Useful C/Linux Functions in CSC 374 Computer Systems 2

Last modified 2014 June 9

C-string related functions

Function	Purpose
`char* fgets(char* charArray, int size,stdin)`	Reads up to `size-1` characters from `stdin` and places them in `charArray`. Stops reading upon end-of-line ('\n') or end-of-file. Stores '\0' to end string. Returns `charArray` on success or `NULL` on failure.
`int snprintf(char* charArray, size_t size, const char* format, . . .)`	Prints up to `size` bytes to `charArray` (including the ending '\0') that are the formated printing of the further arguments into `format`. Returns number of characters written into `charArray`.
`char* strncpy (char* dest, const char* source, size_t size)`	Copies at most `size` characters from `source` into `dest`. (Warning: If there is no '\0' among the first `size` bytes of `source`, the string placed in `dest` will not be null-terminated.) Returns `dest`.
`char* strncat (char* dest, const char* source, size_t size)`	Appends the characters from `source` to the end of `dest`, but not letting `dest` be more than `size` chars long total. The resulting string in dest is always '\0'-terminated. Returns `dest`.
`char* strncmp (const char* s1, const char* s2, size_t size)`	Compares the first `size` chars of `s1` with `s2`. It returns an integer less than, equal to, or greater than zero if `s1` is found, respectively, to be less than, to match, or be greater than `s2`.
`size_t strnlen (const char* s, size_t size)`	Returns the length of string `s`, or `size`, which ever is shorter
`char* strndup (const char* s, size_t size)`	Returns a pointer to the first `size` bytes of `s` allocated from the heap. Ending '\0' is added if `s` is longer than `size`.
`int strtol(const char* s,char** ptrPtr, int base)`	Returns the integer that is written as a base `base` number in `s`. For example, `strtol("-12",NULL,10) == -12`. If `base` == `0` then the rules used by the C compiler will be used:

0x40	Hexadecimal 40 (= 64 decimal)
040	Octal 40 (= 32 decimal)
40	Decimal 40 (= 40 decimal)

`double strtod(const char* s, char** ptrPtr)`	Returns the double floating point number that is written as a decimal number in `s`. For example, `strtod("-1.2",NULL) == -1.2`

Process-related functions

Function	Purpose
`pid_t getpid()`	Returns the process id of the process running this.
`pid_t getppid()`	Returns the process id of the *parent* of the process running this.
`int fork()`	Attempts to make a child process. Return value is either: • Negative: no child process made (process table full?) • 0: The process that receives 0 *is* the child process • Positive: The process that receives a positive number is the parent process. The actual number is the process id of the child.

`void execl`
`(const char* progName,`
`const char* progName,`
`const char* arg1,`
`. . .`
`const char* argN,`
`NULL // `**`VERY IMPORTANT`**
`);`
	Stop running the current program and attempt to run the program named `progName`. **NOTE:**
	• `progName` is given *twice*:
	◦ The first time is for the OS: so it knows the program to run
	◦ The second time is for the process: so it knows the program it is running.
	• `NULL` **must** be the last argument.
	One of two things will happen:
	• *If you can run `progName`:* The process will forget about the old program and start running the new one. When it

does:

  - `argc == N+1`
  - `argv[0]` will point to the text of `progName`
  - `argv[1]` will point to the text of `arg1`

    . . .
  - `argv[N]` will point to the text of `argN`

NOTE: even when there are no extra arguments after `progName` (called like `execl(progName,progName,NULL)`) then `argc` will be at least `1`.

- *If you can **not** run `progName`:* The process will do the line *after* the `execl()`. Therefore, it is common to have an `fprintf()` and `exit(EXIT_FAILURE);` after an `execl()` call.

```
int kill (int pid, int signalNum)
```

Sends signal `signalNum` to process `pid`. Don't worry about the return number.

```
struct sigaction        action;

sigemptyset(&action.sa_mask);
action.sa_flags   = 0; // See notes
action.sa_handler = simpleHandler;

sigaction(int signalNum,&action,NULL)
```

Tells the OS that when this process receives signal `signalNum` it is to do function `simpleHandler`. `simpleHandler` should have form:

```
void simpleHandler (int sigNum)
```

`simpleHandler` can also be:

Value	**Meaning**
`SIG_IGN`	"Ignore this signal"
`SIG_DFL`	"Do the default action for this signal"

Useful signals include:

Name	**Default Action**	**Description**
`SIGINT`	terminate process	Ctrl-C interrupt
`SIGKILL`	terminate process	Unblockable interrupt
`SIGUSR1`	terminate process	User defined signal 1
`SIGUSR2`	terminate process	User defined signal 2
`SIGALRM`	terminate process	Alarm clock
`SIGCHLD`	Ignore	Child process finished

<table>
<tr>
<td></td>
<td>Useful flags include:
<table>
<tr><th>Flag</th><th>Meaning</th></tr>
<tr><td><code>SA_NOCLDSTOP</code></td><td>(For <code>SIGCHLD</code>) only do the child handler when the child ends (not when it pauses)</td></tr>
<tr><td><code>SA_RESTART</code></td><td>If the signal comes when you are in the middle of a system call, then restart the system call (as opposed to quitting) when the handler finishes.</td></tr>
</table>
For a more comprehensive table see
http://www.manpagez.com/man/3/Signal/
(This is for BSD, slightly different than Linux.)</td>
</tr>
<tr>
<td><pre>struct sigaction   action;

sigemptyset(&action.sa_mask);
action.sa_flags   = SA_SIGINFO;
// Need SA_SIGINFO to specify advancedHandler
// (the other flags are optional)
action.sa_sigaction = advancedHandler;

sigaction(int signalNum,&action,NULL)</pre></td>
<td>Tells the OS that when this process receives signal <code>signalNum</code> it is to do function <code>advancedHandler</code>. <code>advancedHandler</code> should have form:

<code>void advancedHandler (int sigNum, siginfo_t* infoPtr, void* dataPtr)</code>

<code>infoPtr</code> gives all kinds of info. Perhaps among the most useful is <code>infoPtr->si_pid</code> which tells the process id of who sent the signal (or maybe <code>0</code> if coming from the OS or hardware).

<code>dataPtr</code> is not used so much.

See above for the descriptions of the <code>signalNum</code> and flags.</td>
</tr>
<tr>
<td><code>pid_t wait(int* ptr)</code></td>
<td>If this process has at least one child process still running then waits for it to finish. When it finally does finish (or if one had already finished) then sets <code>*ptr</code> equal to the status returned by the child and returns the process id of the child.

If child ended normally then <code>WIFEXITED(childStatus)</code> return non-zero. If the child crashed then <code>WIFEXITED(childStatus) == 0</code>.

If the child end normally then the portion of the status that was <code>return()</code>ed by child's <code>main()</code>, or which the child <code>exit()</code>ed, is obtained by <code>WEXITSTATUS(childStatus)</code></td>
</tr>
</table>

<table>
<tr><td></td><td>If there are no children for which to wait() then return 0.</td></tr>
<tr><td>pid_t waitpid(pid_t pid, int* statusPtr, int options)</td><td>Like wait() but can wait for specific child with process id pid (or any child if pid == -1) The most important options for options are:
<table>
<tr><td>Value</td><td>Meaning</td></tr>
<tr><td>0</td><td>Act just like wait()</td></tr>
<tr><td>WNOHANG</td><td>Return immediately if no child has exited</td></tr>
</table></td></tr>
<tr><td>void exit(int status)</td><td>Ends the program and return status to the OS. The value of status can be obtained the parent of the quiting program with the expression WEXITSTATUS(status), where the parent's status variable was set by wait(&status)</td></tr>
</table>

Threading-related functions

Be sure to:

1. #include <pthread.h>
2. Compile/link with *-lpthread* on the command line

Function	What it does
int pthread_create (/* Pointer to a pthread_t object */ pthread_t* restrict threadPtr, /* Pointer to optional object for properties of child */ const pthread_attr_t* restrict attr, /* Name of function to run: void* fncName(void* ptr) */ void *(*fncName)(void*), /* Ptr to object that is parameter to fncName() */ void *restrict arg)	Makes a thread in the space pointed to by threadPtr The thread run the function void* fncName(void*) and passes arg to it. Just leave attr as NULL for a generic thread.
int pthread_join (/* Which thread to wait for */ pthread_t thread,	Waits for thread thread to finish. When it does valuePtr (the thing that valuePtrsPtr points to) is set to the thread's

`/* Pointer to pointer to receive pointer`
`returned by exiting thread's function.`
`*/`
`void**                 valuePtrsPtr`
`)`
	function's returned pointer value **or** it is ignored if `valuePtr==NULL`

`int pthread_mutex_init`
`(/* Ptr to space for mutex */`
`pthread_mutex_t *restrict mutexPtr,`
`/* Type of mutex (just pass NULL) */`
`const pthread_mutexattr_t *restrict attr`
`);`
	Initializes lock object pointed to by `mutexPtr`. Just use NULL for 2nd parameter.

`int pthread_mutex_destroy`
`(/* Ptr to mutex to destroy *.`
`pthread_mutex_t *mutex`
`);`
	Releases resources taken by mutex pointed to by `mutexPtr`.

`int pthread_mutex_lock`
`(/* Pointer to mutex to lock */`
`pthread_mutex_t *mutexPtr`
`);`
	Either
	1. Gains lock and proceeds, or
	2. Waits for lock to become available

`int pthread_mutex_unlock`
`(/* Pointer to mutex to unlock */`
`pthread_mutex_t *mutexPtr`
`);`
	Releases lock.

`int pthread_cond_init`
`(/* Pointer to space in which to make condition */`
`pthread_cond_t *restrict condPtr,`
`/* Type of condition (just pass NULL) */`
`const pthread_condattr_t *restrict attr`
`);`
	Creates a condition.

`int pthread_cond_destroy`
`(/* Pointer to condition to destroy */`
`pthread_cond_t *condPtr`
`);`
	Destroys pointed to condition.

`int pthread_cond_wait`
`(/* Pointer to condition on which to wait */`
` pthread_cond_t *restrict condPtr,`
 	
` /* Pointer to mutex to surrender until receive signal */`
` pthread_mutex_t *restrict mutexPtr`
`);`
	Suspends thread until receives signal on `condPtr`. While thread is suspended it surrenders lock on `mutexPtr`

`int pthread_cond_signal`
`(/* Ptr to condition which is signaled */`
` pthread_cond_t *condPtr`
`);`
	Wakes up at least one thread waiting for signal on `condPtr`.

Directory reading related functions

Be sure to:

1. #include <sys/types.h> // For opendir()
2. #include <dirent.h> // For opendir()

Function	Purpose
DIR* opendir(const char* name)	To open return a DIR pointer that allows programmer to read each entry in the directory named `name`, or `NULL` on error.

struct dirent *readdir(DIR *dirp)	Return a pointer to the next directory entry in the opened directory pointed to by `dirp`. Returns `NULL` on no more entries or error.
	
	Fields of `struct dirent` include:
	
	`struct dirent`
	`{`
	`  ino_t          d_ino;       // Inode number`
	`  off_t          d_off;       // Offset to the next dirent`
	`  unsigned short d_reclen;    // Length of this record`
	`  unsigned char  d_type;      // Type of file; not supported`
	`                              // by all file system types`
	`  char           d_name[256]; // Filename`
	`};`

int closedir(DIR* dirp)	To close the directory pointed to by `dirp`.

Higher level file I/O-related functions

Be sure to:

1. #include <stdio.h>

FILE *fopen(const char *path, const char *mode);	Return a pointer of type `FILE*` that represents the openning of file `path` by mode `mode`. Returns `NULL` if could not open file. Common modes include: "r" Reading from beginning "w" Writing (or overwriting existing files) "a" Appending (or creating non-existing files)
int fclose(FILE *fp)	To close the file pointed to by `fp`.
int fflush(FILE *fp)	To ask the OS to really send the bytes written to file `fp` to the harddrive/screen/etc. instead of keeping them buffered in memory.
int fprintf(FILE* fp, const char* format, ...)	To do formatted (`printf()`-style) printing to file `fp` given format string `format` and arguments in `...`. Like `printf()`, returns the number of chars printed (or `-1` on error).
char *fgets(char *s, int size, FILE *stream)	Attempt to read up to either on line or `size` bytes from `stream` and place into `s`. Returns `s` on success or `NULL` on end-of-file (EOF) or error.

File information getting-related functions

Be sure to:

1. #include <sys/types.h>
2. #include <sys/stat.h>
3. #include <unistd.h>

```
struct stat  statBuffer;

int stat(const char *path, &statBuffer)
```

To attempt to write into `buf` information on directory entry `path`. Returns 0 on success or -1 otherwise.

The info that is written is:

```
struct stat
{
  dev_t     st_dev;     // ID of device containing file
  ino_t     st_ino;     // Inode number
  mode_t    st_mode;    // Type of entry
  nlink_t   st_nlink;   // Number of hard links
  uid_t     st_uid;     // User ID of owner
  gid_t     st_gid;     // Group ID of owner
  dev_t     st_rdev;    // Device ID (if special file)
  off_t     st_size;    // Total size, in bytes
  blksize_t st_blksize; // Blocksize for file system I/O
  blkcnt_t  st_blocks;  // Number of 512B blocks allocated
  time_t    st_atime;   // Time of last access
  time_t    st_mtime;   // Time of last modification
  time_t    st_ctime;   // Time of last status change
};
```

Among the most useful of these is `buf.st_mode` that can tell you what type of entry `path` is:

S_ISREG(buf.st_mode)	Is it a regular file?
S_ISDIR(buf.st_mode)	Is it a directory?
(There are others, but those are to two most important.)	

Socket and low-level file I/O-related functions

Be sure to:

1. #include <unistd.h> // For sleep()
2. #include <sys/socket.h> // For socket()
3. #include <netinet/in.h> // For sockaddr_in and htons()
4. #include <netdb.h> // For gethostbyname()
5. #include <errno.h> // For errno var

How to:	Usage:

Get a file descriptor for a socket	`int socket(AF_INET,int protocol,int type)` Returns a file descriptor for the socket, or -1 on error. Protocol protocol type TCP SOCK_STREAM 0 UDP SOCK_DGRAM 0
Tell server max. number of waiting clients	`int listen(int serverSocketFD, int maxNumWaitingClients)` Tells OS that the server socket file descriptor `serverSocketFD` should have a maximum of `maxNumWaitingClients` clients waiting to connect. Returns -1 on error.
Have server wait until a client connects	`int accept(int socketFD,NULL,NULL)` `socketFD` tells the file descriptor of the socket on which to wait. Returns new file descriptor for communicating with the connected client, or -1 on error.
Close file, socket, *etc*.	`int close(int fileD)` Closes file descriptor `fileD`. Returns -1 on error.
Send bytes	`int write(int fileDes,const void* bufferPtr, int numBytes)` Writes *numBytes* bytes pointed to by *bufferPtr* to file descriptor *fileDes*. Returns number of bytes written (0 means "none"), or -1 which means "error".
Read bytes (I)	`int read(int fileDes,void* bufferPtr, int bufferLen)` Reads up to *bufferLen* bytes into the buffer pointed to by *bufferPtr* from file descriptor *fileDes*. Waits until something is available. Returns number of bytes read, or returns -1 on error.
Read bytes (II)	`int recv(int fileDes,void* bufferPtr, int bufferLen, int flags)` Reads up to *bufferLen* bytes into the buffer pointed to by *bufferPtr* from file descriptor *fileDes*. *flags* tells how to read, where *MSG_DONTWAIT* means "non-blocking".

	Returns number of bytes read, or returns -1 and sets `errno` to `EAGAIN` if the flag was *MSG_DONTWAIT* and there was nothing to read.
Convert a 32-bit integer from network's endian to host's endian	`uint32_t ntohl(uint32_t networkInt)` Returns 32-bit integer *networkInt* so that it is in the endian of the current computer instead of for the network.
Convert a 16-bit integer from network's endian to host's endian	`uint16_t ntohs(uint16_t networkInt)` Returns 16-bit integer *networkInt* so that it is in the endian of the current computer instead of for the network.
Convert a 32-bit integer from host's endian to network's endian	`uint32_t htonl(uint32_t hostInt)` Returns 32-bit integer *hostInt* so that it is in the endian of the network instead of for the current computer.
Convert a 16-bit integer from host's endian to network's endian	`uint16_t htons(uint16_t hostInt)` Returns 16-bit integer *hostInt* so that it is in the endian of the network instead of for the current computer.

ncurses package-related functions

Be sure to:

1. #include <curses.h>
2. Compile/link with *-lncurses* on the command line

How to:	Usage:
Start ncurses	`initscr()`
Stop ncurses	`endwin()`

Return a pointer to a new window	`WINDOW* newwin(int numRows,`
	`               int numCols,`
	`               int topRowNum,`
	`               int leftMostColNum`
	`              );`
	`// Top left is position (0,0)`
	`// Pre-made window 'stdscr' refers to whole screen`

Destroys a window	`delwin(WINDOW* window)`
Clear the screen	`clear()`
Clear window *win*	`wclear(WINDOW* win)`
Refresh the whole screen	`refresh()`
Refresh window *win*	`wrefresh(WINDOW* win)`
Turn off line buffering	`cbreak()`
Turn off echoing of typed chars	`noecho()`
Make `getch()` "non-blocking", meaning it just sees if a key was already pressed and either returns that key if there is one or returns `ERR` if not. It does not wait at all for a key. (By default `getch()` waits for the user to press a key, or if `halfdelay()` has been called it waits for a specified amount of time for a key.)	`nodelay(stdscr,TRUE)`
Make `getch()` quit and return `ERR` if no key has been pressed after `tenths` tenths of a second. `getch()` will either return a key if one has been pressed within the given time, or return `ERR` after `tenths` tenths of a second if no key has been pressed. (By default `getch()` waits for the user to press a key, or if `nodelay()` has been called it just sees if a key was pressed and does not wait at all.)	`halfdelay(int tenths)`
Allow usage of keypad chars	`keypad (stdscr,TRUE)`
Disallow scrolling	`scrollok(windowPtr, FALSE)`
Move the cursor on the whole screen	`move(int row, int col)` Moves the cursor to row *row*, column *col* within the whole screen. 0,0 is the upper left corner.
Move the cursor within a given window	`wmove(WINDOW* wPtr, int row, int col)` Moves the cursor to row `row`, column `col` within window `*wPtr`. 0,0 is the upper left corner.
Write a char to the whole screen	`addch(chtype character)` Writes character *character* to the current cursor position.

Write a char to a particular window	`waddch(WINDOW* win, chtype character)` Writes character *character* to the current cursor position in *win*
Write a char to a particular position	`mvaddch(int y, int x, chtype character)` Writes character *character* to cursor position row *y* column *x*
Write a char to a particular position of a particular window	`mvwaddch(WINDOW* win, int y, int x, chtype character)` Writes character *character* to cursor position row *y* column *x* in window *win*.
Write a string to the whole screen	`addstr(const char* toPrintPtr)` Writes the C-string pointed to by *toPrintPtr* to the current cursor position
Write a string to a particular window	`waddstr(WINDOW* win, const char* toPrintPtr)` Writes the C-string pointed to by *toPrintPtr* to the current cursor position of *win*.
Write a string to a particular position of the whole screen	`mvaddstr(int y, int x, const char* toPrintPtr)` Writes the C-string pointed to by *toPrintPtr* to cursor position row *y* column *x*.
Write a string to a particular position of a particular window	`mvwaddstr(WINDOW* win, int y, int x, const char* toPrintPtr)` Writes the C-string pointed to by *toPrintPtr* to the cursor position row *y* column *x* of *win*.
Get a character from the keyboard	`int getch()`