, off64_t *off_in, int fd_out, off64_t *off_out, size_t len, unsigned int flags)
```

↪ Moves data between two fds via a kernel pipe buffer, with no user-space copy and no `mmap` requirement. At least one fd must be a pipe. `flags`: `SPLICE_F_MOVE`, `SPLICE_F_NONBLOCK`, `SPLICE_F_MORE`. More flexible than `sendfile`.

```
ssize_t tee(int fd_in, int fd_out, size_t len, unsigned int flags)
```

↪ Duplicates data from pipe `fd_in` to pipe `fd_out` without consuming `fd_in` (the data remains readable). Both fds must be pipes. Used to "tee" a data stream to multiple consumers simultaneously.

File Metadata — stat

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

↪ Fills `sb` with file metadata. Follows symbolic links.

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

↪ Like `stat` but does NOT follow symlinks; reports the link itself.

```
int fstat(int fd, struct stat *sb)
```

↪ Like `stat` but on an open fd. Immune to TOCTOU race conditions — prefer this.

```
// struct stat key fields:
sb.st_mode   // file type + permission bits
sb.st_size   // size in bytes
sb.st_nlink  // number of hard links
sb.st_ino    // inode number (unique within fs)
sb.st_uid    sb.st_gid  // owner user and group ID
sb.st_dev    // device where file lives
sb.st_blocks // 512-byte blocks allocated
sb.st_atime  sb.st_mtime  sb.st_ctime

// File type test macros (take st_mode):
S_ISREG(m)   S_ISDIR(m)   S_ISLNK(m)
S_ISFIFO(m)  S_ISSOCK(m)  S_ISCHR(m)
     S_ISBLK(m)

// Permission bits (use with & operator):
S_IRUSR(0400) S_IWUSR(0200) S_IXUSR(0100)
S_IRGRP(0040) S_IWGRP(0020) S_IXGRP(0010)
S_IROTH(0004) S_IWOTH(0002) S_IXOTH(0001)
S_ISUID(04000)  S_ISGID(02000)  S_ISVTX(01000)
```

Filesystem Operations

```
int link(const char *old, const char *new)
```

↪ Creates a hard link: `new` points to the same inode as `old`. Increments `st_nlink`. Cannot cross filesystems; cannot link directories.

```
int unlink(const char *path)
```

↪ Removes a directory entry. If this was the last link and no process has the file open, the inode and data are freed. Unlinking an open file defers deletion until all fds are closed.

```
int rename(const char *old, const char *new)
```

↪ Atomically renames within the same filesystem. If `new` exists, it is replaced atomically (no window where neither name exists — important for safe file replacement).

```
int symlink(const char *target, const char *linkpath)
```

↪ Creates a symbolic link at `linkpath` storing the string `target`. Can cross filesystems; can point to non-existent targets.

```
ssize_t readlink(const char *path, char *buf, size_t n)
```

↪ Reads the target string of a symlink into `buf`. Not NUL-terminated; returns byte count.

```
int mkdir(const char *path, mode_t mode)
```

↪ Creates a new directory. The new directory contains only `.` and `..`. Permissions are `mode & ~umask`.

```
int rmdir(const char *path)
```

↪ Removes an empty directory.

```
int chmod(const char *path, mode_t mode)
```

↪ Changes permission bits. Only the owner or root may change permissions.

```
int chown(const char *path, uid_t owner, gid_t group)
```

↪ Changes owner and group. Only root can change

the owner.

`int truncate(const char *path, off_t len)`

↪ Sets a file's size to exactly `len` bytes. If extended, the gap is a zero-filled hole (sparse file). If shortened, data beyond `len` is lost.

`int ftruncate(int fd, off_t len)`

↪ Like `truncate` but operates on an open fd. Preferred over `truncate` to avoid TOCTOU races. Commonly used after `shm_open` or `open(O_CREAT)` to set initial size.

`int fsync(int fd)`

↪ Flushes all modified data and metadata for `fd` to the underlying storage device. Blocks until the device confirms the write. Guarantees durability (survives power loss). Expensive; use only when you need strict durability guarantees (databases, journals).

`int fdatasync(int fd)`

↪ Like `fsync` but only flushes data and metadata needed to retrieve the data (`st_size`, `st_mtime`). Does not flush non-essential metadata (`st_atime`). Faster than `fsync` for database-style workloads.

`void sync(void)`

↪ Schedules all dirty kernel buffers to be written to disk (non-blocking; returns before write completes). A process-wide "hint"; prefer `fsync`/`fdatasync` for per-file guarantees.

`int fchmod(int fd, mode_t mode)`

↪ Like `chmod` but operates on an open fd. Avoids TOCTOU race.

`int fchown(int fd, uid_t owner, gid_t group)`

↪ Like `chown` but on an open fd.

`int lchown(const char *path, uid_t owner, gid_t group)`

↪ Like `chown` but does not follow symlinks — changes ownership of the link itself.

`int futimens(int fd, const struct timespec times[2])`

↪ Sets access time (`times[0]`) and modification time (`times[1]`) of an open fd. Pass `NULL` for both to set both to current time. Pass `UTIME_NOW` or `UTIME_OMIT` in `tv_nsec` for special behavior.

`mode_t umask(mode_t mask)`

↪ Sets the process's file creation mask. Bits set in `mask` are cleared from permissions passed to `open`/`mkdir`. Returns old mask. Call `umask(0)` in servers and daemons to avoid surprises.

STDIO — BUFFERED I/O

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

↪ Opens a stream with buffering. Modes: `"r"` read, `"w"` write (truncate/create), `"a"` append, `"r+"` read+write, `"w+"` read+write (truncate), `"a+"` read+append. Add `b` for binary mode. Returns `NULL` on error.

`FILE *fdopen(int fd, const char *mode)`

↪ Wraps an existing file descriptor in a `FILE*` stream. Useful after `socket()`, `pipe()`, etc. to use formatted I/O on raw fds.

`int fclose(FILE *fp)`

↪ Flushes buffered output to the OS and closes the underlying fd. Returns `EOF` on error (e.g., disk full on final flush — always check).

Reading from Streams

`int fgetc(FILE *fp)`

↪ Reads one byte; returns it as `int` or `EOF` (`-1`) at end/error. Must use `int` (not `char`) to correctly detect EOF.

`char *fgets(char *s, int n, FILE *fp)`

↪ Reads up to `n-1` chars, stopping at (and including) newline or EOF. Always NUL-terminates. Returns `s` or `NULL` at EOF/error. The safe replacement for the banned `gets()`.

`size_t fread(void *ptr, size_t size, size_t nmemb, FILE *fp)`

↪ Reads `nmemb` objects of `size` bytes each. Returns objects read; less than `nmemb` means EOF or error — check `feof`/`ferror` to distinguish.

`int fscanf(FILE *fp, const char *fmt, ...)`

↪ Formatted input from stream. Returns items assigned, or `EOF`. Fragile with whitespace and partial matches; prefer `fgets`+`sscanf`.

Writing to Streams

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

↪ Writes one character. Returns the character written, or `EOF` on error.

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

↪ Writes string `s` (no newline added). Returns non-negative on success, `EOF` on error.

`size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *fp)`

↪ Writes `nmemb` objects of `size` bytes. Returns objects written; fewer than `nmemb` indicates an error.

`int fprintf(FILE *fp, const char *fmt, ...)`

↪ Formatted output to stream. Returns chars written, or negative on error.

`int printf(const char *fmt, ...)`

↪ Formatted output to `stdout`. Buffered; may not appear immediately on terminal.

`int dprintf(int fd, const char *fmt, ...)`

↪ Like `fprintf` but writes directly to an fd (bypasses stdio buffering). Useful for sockets or logging to a raw fd.

Buffer Control and Positioning

`int fflush(FILE *fp)`

↪ Forces any buffered output data to be written to the OS (not necessarily to disk). Pass `NULL` to flush all open output streams. Crucial before `fork()` to prevent double-output of buffered data in both parent and child.

`void setvbuf(FILE *fp, char *buf, int mode, size_t size)`

↪ Controls buffering before any I/O on the stream. `_IONBF`: unbuffered (every write goes to OS immediately); `_IOLBF`: line-buffered (flush on newline; default for terminals); `_IOFBF`: fully buffered (flush when full; default for files). Pass `buf=NULL` to let the library allocate.

`int fseek(FILE *fp, long offset, int whence)`

↪ Repositions stream's file position. Same `whence` values as `lseek`.

`long ftell(FILE *fp)`

↪ Returns current byte offset in the stream.

`int feof(FILE *fp)`

↪ Returns non-zero if the EOF indicator is set (last read hit end of file).

`int ferror(FILE *fp)`

↪ Returns non-zero if the error indicator is set.

`int fileno(FILE *fp)`

↪ Returns the underlying fd of a stream. Use to call syscalls on a stdio-wrapped fd.

String Formatting

`int snprintf(char *str, size_t n, const char *fmt, ...)`

↪ Writes formatted output to buffer `str` of size `n`; always NUL-terminates. Returns chars that *would* have been written (excl. NUL); if $\geq n$, output was truncated. Always use over `sprintf`.

`int sscanf(const char *str, const char *fmt, ...)`

↪ Parses formatted input from a string. Returns items assigned.

`int asprintf(char **strp, const char *fmt, ...)`

↪ Allocates a buffer (via `malloc`) large enough for the result and stores its pointer in `*strp`. Caller must `free(*strp)`.

DIRECTORY OPERATIONS

`DIR *opendir(const char *name)`

↪ Opens a directory for reading. Returns a `DIR*` handle, or `NULL` on error.

`DIR *fdopendir(int fd)`

↪ Opens a directory from an existing fd. Enables race-free traversal when combined with `openat` and `fstatat`.

`struct dirent *readdir(DIR *dirp)`

↪ Returns the next directory entry, or `NULL` at end (check `errno==0` vs. error). Key fields: `d_name` (filename), `d_ino` (inode), `d_type` (`DT_REG`, `DT_DIR`, `DT_LNK`, `DT_FIFO`, `DT_SOCK`, `DT_UNKNOWN`). Order is not guaranteed; use `readdir_r` in threads.

`int closedir(DIR *dirp)`

↪ Closes directory stream and frees resources.

`void rewinddir(DIR *dirp)`

↪ Resets position to beginning of directory.

`int chdir(const char *path)`

↪ Changes the process's current working directory.

`char *getcwd(char *buf, size_t size)`

↪ Copies absolute path of CWD into `buf`. Pass `NULL, 0` on glibc to get a malloc'd string.

SIGNALS

`int kill(pid_t pid, int sig)`

↪ Sends signal `sig` to a process or group. `pid>0`: specific process; `pid==0`: whole process group; `pid==-1`: all processes the caller can signal; `pid<-1`: group `|pid|`. `sig==0`: permission test only (no signal

sent).

`int raise(int sig)`

↪ Sends signal to the calling process itself. Equivalent to `kill(getpid(), sig)`.

`sighandler_t signal(int sig, sighandler_t handler)`

↪ The original (non-POSIX) way to install a signal handler. `handler` is `void (*)(int)`, or `SIG_DFL` (restore default) or `SIG_IGN` (ignore). Returns old handler, or `SIG_ERR` on error. **Avoid in new code**: behavior regarding SA_RESTART, signal mask during handler, and re-entry is implementation-defined and differs between Linux and BSD. Always prefer `sigaction`.

`int sigpending(sigset_t *set)`

↪ Fills `set` with signals that are currently blocked *and* pending delivery for the calling thread. Useful to inspect what is queued before unblocking with `sigprocmask`.

`int sigwait(const sigset_t *set, int *sig)`

↪ Atomically waits for one signal in `set` to become pending, dequeues it, and stores the number in `*sig`. Signals in `set` **must be blocked** before calling (otherwise the default action fires first). Returns 0 on success. Simpler than `sigwaitinfo` when you do not need extra info.

`int sigtimedwait(const sigset_t *set, siginfo_t *info, const struct timespec *timeout)`

↪ Like `sigwaitinfo` but returns `EAGAIN` if no signal in `set` is pending within `timeout`. Pass `timeout={0,0}` to poll (non-blocking check of pending signals).

`int pthread_sigmask(int how, const sigset_t *set, sigset_t *oldset)`

↪ Thread-safe version of `sigprocmask`. In a multi-threaded program always use this, not `sigprocmask` (whose behavior is undefined for threads). Each thread has its own signal mask. A signal delivered to the process is handled by any thread that does not have it blocked.

`int pthread_kill(pthread_t tid, int sig)`

↪ Sends signal `sig` to a specific thread `tid` within the same process. `sig=0` tests for thread existence. Useful for interrupting a thread blocked in a syscall, or for directed notification without killing the whole process.

`int signalfd(int fd, const sigset_t *mask, int flags)`

↪ Linux-specific: creates an fd that becomes readable when a signal in `mask` is pending. `read()` on it returns `struct signalfd_siginfo` records. Signals in `mask` must be blocked first. Pass `fd=-1` to create; pass an existing fd to update its mask. Composable with `epoll`/`poll` for event-loop signal handling without async-signal-safety concerns. `flags`: `SFD_NONBLOCK`, `SFD_CLOEXEC`.

```
// signalfd usage pattern:
sigset_t mask;
sigemptyset(&mask);
sigaddset(&mask, SIGINT);
sigaddset(&mask, SIGTERM);
sigprocmask(SIG_BLOCK, &mask, NULL); // block
    first!
int sfd = signalfd(-1, &mask, SFD_CLOEXEC);
// now read(sfd, &fdsi, sizeof(fdsi)) in
    event loop

// Common signals:
SIGHUP  (1)  // hangup; daemons reload config
SIGINT  (2)  // Ctrl-C; interrupt
SIGQUIT (3)  // Ctrl-\; quit + core dump
SIGABRT (6)  // abort(); failed assert
SIGBUS  (7)  // bus error (alignment fault)
SIGFPE  (8)  // arithmetic error / divide by
    zero
SIGKILL (9)  // force-kill; cannot be caught
    or ignored
SIGSEGV (11) // segmentation fault
SIGPIPE (13) // write to pipe with no readers
SIGALRM (14) // alarm() timer expired
SIGTERM (15) // polite termination request
SIGCHLD (17) // child stopped or terminated
SIGCONT (18) // resume a stopped process
SIGSTOP (19) // stop; cannot be caught or
    ignored
SIGTSTP (20) // Ctrl-Z; stoppable by handler
SIGUSR1/2    // user-defined; safe for IPC
```

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

↪ The POSIX way to install signal handlers (always prefer over `signal()`). `sa_handler`: handler function or `SIG_DFL`/`SIG_IGN`; `sa_mask`: additional signals to block while handler runs; `sa_flags`: `SA_RESTART` (auto-restart interrupted syscalls), `SA_SIGINFO` (handler gets extra info via `sa_sigaction`), `SA_NOCLDWAIT` (no zombie children).

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

↪ Blocks or unblocks signal delivery. `SIG_BLOCK`: add `set` to mask; `SIG_UNBLOCK`: remove `set`; `SIG_SETMASK`: replace mask. Blocked signals are queued (pending) until unblocked. Use to create critical sections that cannot be interrupted.

```
// Signal set manipulation:
sigemptyset(&set); // initialize empty set
sigfillset (&set); // add all signals to set
sigaddset  (&set, SIGINT);   // add one signal
sigdelset  (&set, SIGTERM);  // remove one
    signal
sigismember(&set, SIGCHLD);  // test
    membership
```

`int sigsuspend(const sigset_t *mask)`

↪ Atomically replaces the signal mask with `mask` and suspends until a signal is delivered. Restores old mask on return. This atomicity solves the race between unblocking a signal and sleeping — there is no window where the signal could be lost.

`int sigwaitinfo(const sigset_t *set, siginfo_t *info)`

↪ Synchronously waits for a signal in `set` to be pending, then dequeues it. Returns the signal number. Cleaner than `sigsuspend` for signal-driven servers.

`unsigned int alarm(unsigned int seconds)`

↪ Schedules `SIGALRM` after `seconds` of real time. Cancels any existing alarm. Returns remaining time of previous alarm. Pass 0 to cancel.

PIPES AND IPC

Anonymous Pipes

`int pipe(int pipefd[2])`

↪ Creates a unidirectional channel: `pipefd[0]` is read end, `pfd[1]` is write end. Data is FIFO byte-stream with a kernel buffer (typically 64 KB). Writing blocks when full; reading blocks when empty. Read returns EOF when all write ends are closed. Used between parent and child after `fork`.

`int pipe2(int pipefd[2], int flags)`

↪ Like `pipe` but atomically sets `O_CLOEXEC` and/or `O_NONBLOCK` on both fds. Avoids a race condition in multi-threaded programs where a child could inherit the fd before `fcntl` sets `CLOEXEC`.

Named Pipes (FIFOs)

`int mkfifo(const char *path, mode_t mode)`

↪ Creates a named pipe visible in the filesystem. Any process can open it by path (subject to permissions). Both a reader and a writer must open it before either can transfer data (open blocks until both ends are present).

Memory-Mapped Files and Shared Memory

`void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset)`

↪ Maps `length` bytes of `fd` starting at `offset` into the process's virtual address space. `addr=NULL` lets the OS choose. `prot`: `PROT_READ`, `PROT_WRITE`, `PROT_EXEC`, `PROT_NONE`. `MAP_SHARED`: writes visible to other mappers and written back to file; `MAP_PRIVATE`: copy-on-write, file not changed; `MAP_ANONYMOUS`: not backed by a file (`fd=-1`), useful for shared memory between parent/child. Returns `MAP_FAILED` on error.

`int munmap(void *addr, size_t length)`

↪ Removes the mapping. Does not guarantee `MAP_SHARED` writes are on disk; call `msync` first.

`int msync(void *addr, size_t length, int flags)`

↪ Flushes `MAP_SHARED` changes to the underlying file. `MS_SYNC`: wait until complete; `MS_ASYNC`: initiate and return; `MS_INVALIDATE`: discard cached copies.

`int mprotect(void *addr, size_t len, int prot)`

↪ Changes memory protection on a mapped region. Used for guard pages (set `PROT_NONE` after stack to catch overflows) or RWX permission control.

`int madvise(void *addr, size_t len, int advice)`

↪ Hints to the kernel about access patterns. `MADV_SEQUENTIAL`: prefetch ahead; `MADV_RANDOM`: skip prefetch; `MADV_DONTNEED`: free pages (next access re-reads/zeros); `MADV_WILLNEED`: begin read-ahead now. Does not change correctness.

POSIX Shared Memory

`int shm_open(const char *name, int oflag, mode_t mode)`

↪ Opens or creates a POSIX shared memory object (name starts with `/`). Returns an fd. Follow with `ftruncate` to set size, then `mmap` to map into address space. Link with `-lrt`.

`int shm_unlink(const char *name)`

↪ Removes the shared memory name. The object persists until all mappings are unmapped and all fds closed.

POSIX Message Queues

`mqd_t mq_open(const char *name, int oflag, ...)`

↪ Opens or creates a POSIX message queue (name starts with `/`). Each message has a prior-

ity; mq_receive always returns the highest-priority pending message first. Link with -lrt.

```
int mq_send(mqd_t mq, const char *msg, size_t len, unsigned prio)
```
↪ Enqueues a message of len bytes with priority prio (higher = more urgent). Blocks if the queue is full, unless O_NONBLOCK was set at open.

```
ssize_t mq_receive(mqd_t mq, char *msg, size_t maxlen, unsigned *prio)
```
↪ Dequeues the highest-priority pending message into msg. Blocks if empty. Stores priority in *prio.

```
int mq_unlink(const char *name)
```
↪ Removes the queue name; queue persists until all mqd_t handles are closed.

eventfd (Linux)

```
int eventfd(unsigned int initval, int flags)
```
↪ Creates an event notification fd with an internal 64-bit counter initialized to initval. write() adds to the counter; read() returns and resets the counter (blocks if zero). flags: EFD_NONBLOCK, EFD_CLOEXEC, EFD_SEMAPHORE (read decrements by 1 instead of resetting). Composable with epoll/poll. Lightweight alternative to a pipe for thread/process wakeup signaling — one syscall vs two (pipe).

inotify — Filesystem Event Monitoring (Linux)

```
int inotify_init1(int flags)
```
↪ Creates an inotify instance for watching filesystem events. Returns an fd; read() on it yields struct inotify_event records. flags: IN_NONBLOCK, IN_CLOEXEC.

```
int inotify_add_watch(int fd, const char *path, uint32_t mask)
```
↪ Adds a watch for path. mask specifies events: IN_CREATE, IN_DELETE, IN_MODIFY, IN_MOVED_FROM/IN_MOVED_TO (renames), IN_ATTRIB (metadata), IN_CLOSE_WRITE, IN_ACCESS. Returns a watch descriptor (wd).

```
int inotify_rm_watch(int fd, int wd)
```
↪ Removes a watch. The watched path will also be automatically unwatched if deleted.

THREADS — PTHREADS

Thread Lifecycle

```
int pthread_create(pthread_t *tid, const pthread_attr_t *attr, void *(*start)(void *), void *arg)
```
↪ Creates a new thread running start(arg). Threads share the process address space, fds, and signal handlers, but each has its own stack and errno. attr=NULL uses defaults (joinable, default stack size). Pthreads return error codes directly; they do NOT set errno.

```
int pthread_join(pthread_t tid, void **retval)
```
↪ Blocks until tid terminates. Stores the thread's return value (or PTHREAD_CANCELED) in *retval. Releases thread resources. A thread must be either joined or detached; otherwise it becomes a "zombie thread" leaking resources.

```
int pthread_detach(pthread_t tid)
```
↪ Marks tid as detached. Resources are freed automatically when it exits. Cannot be joined afterwards. Use for fire-and-forget threads.

```
void pthread_exit(void *retval)
```
↪ Terminates the calling thread with retval. Like returning from start, but works from any call depth.

```
int pthread_cancel(pthread_t tid)
```
↪ Requests cancellation of tid. Thread must reach a cancellation point (read, write, sleep, pthread_cond_wait, etc.) to actually be cancelled. Control with pthread_setcancelstate/pthread_setcanceltype.

Mutexes

```
int pthread_mutex_init(pthread_mutex_t *m, const pthread_mutexattr_t *attr)
```
↪ Initializes a mutex. Alternatively use static initializer PTHREAD_MUTEX_INITIALIZER. Types (via attr): PTHREAD_MUTEX_NORMAL (deadlocks on recursive lock by same thread), PTHREAD_MUTEX_RECURSIVE (allows re-locking), PTHREAD_MUTEX_ERRORCHECK (returns EDEADLK instead of deadlocking).

```
int pthread_mutex_lock(pthread_mutex_t *m)
```
↪ Acquires the mutex; blocks if already held by another thread. A thread that holds the lock must not call lock again (deadlock on NORMAL type).

```
int pthread_mutex_trylock(pthread_mutex_t *m)
```
↪ Non-blocking attempt. Returns EBUSY immediately if the mutex is already held.

```
int pthread_mutex_unlock(pthread_mutex_t *m)
```
↪ Releases the mutex. Only the owning thread should call this.

```
int pthread_mutex_destroy(pthread_mutex_t *m)
```
↪ Destroys mutex. Must be unlocked. Frees any OS resources.

Condition Variables

```
int pthread_cond_wait(pthread_cond_t *c, pthread_mutex_t *m)
```
↪ Atomically releases m and waits on c. When signaled, re-acquires m before returning. ALWAYS wrap in a while loop checking your predicate, because of spurious wakeups (the OS may wake the thread even without a signal).

```
int pthread_cond_timedwait(pthread_cond_t *c, pthread_mutex_t *m, const struct timespec *abs)
```
↪ Like pthread_cond_wait but returns ETIMEDOUT after absolute time abs (CLOCK_REALTIME).

```
int pthread_cond_signal(pthread_cond_t *c)
```
↪ Wakes at least one thread waiting on c. Call while holding the associated mutex to avoid lost-wakeup races.

```
int pthread_cond_broadcast(pthread_cond_t *c)
```
↪ Wakes ALL threads waiting on c. Use when multiple waiters might have their condition satisfied (e.g., reader-writer pattern with multiple readers).

Reader-Writer Locks

```
int pthread_rwlock_rdlock(pthread_rwlock_t *rw)
```
↪ Acquires a shared read lock. Multiple readers can hold simultaneously. Blocks if a writer holds the lock or is waiting (policy is implementation-dependent).

```
int pthread_rwlock_wrlock(pthread_rwlock_t *rw)
```
↪ Acquires an exclusive write lock. Blocks until all current readers AND writers release.

```
int pthread_rwlock_unlock(pthread_rwlock_t *rw)
```
↪ Releases a read or write lock.

Thread-Local Storage and Once

```
int pthread_once(pthread_once_t *o, void (*init)(void))
```
↪ Guarantees init() is called exactly once across all threads, regardless of how many call pthread_once concurrently. Use for lazy, thread-safe initialization of global state.

```
int pthread_key_create(pthread_key_t *key, void (*destr)(void *))
```
↪ Creates a TLS key. Each thread has its own value for this key, initially NULL. destr is called with the thread's value when the thread exits (use for cleanup like free).

```
void *pthread_getspecific(pthread_key_t key)
```
↪ Returns this thread's value for key, or NULL if not set.

```
int pthread_setspecific(pthread_key_t key, const void *value)
```
↪ Sets this thread's value for key.

SEMAPHORES (POSIX)

```
int sem_init(sem_t *sem, int pshared, unsigned val)
```
↪ Initializes an unnamed semaphore with initial value val. pshared=0: usable only within one process (between threads); pshared!=0: must be placed in shared memory for cross-process use.

```
int sem_wait(sem_t *sem)
```
↪ The P (down) operation: decrements the semaphore. If the value is 0, blocks until another thread/process calls sem_post. Returns EINTR if interrupted by a signal; use a retry loop.

```
int sem_trywait(sem_t *sem)
```
↪ Non-blocking P: returns EAGAIN immediately if the value is 0. Useful for polling or avoiding deadlock.

```
int sem_timedwait(sem_t *sem, const struct timespec *abs)
```
↪ Blocking P with an absolute timeout. Returns ETIMEDOUT if the semaphore is not decremented before abs.

```
int sem_post(sem_t *sem)
```
↪ The V (up) operation: increments the semaphore, waking one blocked waiter if any. ASYNC-SIGNAL-SAFE — the only synchronization primitive that can safely be called from a signal handler.

```
int sem_getvalue(sem_t *sem, int *val)
```
↪ Reads current semaphore value into *val. May be negative (number of waiters) on some implementations.

```
sem_t *sem_open(const char *name, int oflag, ...)
```

↪ Opens or creates a named semaphore (name starts with /). Visible to any process by name. Unlike unnamed semaphores, no shared memory setup needed.

```
int sem_close(sem_t *sem)
```

↪ Closes this handle to a named semaphore. Does not destroy the semaphore.

```
int sem_unlink(const char *name)
```

↪ Removes the semaphore name; the semaphore persists until all handles are closed.

SOCKETS

```
int socket(int domain, int type, int protocol)
```

↪ Creates a communication endpoint. `AF_INET`/`AF_INET6`/`AF_UNIX` for IPv4, IPv6, or local IPC. `SOCK_STREAM`: reliable, ordered, connection-based (TCP); `SOCK_DGRAM`: unreliable, message-based (UDP); `SOCK_RAW`: raw protocol access. `protocol=0` auto-selects. Returns an fd.

```
int bind(int sockfd, const struct sockaddr
*addr, socklen_t addrlen)
```

↪ Associates the socket with a local address and port. Servers must bind before listening. Set `sin_addr.s_addr=INADDR_ANY` to accept connections on all network interfaces.

```
int listen(int sockfd, int backlog)
```

↪ Marks socket as a passive server socket. `backlog` is the kernel queue size for completed-but-not-yet-accepted connections. Call after `bind`.

```
int accept(int sockfd, struct sockaddr *addr,
socklen_t *addrlen)
```

↪ Dequeues the next completed connection and returns a NEW socket fd for that specific connection. The original `sockfd` continues listening. Blocks if no connections are waiting. Fills `addr` with the remote address.

```
int connect(int sockfd, const struct sockaddr
*addr, socklen_t addrlen)
```

↪ Initiates a connection to the remote address. Blocks until the TCP 3-way handshake completes (for `SOCK_STREAM`) or the address is associated (for `SOCK_DGRAM`).

```
int accept4(int sockfd, struct sockaddr *addr,
socklen_t *addrlen, int flags)
```

↪ Like `accept` but atomically sets flags on the new socket fd. `flags`: `SOCK_CLOEXEC` (set `O_CLOEXEC`), `SOCK_NONBLOCK`. Avoids a race between `accept` and a subsequent `fcntl` in multi-threaded servers.

```
int socketpair(int domain, int type, int
protocol, int sv[2])
```

↪ Creates a pair of connected sockets: data written to `sv[0]` can be read from `sv[1]` and vice versa. Equivalent to a bidirectional pipe. `domain` is usually `AF_UNIX`. Used to create a communication channel between a parent and child process after `fork`, or between threads. Each end is a full socket fd.

```
int getsockopt(int s, int level, int optname,
void *optval, socklen_t *optlen)
```

↪ Reads the current value of a socket option into `*optval`. Same levels and option names as `setsockopt`. Use to verify options were actually set or to read state (e.g., `SO_ERROR` to check for async connect errors).

```
ssize_t sendmsg(int sockfd, const struct msghdr
*msg, int flags)
```

↪ The most general send function. `struct msghdr` allows scatter-gather I/O (`msg_iov`/`msg_iovlen`), specifying a destination address (`msg_name`), and sending ancillary (control) data (`msg_control`/`msg_controllen`). Ancillary data enables passing open file descriptors between processes over `AF_UNIX` sockets (`SCM_RIGHTS`) or credentials (`SCM_CREDENTIALS`).

```
ssize_t recvmsg(int sockfd, struct msghdr *msg,
int flags)
```

↪ The most general receive function. Fills scatter-gather buffers and ancillary data. `msg_flags` is filled by the kernel with `MSG_TRUNC` (datagram was truncated), `MSG_CTRUNC` (control data truncated), `MSG_EOR`, etc.

```
int getsockname(int sockfd, struct sockaddr
*addr, socklen_t *addrlen)
```

↪ Returns the local address the socket is bound to. Useful after `bind` with port 0 to find which port the OS assigned.

```
int getpeername(int sockfd, struct sockaddr
*addr, socklen_t *addrlen)
```

↪ Returns the remote address of a connected socket. Equivalent to what `accept` would have filled in `addr`.

Sending and Receiving

```
ssize_t send(int sockfd, const void *buf,
size_t len, int flags)
```

↪ Like `write` but with flags: `MSG_DONTWAIT` (non-blocking), `MSG_NOSIGNAL` (suppress `SIGPIPE` on broken connection), `MSG_MORE` (hint that more data follows; batch into one TCP segment).

```
ssize_t recv(int sockfd, void *buf, size_t len,
int flags)
```

↪ Like `read` but with flags: `MSG_WAITALL` (block until all `len` bytes received), `MSG_PEEK` (inspect data without consuming), `MSG_DONTWAIT`.

```
ssize_t sendto(int sockfd, const void *buf,
size_t len, int flags, const struct sockaddr
*dest, socklen_t addrlen)
```

↪ Like `send` but specifies destination. Necessary for connectionless (`SOCK_DGRAM`) sockets that have not called `connect`.

```
ssize_t recvfrom(int sockfd, void *buf, size_t
len, int flags, struct sockaddr *src, socklen_t
*addrlen)
```

↪ Like `recv` but fills `src` with the sender's address. Used with UDP to know who sent each datagram.

```
int setsockopt(int s, int level, int optname,
const void *optval, socklen_t optlen)
```

↪ Sets socket options. `SO_REUSEADDR`: allow re-binding in `TIME_WAIT` state (always set on servers!); `SO_REUSEPORT`: allow multiple processes on same port; `SO_KEEPALIVE`: send keepalive probes; `TCP_NODELAY`: disable Nagle's algorithm (reduces latency at the cost of throughput).

```
int shutdown(int sockfd, int how)
```

↪ Partially closes a connection without closing the fd. `SHUT_RD`: no more receives; `SHUT_WR`: send FIN (signals EOF to peer); `SHUT_RDWR`: both. Use `SHUT_WR` to signal end-of-data while still reading the peer's response.

Address Utilities

```
// Byte-order (network = big-endian):
uint32_t htonl(uint32_t h); // host->net
    32-bit
uint16_t htons(uint16_t h); // host->net
    16-bit
uint32_t ntohl(uint32_t n); // net->host
    32-bit
uint16_t ntohs(uint16_t n); // net->host
    16-bit
```

```
// Text <-> binary IP address:
int          inet_pton(int af, const char
    *src, void *dst);
const char *inet_ntop(int af, const void *src,
                      char *dst, socklen_t
    size);
// af: AF_INET or AF_INET6

// Typical IPv4 server setup:
struct sockaddr_in sa = {0};
sa.sin_family      = AF_INET;
sa.sin_port        = htons(8080);
sa.sin_addr.s_addr = INADDR_ANY;
```

```
int getaddrinfo(const char *node, const char
*service, const struct addrinfo *hints, struct
addrinfo **res)
```

↪ Modern name resolution. Handles IPv4 and IPv6 transparently. `node`: hostname or IP string; `service`: port number or name (`"http"`). Returns a linked list of addresses; iterate and try each until `connect`/`bind` succeeds. Must call `freeaddrinfo(res)` when done.

MULTIPLEXING — select / poll / epoll

```
int select(int nfds, fd_set *readfds, fd_set
*writefds, fd_set *exceptfds, struct timeval
*timeout)
```

↪ Monitors fds for readability, writability, or exceptions. `timeout=NULL` blocks forever; `timeout={0,0}` polls (returns immediately). Returns count of ready fds. Modifies the fd_set args; must reinitialize every call. Hard limit: `FD_SETSIZE` (usually 1024). Older API; prefer `poll` or `epoll`.

```
FD_ZERO(&set);        // clear the set
FD_SET(fd, &set);     // add fd to watch
FD_CLR(fd, &set);     // remove fd
FD_ISSET(fd, &set);   // test if fd is ready
// nfds = max(all fds being watched) + 1
```

```
int poll(struct pollfd *fds, nfds_t nfds, int
timeout_ms)
```

↪ Like `select` but takes an explicit array (no `FD_SETSIZE` limit, no reinitialization). Each `struct pollfd`: `fd` (which fd), `events` to watch (`POLLIN`, `POLLOUT`, `POLLPRI`), `revents` filled by kernel (`POLLERR`, `POLLHUP`, `POLLNVAL` always reported regardless of `events`). `timeout=-1` blocks forever. More scalable than `select` for <1K fds.

```
int epoll_create1(int flags)
```

↪ Creates a Linux epoll instance (kernel-maintained interest list). Returns an fd. Use `EPOLL_CLOEXEC`. Scales to millions of fds in $O(1)$ — the engine behind nginx, Redis, etc.

`int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event)`

↪ Adds (`EPOLL_CTL_ADD`), modifies (`EPOLL_CTL_MOD`), or removes (`EPOLL_CTL_DEL`) an fd from the interest list. `event.events`: `EPOLLIN`, `EPOLLOUT`, `EPOLLERR`, `EPOLLHUP`; `EPOLLET` (edge-triggered: fire only on state change, not while data remains — must drain fully); `EPOLLONESHOT` (auto-disarm after first event).

`int epoll_wait(int epfd, struct epoll_event *events, int maxevents, int timeout_ms)`

↪ Waits for events, filling up to `maxevents` entries in `events`. Returns count of ready fds. Only returns events for fds in the interest list that are actually ready.

MEMORY MANAGEMENT

`void *malloc(size_t size)`

↪ Allocates `size` bytes of uninitialized heap memory. Returns pointer aligned for any basic type, or `NULL` on failure. The allocator requests memory from the OS in large chunks (via `mmap` or `sbrk`) and subdivides; freed memory returns to the allocator pool, not immediately to the OS.

`void *calloc(size_t nmemb, size_t size)`

↪ Allocates `nmemb*size` bytes, zero-initialized. Safer than `malloc+memset` because it checks for integer overflow in the multiplication.

`void *realloc(void *ptr, size_t size)`

↪ Resizes an allocation; may move the block (returning a new pointer and freeing the old one). If `ptr=NULL`, behaves like `malloc`. Do not assign back to the same pointer — if it fails, you lose the original pointer.

`void free(void *ptr)`

↪ Returns memory to the allocator. Freeing `NULL` is a no-op. Double-free is undefined behavior (often exploitable). Passing a pointer not from `malloc`/`calloc`/`realloc` is also UB.

`int posix_memalign(void **memptr, size_t alignment, size_t size)`

↪ Allocates `size` bytes aligned to `alignment` (power of 2, multiple of `sizeof(void*)`). Required for SIMD (AVX needs 32-byte alignment), DMA buffers, and `O_DIRECT` I/O.

`void *alloca(size_t size)`

↪ Allocates on the *stack*. Freed automatically when the enclosing function returns. Fast but: no error return, can overflow stack silently, pointer becomes dangling if returned.

`void *mremap(void *old_addr, size_t old_size, size_t new_size, int flags, ...)`

↪ Linux-specific: resizes (and optionally moves) an existing `mmap` mapping. `flags=MREMAP_MAYMOVE`: allow the mapping to be relocated. `flags=MREMAP_FIXED`: move to a specific address (pass new addr as 5th argument). Avoids unmap+remap; efficient for growing large memory-mapped files.

`int mlock(const void *addr, size_t len)`

↪ Locks pages in the address range into physical RAM, preventing them from being swapped out. Used for real-time applications, cryptographic key storage, or any memory that must have bounded latency. Requires `CAP_IPC_LOCK` or `RLIMIT_MEMLOCK` budget.

`int munlock(const void *addr, size_t len)`

↪ Unlocks previously locked pages, allowing them to be swapped again.

`int mlockall(int flags)`

↪ Locks all current and/or future pages of the process. `MCL_CURRENT`: lock all existing mappings; `MCL_FUTURE`: lock all future mappings too. Used by real-time processes that must never page fault.

`int munlockall(void)`

↪ Unlocks all locked mappings for the process.

ERROR HANDLING

`char *strerror(int errnum)`

↪ Returns a human-readable string for `errnum`. Not thread-safe (uses a static buffer); use `strerror_r` in threaded code.

`void perror(const char *s)`

↪ Prints `"s: <strerror(errno)>\n"` to stderr. Quickest way to report a syscall failure in a program.

```
// Key errno values:
EPERM        // operation not permitted
EACCES       // permission denied
ENOENT       // no such file or directory
EEXIST       // file already exists
EINTR        // interrupted by signal; retry syscall
EAGAIN/EWOULDBLOCK // non-blocking op would block
EINVAL       // invalid argument
ENOMEM       // out of memory
EBADF        // bad file descriptor
EBUSY        // resource busy
EPIPE        // write to pipe with no readers
EDEADLK      // deadlock would occur
ETIMEDOUT    // operation timed out
EMFILE       // process has too many open fds
ENFILE       // system-wide open file limit hit
ENOSPC       // no space left on device
EROFS        // read-only filesystem
EXDEV        // cross-device link attempted
ENOTDIR      // expected directory, got file
EISDIR       // expected file, got directory
ENOTEMPTY    // rmdir on non-empty directory
ECONNREFUSED // nothing listening at that address
ECONNRESET   // peer forcibly closed connection
EADDRINUSE   // address/port already in use

// EINTR retry pattern:
ssize_t r;
do { r = read(fd, buf, n); }
while (r == -1 && errno == EINTR);
```

TIME

`int clock_gettime(clockid_t clk, struct timespec *tp)`

↪ High-resolution time. `CLOCK_REALTIME`: wall clock (adjusted by NTP; can jump backwards); `CLOCK_MONOTONIC`: always increases, safe for measuring elapsed time; `CLOCK_PROCESS_CPUTIME_ID`: CPU time used by the process; `CLOCK_THREAD_CPUTIME_ID`: CPU time by this thread. `tp.tv_sec` + `tp.tv_nsec` (0–999999999).

`time_t time(time_t *tloc)`

↪ Returns seconds since Unix epoch. Low-resolution (1 second) but universally available. Stores in `*tloc` if non-NULL.

`int nanosleep(const struct timespec *req, struct timespec *rem)`

↪ Sleeps for at least the duration in `req`. If interrupted by a signal, stores remaining time in `rem` and returns `EINTR` (loop to complete the sleep).

`int clock_nanosleep(clockid_t clk, int flags, const struct timespec *req, struct timespec *rem)`

↪ Like `nanosleep` but uses a specific clock and supports absolute time. `flags=TIMER_ABSTIME`: `req` is an absolute time on `clk` (not a relative duration); eliminates drift in periodic loops. `clk`: `CLOCK_MONOTONIC` recommended for timers.

`int setitimer(int which, const struct itimerval *new, struct itimerval *old)`

↪ Sets a repeating or one-shot interval timer. `ITIMER_REAL`: real time, delivers `SIGALRM`; `ITIMER_VIRTUAL`: process CPU time, delivers `SIGVTALRM`; `ITIMER_PROF`: process+kernel CPU time, delivers `SIGPROF` (used by profilers). `it_value` = initial delay; `it_interval` = period (0 = one-shot).

`int timer_create(clockid_t clk, struct sigevent *sevp, timer_t *timerid)`

↪ Creates a POSIX per-process interval timer. `sevp` controls notification: `SIGEV_SIGNAL` (send a signal), `SIGEV_THREAD` (call a function in a new thread), `SIGEV_NONE`. More flexible than `setitimer`; supports multiple independent timers. Link `-lrt`.

`int timer_settime(timer_t id, int flags, const struct itimerspec *new, struct itimerspec *old)`

↪ Arms or disarms a `timer_create` timer. `flags=TIMER_ABSTIME` for absolute expiry. `it_value`: initial expiry; `it_interval`: period (0 = one-shot).

`int timerfd_create(int clk, int flags)`

↪ Linux-specific: creates a timer as a file descriptor. Returns an fd that becomes readable when the timer expires. `flags`: `TFD_NONBLOCK`, `TFD_CLOEXEC`. Composable with `epoll`/`poll` for timeout handling in event loops without signals.

`int timerfd_settime(int fd, int flags, const struct itimerspec *new, struct itimerspec *old)`

↪ Arms/disarms a `timerfd`. Same semantics as `timer_settime`. `read()` on the fd returns an `uint64_t` with the number of expirations since last read.

PROCESS AND SYSTEM INFO

`long sysconf(int name)`

↪ Queries runtime system limits. `_SC_NPROCESSORS_ONLN`: online CPU count; `_SC_PAGESIZE`: page size (usually 4096); `_SC_OPEN_MAX`: max open fds; `_SC_CLK_TCK`: ticks/second for `times()`.

`int gettimeofday(struct timeval *tv, struct timezone *tz)`

↪ Returns wall-clock time as `tv.tv_sec` (seconds) + `tv.tv_usec` (microseconds). `tz` is obsolete; pass `NULL`. Prefer `clock_gettime(CLOCK_REALTIME)` for new code (nanosecond resolution, POSIX).

`clock_t times(struct tms *buf)`

↪ Fills `buf` with CPU times: `tms_utime` (user CPU), `tms_stime` (kernel CPU), `tms_cutime`/`tms_cstime` (same for waited-for children). Returns elapsed real time in clock ticks (`sysconf(_SC_CLK_TCK)`). Use `getrusage` for more detail.

`int getrusage(int who, struct rusage *usage)`

↪ Returns resource usage statistics. `who`: `RUSAGE_SELF` (this process), `RUSAGE_CHILDREN` (waited-for children), `RUSAGE_THREAD`. Key fields: `ru_utime` (user CPU time), `ru_stime` (system CPU time), `ru_maxrss` (peak RSS in KB), `ru_minflt`/`ru_majflt` (soft/hard page faults), `ru_nvcsw`/`ru_nivcsw` (voluntary/involuntary context switches).

`int getrlimit(int resource, struct rlimit *rlim)`

↪ Returns soft (`rlim_cur`, enforced) and hard (`rlim_max`, ceiling) limits. `RLIMIT_NOFILE`: open fds; `RLIMIT_AS`: virtual address space; `RLIMIT_STACK`: stack size; `RLIMIT_CORE`: core dump size (set to 0 to disable core dumps); `RLIMIT_NPROC`: max child processes.

`int setrlimit(int resource, const struct rlimit *rlim)`

↪ A process can lower its hard limit (irreversible without root) or raise soft up to hard. Only root can raise the hard limit. Used to sandbox child processes or tune server file descriptor limits.

`uid_t getuid(void)`

↪ Returns real user ID (who launched the process).

`uid_t geteuid(void)`

↪ Returns effective user ID (what permissions are used; differs from real UID if the binary is setuid).

`int setuid(uid_t uid)`

↪ Changes real, effective, and saved UIDs to `uid`. Root can set to any UID. Non-root can only restore to real UID. Used to permanently drop root privileges.

`int seteuid(uid_t uid)`

↪ Changes only effective UID. Allows temporarily suspending/restoring elevated privileges without permanently dropping them.

x86 I/O PORTS (DEVICE DRIVERS)

Overview

x86 has two distinct address spaces: **memory** (accessed via load/store) and **I/O ports** (accessed via special `IN`/`OUT` instructions). Port-mapped I/O (PMIO) is used by legacy hardware: the 8259 PIC, 8253/8254 PIT timer, 16550 UART serial port, IDE controllers, and the PS/2 controller. Modern hardware prefers memory-mapped I/O (MMIO) instead. In the Linux kernel, these live in `<asm/io.h>`; in userspace, `<sys/io.h>` with privilege setup required.

Userspace Setup (requires root)

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

// Grant access to 'count' ports starting at
     'from':
int ioperm(unsigned long from, unsigned long
     count,
           int turn_on);
// turn_on=1 grants, turn_on=0 revokes.
// Only works for ports 0x000-0x3FF.
// Process must be root (or have
     CAP_SYS_RAWIO).

// Grant access to ALL I/O ports (dangerous):
int iopl(int level);
// level=3: full I/O port access.
// level=0: revoke (back to normal).
// Inherited across fork(); lost on exec().
```

Port Write Functions (outb family)

`void outb(unsigned char val, unsigned short port)`

↪ Writes one byte `val` to I/O port `port`. Maps to the x86 `OUT` instruction. Used to send commands or data to hardware registers (e.g., send an EOI to the PIC at port `0x20`, or set PIT frequency at port `0x40`).

`void outw(unsigned short val, unsigned short port)`

↪ Writes a 16-bit word to `port`. Used when a device register is 16-bits wide.

`void outl(unsigned int val, unsigned short port)`

↪ Writes a 32-bit double-word to `port`. Used for 32-bit hardware registers.

`void outb_p(unsigned char val, unsigned short port)`

↪ Like `outb` but inserts a small I/O delay (a pause) after the write. Needed for very slow legacy ISA devices that cannot keep up with modern CPU speeds.

`void outw_p(unsigned short val, unsigned short port)`

↪ Like `outw` with I/O delay.

`void outl_p(unsigned int val, unsigned short port)`

↪ Like `outl` with I/O delay.

Port Read Functions (inb family)

`unsigned char inb(unsigned short port)`

↪ Reads one byte from I/O port `port`. Maps to the x86 `IN` instruction. Used to read hardware status registers (e.g., read the PIC's in-service register, or check UART's line status register at port `0x3F8+5`).

`unsigned short inw(unsigned short port)`

↪ Reads a 16-bit word from `port`.

`unsigned int inl(unsigned short port)`

↪ Reads a 32-bit double-word from `port`.

`unsigned char inb_p(unsigned short port)`

↪ Like `inb` with I/O delay for slow legacy hardware.

`unsigned short inw_p(unsigned short port)`

↪ Like `inw` with I/O delay.

`unsigned int inl_p(unsigned short port)`

↪ Like `inl` with I/O delay.

String (Repeated) I/O

`void outsb(unsigned short port, const void *addr, unsigned long count)`

↪ Writes `count` bytes from memory buffer `addr` to `port` in a tight loop. Uses the x86 `REP OUTSB` instruction — much faster than calling `outb` in a loop. Used for bulk data transfers to devices like IDE/ATA.

`void outsw(unsigned short port, const void *addr, unsigned long count)`

↪ Like `outsb` but transfers 16-bit words (`REP OUTSW`).

`void outsl(unsigned short port, const void *addr, unsigned long count)`

↪ Like `outsb` but transfers 32-bit words (`REP OUTSL`).

`void insb(unsigned short port, void *addr, unsigned long count)`

↪ Reads `count` bytes from `port` into memory buffer `addr` using `REP INSB`. Used for bulk input from hardware (e.g., reading a sector from an ATA drive).

`void insw(unsigned short port, void *addr, unsigned long count)`

↪ Like `insb` but 16-bit words.

`void insl(unsigned short port, void *addr, unsigned long count)`

↪ Like `insb` but 32-bit words.

Memory-Mapped I/O (MMIO) — Kernel

`void __iomem *ioremap(phys_addr_t offset, size_t size)`

↪ Maps a physical address range (hardware register space) into kernel virtual address space. Required before accessing MMIO registers. Returns a pointer tagged `__iomem` (sparse annotation; do not dereference directly).

`void iounmap(void __iomem *addr)`

↪ Unmaps a region previously mapped with `ioremap`.

`unsigned int readb(const volatile void __iomem *addr)`

↪ Reads one byte from an MMIO address. Must use `readb`/`readw`/`readl` (not plain `*addr`) to ensure proper ordering and prevent the compiler from optimizing away the access.

`unsigned int readw(const volatile void __iomem *addr)`

↪ Reads 16-bit word from MMIO address.

`unsigned int readl(const volatile void __iomem *addr)`

↪ Reads 32-bit word from MMIO address.

`void writeb(unsigned char val, volatile void __iomem *addr)`

↪ Writes one byte to MMIO address.

`void writew(unsigned short val, volatile void __iomem *addr)`

↪ Writes 16-bit word to MMIO address.

`void writel(unsigned int val, volatile void __iomem *addr)`

↪ Writes 32-bit word to MMIO address.

Memory Barriers (ordering)

`void mb(void)`

↪ Full memory barrier: ensures all memory reads and writes before the barrier complete before any after it. Prevents CPU and compiler reordering across the barrier. Required when ordering matters between MMIO accesses and regular memory.

`void rmb(void)`

↪ Read memory barrier: ensures all reads before the barrier complete before reads after it. Lighter than `mb()`.

```
void wmb(void)
```

↪ Write memory barrier: ensures all writes before the barrier are visible before writes after it.

Key I/O Port Addresses (x86)

Port(s)	Device / Use
0x20-0x21	Master 8259 PIC (IRQ 0-7)
0xA0-0xA1	Slave 8259 PIC (IRQ 8-15)
0x40-0x43	8253/8254 PIT (timer channels 0-2)
0x60, 0x64	PS/2 keyboard/mouse controller
0x3F8-0x3FF	COM1 serial port (16550 UART)
0x2F8-0x2FF	COM2 serial port
0x1F0-0x1F7	Primary IDE/ATA controller
0x3F6	Primary IDE control register
0x80	POST/diagnostic port (write delay)
0x70-0x71	CMOS RTC / NMI control
0xCF8-0xCFF	PCI config space access

Common Patterns

```
// Send End-Of-Interrupt (EOI) to PIC after IRQ:
outb(0x20, 0x20);  // to master PIC
outb(0x20, 0xA0);  // also to slave (IRQ 8-15)

// Set PIT channel 0 to rate generator (mode 2):
outb(0x34, 0x43);         // command: ch0, lobyte/hibyte, mode2
outb(divisor & 0xFF, 0x40);    // low byte of divisor
outb((divisor >> 8) & 0xFF, 0x40); // high byte
// divisor = 1193182 / desired_hz

// Read a byte from COM1 (poll until ready):
while (!(inb(0x3F8 + 5) & 0x01)); // wait: data ready bit
char c = inb(0x3F8);                  // read received byte

// Write a byte to COM1:
while (!(inb(0x3F8 + 5) & 0x20)); // wait: tx buffer empty
outb(c, 0x3F8);                       // send byte
```

PMIO vs MMIO Comparison

	PMIO	MMIO
Access	`in`/`out` instrs	load/store
Space	64KB (16-bit)	full addr space
Hardware	Legacy ISA	PCI, modern SoC
Kernel	`<asm/io.h>`	`ioremap` + `readl`/`writel`
User	`<sys/io.h>` + `ioperm`	`mmap` `/dev/mem`

USEFUL POSIX UTILITIES

Memory Operations (<string.h>)

```
void *memset(void *s, int c, size_t n)
```

↪ Fills the first `n` bytes of `s` with byte value `c`. Returns `s`. Use `memset(buf, 0, n)` to zero-initialize. Note: `memset` may be optimized away by the compiler when zeroing sensitive data — use `explicit_bzero` or `memset_s` for security-critical zeroing.

```
void *memcpy(void *dst, const void *src, size_t n)
```

↪ Copies `n` bytes from `src` to `dst`. Regions must not overlap — undefined behavior if they do. Use `memmove` if overlap is possible.

```
void *memmove(void *dst, const void *src, size_t n)
```

↪ Like `memcpy` but correctly handles overlapping regions (e.g., shifting data within a buffer). Slightly slower than `memcpy`; always safe to use.

```
void *memchr(const void *s, int c, size_t n)
```

↪ Scans `n` bytes of `s` for byte `c`. Returns pointer to first match, or `NULL`.

```
int memcmp(const void *s1, const void *s2, size_t n)
```

↪ Compares `n` bytes lexicographically. Returns `<0`, `0`, or `>0`. Not constant-time — use `CRYPTO_memcmp` for password/key comparison.

String Utilities (<string.h>)

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

↪ Returns length of string (not counting NUL).

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

↪ Copies string to `dst`. Unsafe if `src` is longer than `dst` buffer. Prefer `strlcpy` or `snprintf`.

```
char *strncpy(char *dst, const char *src, size_t n)
```

↪ Copies at most `n` bytes. Does NOT guarantee NUL-termination if `src` is $\geq n$ bytes. Pads with NUL if shorter. Quirky — prefer `strlcpy` or `snprintf`.

```
size_t strlcpy(char *dst, const char *src, size_t size)
```

↪ BSD/POSIX.1-2024: copies at most `size-1` bytes, always NUL-terminates. Returns length of `src` (to detect truncation). Safer than `strcpy`/`strncpy`.

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

↪ Appends `src` to `dst`. Unsafe; prefer `strlcat` or `snprintf`.

```
int strcmp(const char *s1, const char *s2)
```

↪ Lexicographic comparison. Returns `<0`, `0`, or `>0`.

```
int strncmp(const char *s1, const char *s2, size_t n)
```

↪ Compare at most `n` characters.

```
char *strchr(const char *s, int c)
```

↪ Finds first occurrence of character `c` in string. Returns pointer or `NULL`.

```
char *strrchr(const char *s, int c)
```

↪ Finds last occurrence of character `c`.

```
char *strstr(const char *haystack, const char *needle)
```

↪ Finds first occurrence of substring `needle`.

```
char *strtok_r(char *str, const char *delim, char **saveptr)
```

↪ Thread-safe tokenizer. First call: `str=input`; subsequent calls: `str=NULL`. `saveptr` holds state between calls. Modifies the input string (inserts NUL bytes). Use over `strtok` in threaded code.

```
char *strdup(const char *s)
```

↪ Allocates a copy of string `s` via `malloc`. Caller must `free`.

```
char *strndup(const char *s, size_t n)
```

↪ Like `strdup` but copies at most `n` characters.

Number Parsing (<stdlib.h>)

```
long strtol(const char *str, char **endptr, int base)
```

↪ Converts string to `long`. Skips leading whitespace. `base=0`: auto-detect (0x=hex, 0=octal, else decimal). Stores pointer to first unconverted char in `*endptr` — always check this to detect partial conversion or errors. Returns `LONG_MIN`/`LONG_MAX` and sets `errno=ERANGE` on overflow.

```
long long strtoll(const char *str, char **endptr, int base)
```

↪ Like `strtol` for `long long`.

```
unsigned long strtoul(const char *str, char **endptr, int base)
```

↪ Like `strtol` for `unsigned long`.

```
double strtod(const char *str, char **endptr)
```

↪ Converts string to `double`. Handles NaN, Inf. Check `*endptr` and `errno=ERANGE` for overflow/underflow.

```
int atoi(const char *s)
```

↪ Converts string to `int`. No error detection — use `strtol` for robust code.

Program Arguments (<unistd.h>)

```
int getopt(int argc, char *const argv[], const char *optstring)
```

↪ Parses command-line options. Call repeatedly; returns the next option character, `-1` when done, or `'?'` for unknown option. `optstring`: each char is an option letter; `':'` after means it takes an argument (stored in `optarg`). Global `optind` is the index of the next argument.

```
int getopt_long(int argc, char *const argv[], const char *optstring, const struct option *longopts, int *longindex)
```

↪ Like `getopt` but also handles long options (`-option`). `longopts` is a null-terminated array of `struct option {name, has_arg, flag, val}`.

Non-local Jumps (<setjmp.h>)

```
int sigsetjmp(sigjmp_buf env, int savemask)
```

↪ Like `setjmp` but also saves the current signal mask if `savemask!=0`. Use when calling `siglongjmp` from within or after a signal handler, so the signal mask is properly restored.

```
void siglongjmp(sigjmp_buf env, int val)
```

↪ Like `longjmp` but restores the signal mask that was saved by `sigsetjmp`. Always use this pair (`sigsetjmp`/`siglongjmp`) when jumping out of signal handlers, not `setjmp`/`longjmp`.

Additional Pthread Primitives

```
int pthread_attr_init(pthread_attr_t *attr)
```

↪ Initializes a thread attribute object with defaults. Must call before using any `pthread_attr_set*` functions.

`int pthread_attr_setstacksize(pthread_attr_t *attr, size_t stacksize)`

↪ Sets the stack size for threads created with this attr. Default is typically 8 MB. Reduce for embedded/high-thread-count scenarios; increase for deeply recursive code.

`int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate)`

↪ Sets the initial detach state: `PTHREAD_CREATE_JOINABLE` (default) or `PTHREAD_CREATE_DETACHED`. Detached threads do not need `pthread_join`.

`int pthread_attr_destroy(pthread_attr_t *attr)`

↪ Destroys the attribute object. Does not affect threads already created with it.

`int pthread_spin_init(pthread_spinlock_t *lock, int pshared)`

↪ Initializes a spinlock. Unlike a mutex, a spinning thread busy-waits (burns CPU) instead of sleeping. Only appropriate when the critical section is extremely short and the system has multiple CPUs.

`int pthread_spin_lock(pthread_spinlock_t *lock)`

↪ Acquires spinlock; busy-waits until available.

`int pthread_spin_trylock(pthread_spinlock_t *lock)`

↪ Non-blocking; returns `EBUSY` if not available.

`int pthread_spin_unlock(pthread_spinlock_t *lock)`

↪ Releases spinlock.

`int pthread_spin_destroy(pthread_spinlock_t *lock)`

↪ Destroys spinlock.

`int pthread_barrier_init(pthread_barrier_t *b, const pthread_barrierattr_t *attr, unsigned count)`

↪ Creates a barrier for `count` threads. All threads must arrive at the barrier before any can proceed. Used to synchronize phases of parallel computation.

`int pthread_barrier_wait(pthread_barrier_t *b)`

↪ Blocks until `count` threads have called `pthread_barrier_wait`. One thread gets `PTHREAD_BARRIER_SERIAL_THREAD` as return value (to elect a single thread for post-barrier cleanup); all others get 0.

`int pthread_barrier_destroy(pthread_barrier_t *b)`

↪ Destroys the barrier.

`int pthread_atfork(void (*prepare)(void), void (*parent)(void), void (*child)(void))`

↪ Registers handlers called around `fork()`: `prepare` (in parent, before fork; lock mutexes), `parent` (in parent, after fork; unlock), `child` (in child, after fork; unlock/reinitialize). Critical for making mutex-using libraries fork-safe.

`int setjmp(jmp_buf env)`

↪ Saves the current execution context (registers, stack pointer) into `env`. Returns `0` on initial call. Used with `longjmp` for non-local exits from deeply nested functions (e.g., error recovery in parsers or longjmp-based exception handling).

`void longjmp(jmp_buf env, int val)`

↪ Restores context saved by `setjmp`, making it appear `setjmp` returned `val` (must be non-zero; zero becomes 1). Unwinds stack without calling destructors. For signal-safe version that also restores signal mask, use `sigsetjmp`/`siglongjmp`.

`void *dlopen(const char *filename, int flags)`

↪ Loads a shared library at runtime. `RTLD_LAZY`: resolve symbol addresses on first use; `RTLD_NOW`: resolve all at load time (fails fast if symbols missing); `RTLD_GLOBAL`: make symbols available to subsequently loaded libraries. Returns opaque handle or `NULL`.

`void *dlsym(void *handle, const char *symbol)`

↪ Finds the address of `symbol` in the library. Returns function pointer (cast before use) or `NULL`. Use `RTLD_DEFAULT` to search all currently loaded libraries. Always call `dlerror()` after to check for errors (`NULL` return can be a valid address).

`char *dlerror(void)`

↪ Returns and clears the most recent `dl*` error string, or `NULL` if none.

`int dlclose(void *handle)`

↪ Decrements reference count; library unloaded when count hits zero.

QUICK REFERENCE

Return Value Summary

Function	Success	Failure
`open/openat`	fd ≥ 0	-1, errno
`read/write`	bytes ≥ 0	-1, errno
`pread/pwrite`	bytes ≥ 0	-1, errno
`fork`	0 or child pid	-1, errno
`waitpid/waitid`	child pid	-1, errno
`exec*`	never returns	-1, errno
`malloc/calloc`	ptr	NULL
`mmap/mremap`	ptr	MAP_FAILED
`pthread_*`	0	errcode (not errno!)
`sem_wait/post`	0	-1, errno
`socket/accept`	fd ≥ 0	-1, errno
`connect/bind`	0	-1, errno
`select/poll`	# ready	-1, errno
`epoll_wait`	# ready	-1, errno
`clock_gettime`	0	-1, errno
`shm_open`	fd ≥ 0	-1, errno
`eventfd`	fd ≥ 0	-1, errno
`timerfd_create`	fd ≥ 0	-1, errno
`sendfile/splice`	bytes copied	-1, errno
`signalfd`	fd ≥ 0	-1, errno
`strtol/strtod`	converted value	check `*endptr` + errno

Standard File Descriptors

FD	Macro	FILE*
0	`STDIN_FILENO`	`stdin`
1	`STDOUT_FILENO`	`stdout`
2	`STDERR_FILENO`	`stderr`

Key Headers and Link Flags

Header	Contents
`<unistd.h>`	read/write, fork, exec*, close, dup, pread/pwrite
`<fcntl.h>`	open, openat, fcntl, O_* flags
`<sys/wait.h>`	wait, waitpid, waitid, W* macros
`<signal.h>`	signal, sigaction, kill, sigprocmask, signalfd
`<sys/mman.h>`	mmap, munmap, mprotect, mlock, mremap
`<pthread.h>`	all pthreads (`-lpthread`)
`<semaphore.h>`	sem_* (`-lpthread`)
`<sys/socket.h>`	socket, bind, listen, accept, sendmsg
`<netinet/in.h>`	sockaddr_in, INADDR_ANY
`<sys/epoll.h>`	epoll_* (Linux only)
`<sys/eventfd.h>`	eventfd (Linux only)
`<sys/inotify.h>`	inotify_* (Linux only)
`<sys/timerfd.h>`	timerfd_* (Linux only)
`<sys/sendfile.h>`	sendfile (Linux only)
`<mqueue.h>`	mq_* (`-lrt`)
`<time.h>`	clock_gettime, nanosleep, timer_*
`<sys/resource.h>`	getrlimit, setrlimit, getrusage
`<sys/times.h>`	times()
`<dlfcn.h>`	dlopen, dlsym (`-ldl`)
`<setjmp.h>`	setjmp, longjmp, sigsetjmp, siglongjmp
`<string.h>`	memset, memcpy, memmove, str*
`<stdlib.h>`	strtol, strtod, atoi, malloc, free
`<sys/prctl.h>`	prctl (Linux only)
`<sys/io.h>`	inb/outb userspace (`ioperm`/`iopl` needed)
`<asm/io.h>`	inb/outb kernel, ioremap, readl/writel