# CS100 Syscall Cheatsheet

Read your man pages, and sleep once in a while, k?

This document contains interfaces, libraries, descriptions, and errors for syscalls used in CS100. They're organized by category and order of use in the class (as of Winter 2015). For your linux's safety and your grade's safety, please use `perror()` with every syscall.

| | |
|---|---|
| `perror()` | `void perror(const char* s)` — prints `s` and syscall's error message as defined by global int `errno` to `stderr`. If syscall has error, `errno` is changed.<br>`stdio.h`, `errno.h` |

## 1 Syscall interface and libraries

| Function | Interface, Include, Description |
|---|---|
| `fork()` | `pid_t fork(void)` — Creates child process. Returns `0` to child process and child's process id to parent process, or `-1` if error. No child process made if error occurs.<br>`unistd.h` |
| `exec` | `Exec` note: If `exec` succeeds, the current process will end and `exec` will not return. It returns `-1` if it fails (e.g. program file not found). `char *const argv` also must be `NULL` terminated. |
| `execv()` | `int execv(const char* path, char *const argv[])` — Executes program `path` and passes arguments `argv`. Requires full path name of program.<br>`unistd.h` |
| `execvp()` | `int execvp(const char* file, char *const argv[])` Executes program `path` and passes arguments `argv`., finds program file automatically by checking directories in environmental variable `PATH`.<br>`unistd.h` |
| `wait()` | `pid_t wait(int* status)` — Waits for child process to terminate. `int* status` stores exit status of child process. Use `NULL` if not needed. Returns child pid if succeeds, `-1` if fails (e.g. no child).<br>`sys/wait.h` |
| `waitpid()` | `pid_t waitpid(pid_t pid, int* status, int options)` — Similar to `wait()`. Can specify `pid` to wait for specific child; use 0 to wait for any child. Can also wait for stopped processes by adding option `WUNTRACED` and check immediately instead of waiting with `WNOHANG` (Bitwise OR to combine: '`WUNTRACED \| WNOHANG`'). Returns child pid if succeeds, `-1` if fails (e.g. invalid pid).<br>`sys/wait.h` |
| Directories and Files | Note: When these functions require a directory or file name, they only require a *relative path*. This means that if your process was called in directory `bin/foo/bar/`, instead of using `bin/foo/bar/p.cpp`, the *absolute path*, as your parameter, you can use `p.cpp` or `./p.cpp` instead. |
| `opendir()` | `DIR* opendir(const char* name)` — opens directory stream to directory `name` and returns pointer to its first entry. Returns `NULL` on error.<br>`dirent.h` |
| `closedir()` | `int closedir(DIR* dirp)` — Closes directory. returns `0` on success, `-1` if error<br>`dirent.h` |
| `chdir()` | `int chdir(const char* path)` — change directory of calling process to `path`. Returns `0` on success, `-1` if error.<br>`sys/stat.h`, `unistd.h` |

| | |
|---|---|
| `stat()` | `int stat(const char* path, struct stat* buf)` — Gives information about a file in `struct stat`, e.g. permissions, type of file, time created. See `ls -l` for example of provided information. Macros are also provided that take `mode_t st_mode` in `struct stat` and returns `true`/`false` (e.g. `S_ISDIR(st_mode)`, `S_REG(st_mode)`). Returns `0` on success, `-1` if error.<br>`sys/types.h`, `sys/stat.h`, `unistd.h` |
| `open()` | `int open(const char* pathname, int flags)`<br>`int open(const char* pathname, int flags, mod_t mode)`<br>Opens file and returns file descriptor which can be used, with flags, to read/write/create file.<br>**Flags:** Must use either `O_RDONLY` (read file), `O_WRONLY` (write to file), or `O_RDWR` (both) in call. These can be bitwise `OR`'d (`'O_WRONLY \| O_CREAT'`) with other flags such as: `O_CREAT` creates file if it doesn't exist. `O_TRUNC` overwrites contents of file when writing, `O_APPEND` writes at the end of file instead.<br>**Modes:** When creating files, `mode` arguments can be added to specify permissions of new file, such as `S_IRWXU` - user has read/write/execute permission, `S_IRUSR` - user has read permission, or `S_IWUSR` - user has read permission.<br>**Warning:** Every call to `open()` must have a corresponding `close()` or else file descriptors will be left open (similar to memory leaks with `new` and `delete`).<br>`sys/types.h`, `sys/stat.h`, `fcntl.h` |
| `close()` | `int close(int fd)` — Close file descriptor. Returns `0` on success, `-1` if error (e.g. invalid fd).<br>`unistd.h` |
| `dup()` | `int dup(int oldfd)` — copies file descriptor `oldfd` to the next lowest unused descriptor, returns new file descriptor on success, `-1` if error.<br>**Warning:** be sure to close copies after you finish using them. File descriptors are a limited resource.<br>`unistd.h` |
| `dup2()` | `int dup2(int oldfd, int newfd)` — Copies file descriptor `oldfd` in `newfd`. Closes `newfd` if it already exists. Returns new file descriptor on success, `-1` if error<br>`unistd.h` |
| `pipe()` | `int pipe(int pipefd[2])` — Returns two file descriptors in `pipefd` for reading from and writing to. The file descriptors are one way pipes only, hence the need for two. Data written to `pipefd[1]` can be read from `pipefd[0]`. `pipe` is usually associated with piping in bash (e.g. `ls \| grep .cpp`).<br>`unistd.h` |
| Signals | `SIGINT` to interrupt (`Ctrl+c`), `SIGTSTP` to temporarily stop (`Ctrl+z`), or `SIGSEGV` (segfault). |
| `signal()` | `sighandler_t signal(int signum, sighandler_t handler)` — sets signal handler for signal `signum` to function `handler`. For signal handlers: `SIG_IGN` ignores signal and `SIG_DFL` uses default handler. You can also use a user-defined handler as an argument. `sighandler_t` is a pointer to a void function that has an int parameter: `void myhandler(int sig)`<br>e.g. `signal(SIGINT, myhandler);`<br>`signal.h` |
| `kill()` | `int kill(pid_t pid, int sig)` — Sends any signal to process or proccess group. Returns `0` on success, `-1` if error.<br>`sys/types.h`, `signal.h` |
| `getcwd()` | `char* getcwd(char*buf, size_t size)` — Returns current working directory on success and copies to `buf` of size `size` if not `NULL`. Returns `NULL` if error.<br>`unistd.h` |
| `gethostname()` | `int gethostname(char* name, size_t len)` — Writes hostname (`hammer.cs.ucr.edu`) to char array `name` with size `len`. According to the man page, the guaranteed maximum value for `len` is `HOST_NAME_MAX = 64`, which is in `limits.h`. Returns `0` on success, `-1` if error. Note: Char array must be large enough to hold hostname and `NULL` char.<br>`unistd.h` |
| `getlogin()` | `char* getlogin()` — Returns char pointer to current user's username on success, NULL if error. Do not `delete` pointer as string is static. This also means changing the string will change return value of subsequent calls to `getlogin()`.<br>`unistd.h` |

| | |
|---|---|
| Extra syscalls | These are not necessary for cs100 projects. However, they can be useful if you want more information about a process that receives a signal, groups, users, or need to communicate with a terminal for a project. Enjoy! |
| `sigaction()` | `int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact)` — similar to `signal()`, but is more portable and conforms to POSIX. Not needed for CS100 projects. Sets signal handler of signal `signum` to handler specified in members `sa_handler` or `sa_sigaction` of `struct sigaction act` and saves old handler struct to `oldact`. You can leave `act` or `old` `NULL`.<br>`signal.h` |
| `getgrgid()` | `struct group* getgrgid(gid_t gid)` — Returns struct with group information on success, NULL on not finding group id `gid` or error (set `errno` to `0` before syscall, then check `errno` to tell which).<br>`sys/types.h`, `grp.h` |
| `getpwuid()` | `struct passwd* getpwuid(uid_t uid)` — Returns struct with user information associated with `uid` or `NULL` if error occurs or `uid` not found (set `errno` to `0` before syscall, then check `errno` to tell which).<br>`sys/types.h`, `pwd.h` |
| `ioctl()` | `int ioctl(int d, int request, ...)` — Sends request to file descriptor `d`. Used to manipulate devices and terminals. Returns `0` on success usually (some devices use return as output value), `-1` on error. Request codes are different for each device and so no example is listed here. The third parameter is traditionally a `char* argp`. |