CS100 Syscall Cheatsheet

`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. Technically not a syscall. *Include:* `stdio.h`, `errno.h`
`fork()`	`pid_t fork()` — 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. *Include:* `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` 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. *Include:* `unistd.h`
`execvp()`	`int execvp(const char* file, char *const argv[])` — Executes program `path` and passes arguments `argv`. Finds file automatically by checking directories in environmental variable `PATH`. *Include:* `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). *Include:* `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 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). *Include:* `sys/wait.h`
Directories and Files	Note: All functions work on both relative and absolute paths. A relative path does not begin with a slash (e.g. `bin/rshell/main.cpp`). What a relative path points to depends on your current working directory. Absolute paths begin with a slash (e.g. `/bin/rshell/main.cpp`). What an absolute path points to never depends on your current working directory.
`opendir()`	`DIR* opendir(const char* name)` — Opens directory stream to directory `name` and returns pointer to its first entry. Returns `NULL` on error. *Include:* `dirent.h`
`closedir()`	`int closedir(DIR* dirp)` — Closes directory. returns `0` on success, `-1` if error. *Include:* `dirent.h`
`chdir()`	`int chdir(const char* path)` — Change directory of calling process to `path`. Returns `0` on success, `-1` if error. *Include:* `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. *Include:* `sys/types.h`, `sys/stat.h`, `unistd.h`

`open()`	`int open(const char* pathname, int flags)`
	`int open(const char* pathname, int flags, mod_t mode)`
	Opens file and returns file descriptor which can be used, with flags, to read/write/create file.
	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.
	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.
	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`).
	Include: `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). *Include:* `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. **Warning:** be sure to close copies after you finish using them. File descriptors are limited. *Include:* `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 *Include:* `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`. *Include:* `unistd.h`
Useful Signals	`SIGINT` to interrupt (`Ctrl+c`), `SIGTSTP` to temporarily stop (`Ctrl+z`), or `SIGQUIT` to quit (`Ctrl+\`).
`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)` e.g. `signal(SIGINT, myhandler);` *Include:* `signal.h`
`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`. *Include:* `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. *Include:* `sys/types.h`, `signal.h`
`getcwd()`	`char* getcwd(char*buf, size_t size)` — Returns pointer to current working directory string on success and copies to `buf` of size `size` if not `NULL`. Returns `NULL` if error. *Include:* `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. *Include:* `unistd.h`
`getlogin()`	`char* getlogin()` — Returns 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()`. *Include:* `unistd.h`
`getgrgid()`	`struct group* getgrgid(gid_t gid)` — Returns pointer to 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). *Include:* `sys/types.h`, `grp.h`
`getpwuid()`	`struct passwd* getpwuid(uid_t uid)` — Returns pointer to 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). *Include:* `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`. *Include:* `sys/ioctl.h`