

CS100 Syscall Cheatsheet

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

<code>perror()</code>	<code>void perror(const char* s)</code> — Prints <code>s</code> and syscall's error message as defined by global int <code>errno</code> to <code>stderr</code> . If syscall has error, <code>errno</code> is changed. Technically not a syscall. <i>Include: <code>stdio.h, errno.h</code></i>
<code>fork()</code>	<code>pid_t fork()</code> — 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. <i>Include: <code>unistd.h</code></i>
<code>exec</code>	Exec note: If <code>exec</code> succeeds, the current process will end and <code>exec</code> will not return. It returns -1 if it fails (e.g. program file not found). <code>char *const argv</code> must be NULL terminated.
<code>execv()</code>	<code>int execv(const char* path, char *const argv[])</code> — Executes program <code>path</code> and passes arguments <code>argv</code> . Requires full path name of program. <i>Include: <code>unistd.h</code></i>
<code>execvp()</code>	<code>int execvp(const char* file, char *const argv[])</code> — Executes program <code>path</code> and passes arguments <code>argv</code> . Finds file automatically by checking directories in environmental variable <code>PATH</code> . <i>Include: <code>unistd.h</code></i>
<code>wait()</code>	<code>pid_t wait(int* status)</code> — Waits for child process to terminate. <code>int* status</code> stores exit status of child process. Use NULL if not needed. Returns child pid if succeeds, -1 if fails (e.g. no child). <i>Include: <code>sys/wait.h</code></i>
<code>waitpid()</code>	<code>pid_t waitpid(pid_t pid, int* status, int options)</code> — Similar to <code>wait()</code> . Can specify pid to wait for specific child; use 0 for any child. Can also wait for stopped processes by adding option <code>WUNTRACED</code> and check immediately instead of waiting with <code>WNOHANG</code> (Bitwise OR to combine: <code>'WUNTRACED WNOHANG'</code>). Returns child pid if succeeds, -1 if fails (e.g. invalid pid). <i>Include: <code>sys/wait.h</code></i>
Directories and Files	Note: When these functions require a directory or file name, they only require a <i>relative path</i> . This means that if your process was called in directory <code>bin/foo/bar/</code> , instead of using <code>bin/foo/bar/p.cpp</code> , the <i>absolute path</i> , you can use <code>p.cpp</code> or <code>./p.cpp</code> instead.
<code>opendir()</code>	<code>DIR* opendir(const char* name)</code> — Opens directory stream to directory <code>name</code> and returns pointer to its first entry. Returns NULL on error. <i>Include: <code>dirent.h</code></i>
<code>closedir()</code>	<code>int closedir(DIR* dirp)</code> — Closes directory. returns 0 on success, -1 if error. <i>Include: <code>dirent.h</code></i>
<code>chdir()</code>	<code>int chdir(const char* path)</code> — Change directory of calling process to <code>path</code> . Returns 0 on success, -1 if error. <i>Include: <code>sys/stat.h, unistd.h</code></i>
<code>stat()</code>	<code>int stat(const char* path, struct stat* buf)</code> — Gives information about a file in <code>struct stat</code> , e.g. permissions, type of file, time created. See <code>ls -l</code> for example of provided information. Macros are also provided that take <code>mode_t st_mode</code> in <code>struct stat</code> and returns <code>true/false</code> (e.g. <code>S_ISDIR(st_mode)</code> , <code>S_REG(st_mode)</code>). Returns 0 on success, -1 if error. <i>Include: <code>sys/types.h, sys/stat.h, unistd.h</code></i>
<code>open()</code>	<code>int open(const char* pathname, int flags)</code> <code>int open(const char* pathname, int flags, mod_t mode)</code> Opens file and returns file descriptor which can be used, with flags, to read/write/create file. Flags: Must use either <code>O_RDONLY</code> (read file), <code>O_WRONLY</code> (write to file), or <code>O_RDWR</code> (both) in call. These can be bitwise OR'd (<code>'O_WRONLY O_CREAT'</code>) with other flags such as: <code>O_CREAT</code> creates file if it doesn't exist. <code>O_TRUNC</code> overwrites contents of file when writing, <code>O_APPEND</code> writes at the end of file instead. Modes: When creating files, <code>mode</code> arguments can be added to specify permissions of new file, such as <code>S_IRWXU</code> - user has read/write/execute permission, <code>S_IRUSR</code> - user has read permission, or <code>S_IWUSR</code> - user has read permission. Warning: Every call to <code>open()</code> must have a corresponding <code>close()</code> or else file descriptors will be left open (similar to memory leaks with <code>new</code> and <code>delete</code>). <i>Include: <code>sys/types.h, sys/stat.h, fcntl.h</code></i>
<code>close()</code>	<code>int close(int fd)</code> — Close file descriptor. Returns 0 on success, -1 if error (e.g. invalid fd). <i>Include: <code>unistd.h</code></i>

<code>dup()</code>	<p><code>int dup(int oldfd)</code> — Copies file descriptor <code>oldfd</code> to the next lowest unused descriptor, returns new file descriptor on success, <code>-1</code> if error.</p> <p>Warning: be sure to close copies after you finish using them. File descriptors are limited.</p> <p><i>Include:</i> <code>unistd.h</code></p>
<code>dup2()</code>	<p><code>int dup2(int oldfd, int newfd)</code> — Copies file descriptor <code>oldfd</code> in <code>newfd</code>. Closes <code>newfd</code> if it already exists. Returns new file descriptor on success, <code>-1</code> if error</p> <p><i>Include:</i> <code>unistd.h</code></p>
<code>pipe()</code>	<p><code>int pipe(int pipefd[2])</code> — Returns two file descriptors in <code>pipefd</code> for reading from and writing to. The file descriptors are one way pipes only, hence the need for two. Data written to <code>pipefd[1]</code> can be read from <code>pipefd[0]</code>. <code>pipe</code> is usually associated with piping in bash.</p> <p>e.g. <code>ls grep .cpp</code>.</p> <p><i>Include:</i> <code>unistd.h</code></p>
Useful Signals	<p><code>SIGINT</code> to interrupt (<code>Ctrl+c</code>), <code>SIGTSTP</code> to temporarily stop (<code>Ctrl+z</code>), or <code>SIGQUIT</code> to quit (<code>Ctrl+\</code>).</p>
<code>signal()</code>	<p><code>sighandler_t signal(int signum, sighandler_t handler)</code> — Sets signal handler for signal <code>signum</code> to function <code>handler</code>. For signal handlers: <code>SIG_IGN</code> ignores signal and <code>SIG_DFL</code> uses default handler. You can also use a user-defined handler as an argument. <code>sighandler_t</code> is a pointer to a void function that has an <code>int</code> parameter: <code>void myhandler(int sig)</code></p> <p>e.g. <code>signal(SIGINT, myhandler);</code></p> <p><i>Include:</i> <code>signal.h</code></p>
<code>sigaction()</code>	<p><code>int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact)</code></p> <p>Similar to <code>signal()</code>, but is more portable and conforms to POSIX. Not needed for CS100 projects. Sets signal handler of signal <code>signum</code> to handler specified in members <code>sa_handler</code> or <code>sa_sigaction</code> of <code>struct sigaction act</code> and saves old handler struct to <code>oldact</code>. You can leave <code>act</code> or <code>old</code> <code>NULL</code>.</p> <p><i>Include:</i> <code>signal.h</code></p>
<code>kill()</code>	<p><code>int kill(pid_t pid, int sig)</code> — Sends any signal to process or process group. Returns <code>0</code> on success, <code>-1</code> if error.</p> <p><i>Include:</i> <code>sys/types.h</code>, <code>signal.h</code></p>
<code>getcwd()</code>	<p><code>char* getcwd(char*buf, size_t size)</code> — Returns pointer to current working directory string on success and copies to <code>buf</code> of size <code>size</code> if not <code>NULL</code>. Returns <code>NULL</code> if error.</p> <p><i>Include:</i> <code>unistd.h</code></p>
<code>gethostname()</code>	<p><code>int gethostname(char* name, size_t len)</code> — Writes hostname (<code>hammer.cs.ucr.edu</code>) to char array <code>name</code> with size <code>len</code>. According to the man page, the guaranteed maximum value for <code>len</code> is <code>HOST_NAME_MAX = 64</code>, which is in <code>limits.h</code>. Returns <code>0</code> on success, <code>-1</code> if error.</p> <p>Note: <code>char</code> array must be large enough to hold hostname and <code>NULL</code> char.</p> <p><i>Include:</i> <code>unistd.h</code></p>
<code>getlogin()</code>	<p><code>char* getlogin()</code> — Returns pointer to current user's username on success, <code>NULL</code> if error. Do not delete pointer as string is static. This also means changing the string will change return value of subsequent calls to <code>getlogin()</code>.</p> <p><i>Include:</i> <code>unistd.h</code></p>
<code>getgrgid()</code>	<p><code>struct group* getgrgid(gid_t gid)</code> — Returns pointer to struct with group information on success, <code>NULL</code> on not finding group id <code>gid</code> or error (set <code>errno</code> to <code>0</code> before syscall, then check <code>errno</code> to tell which).</p> <p><i>Include:</i> <code>sys/types.h</code>, <code>grp.h</code></p>
<code>getpwuid()</code>	<p><code>struct passwd* getpwuid(uid_t uid)</code> — Returns pointer to struct with user information associated with <code>uid</code> or <code>NULL</code> if error occurs or <code>uid</code> not found (set <code>errno</code> to <code>0</code> before syscall, then check <code>errno</code> to tell which).</p> <p><i>Include:</i> <code>sys/types.h</code>, <code>pwd.h</code></p>
<code>ioctl()</code>	<p><code>int ioctl(int d, int request, ...)</code> — Sends request to file descriptor <code>d</code>. Used to manipulate devices and terminals. Returns <code>0</code> on success usually (some devices use return as output value), <code>-1</code> on error. Request codes are different for each device and so no example is listed here. The third parameter is traditionally a <code>char* argp</code>.</p> <p><i>Include:</i> <code>sys/ioctl.h</code></p>