CSC 374/407: Computer Systems II: Final (2017 Summer II)

Joe Phillips
Last modified 2017 August 16

Name: _____

Distance Learning Students Only!

If you want your graded final returned to you please write your address below:

_____

_____

_____

4 points free, then 16 points per question

1. **Optimization and Compilers**

   There are at least 4 optimizations that can be made in `optimizeMe()`. Find four optimization and for each:

   a. *do* it,
   b. tell whether the *compiler* or *programmer* should make it,
   c. tell *why* either the compiler or programmer (as opposed to the other) should make it

```
//  PURPOSE:  To return some integer value computed from 'arg0' and 'arg1'.
extern
int             someFunction    (int    arg0,
                                 int    arg1
                                );
                                //  I will spare you the irrelevant details


//  PURPOSE:  To harass Computer Systems II students.  Computes some arbitrary
//      function of 'intArrayLen' and 'intArray' that I pulled out of my a**.
//      Returns its value.
int     optimizeMe              (int    intArrayLen,
                                 int*   intArray
                                )
{
```

```
  int   ind;
  int   hiInd   = 0;

  for  (ind = 1;  ind < intArrayLen/2;  ind = ind * 2)
  {
    int value   = someFunction(intArray[ind],intArray[intArrayLen-ind-1]);

    if  (value > someFunction(intArray[hiInd],intArray[intArrayLen-hiInd-1]))
      hiInd     = ind;

  }

  return(someFunction(intArray[hiInd],intArray[intArrayLen-hiInd-1]));
}
```

Num	Optimitization (just do above)	Compiler or Programmer?	Why done by the person (or program) you said?
(i)			
(ii)			
(iii)			
(iv)			

2. **Memory**

Consider a process running the following program:

```
#include        <stdlib.h>
#include        <stdio.h>

int             globalInt       = 10;

int     main            ()
{
  int   i;
  for  (i = globalInt;  i != 0;  i--)
    printf("i = %d\n",i);

  printf("All done!\n");
  return(EXIT_SUCCESS);
}
```

Please tell where the following objects are stored in memory.

Your choices are:

a. ROM BIOS
b. kernal Memory
c. shared library memory
d. .text segment
e. .rodata segment
f. .data segment
g. .bss segment
h. the heap
i. the stack

Where is:

1. (4 Points) the memory for variable 'i'?
2. (4 Points) the memory for variable 'globalInt'?
3. (4 Points) the for loop?
4. (4 Points) the code that turns the string `"i = %d\n"` to `"i = 6\n"`

3. **Processes, Exceptions and Signals**

a. (4 Points) A parent process `fork()`s a child process to compute a boolean result. The child process computes the result, and wants to send one of two values (either `0` or `1`) back to the parent. Can this be done with signals?
If so, how?
If not, why not?

b. (4 Points) A parent process `fork()`s a child process to compute a complicated result. The child process computes the result, and wants to send an object (e.g. C `struct` instance or C++ `class` instance) back to the parent. Can this be done with signals?
If so, how?
If not, why not?

c. (4 Points) Let us say you write a program to measure how quick a person's fingers are by trapping `SIGINT` and then asking them to press `Ctrl-C` as rapidly as possible. The `SIGINT` signal handler increments a global counter every time `Ctrl-C` is typed. After a predefined time it stops and prints the global counter divided by the time used.
What is a fundamental problem with this program?

d. (4 Points) Why is it important to have a `SIGCHLD` handler for most parent processes that `fork()`s child processes?
What should the `SIGCHLD` handler do?

4. **Threads**

The program below makes two child threads: a guessing thread and an answering thread. It uses 3 global integers, 1 `pthread_mutex_t`, and 2 `pthread_cond_t`

- `turn`: Tells whose turn it is, either `ANSWERER_TURN` or `GUESSER_TURN`.
- `guess`: Holds the most recent guess generated by the guessing thread.
- `shouldContinue`: Holds `1` while the program should continue (the guesser does not have the correct number). Holds `0` after the guesser guesses the correct number.
- `lock`: a `pthread_mutex_t` variable.
- `guessersTurn`: a `pthread_cond_t` variable.
- `answerersTurn`: a `pthread_cond_t` variable.

Output:

```
$ ./guesser
(Don't tell, but the answer is 7)
Is the answer 6?
Sorry, the answer is not 6
Is the answer 9?
Sorry, the answer is not 9
Is the answer 19?
Sorry, the answer is not 19
Is the answer 17?
Sorry, the answer is not 17
Is the answer 31?
Sorry, the answer is not 31
Is the answer 10?
Sorry, the answer is not 10
Is the answer 12?
Sorry, the answer is not 12
Is the answer 9?
Sorry, the answer is not 9
Is the answer 13?
Sorry, the answer is not 13
Is the answer 26?
Sorry, the answer is not 26
Is the answer 11?
Sorry, the answer is not 11
Is the answer 18?
Sorry, the answer is not 18
Is the answer 27?
Sorry, the answer is not 27
Is the answer 3?
Sorry, the answer is not 3
Is the answer 6?
Sorry, the answer is not 6
Is the answer 28?
Sorry, the answer is not 28
```

```
Is the answer 2?
Sorry, the answer is not 2
Is the answer 20?
Sorry, the answer is not 20
Is the answer 24?
Sorry, the answer is not 24
Is the answer 27?
Sorry, the answer is not 27
Is the answer 8?
Sorry, the answer is not 8
Is the answer 7?
Congratulations!  The answer is 7
```

Please finish the program.

- One child thread should run `answerer()`, the other should run `guesser()`.
- The two threads should take turns: the guessing thread should make a guess and then the answering thread should compare it with `answer` (a local variable that only it has).
- If the guesser got the wrong number, then the guesser should go again. *Etc.*

I have finished `main()`, which has initializes all objects. Please finish `answerer()` and `guesser()`. The `vPtr` argument is `NULL` and may be ignored. However, please figure out:

- What needs to be protected.
- Where should the locks and unlock go?.
- Where should conditions and signals go.

```
/*
 *      guesser.c       Joseph Phillips
 */

#include        <stdlib.h>
#include        <stdio.h>
#include        <pthread.h>

#define         ANSWERER_TURN   0
#define         GUESSER_TURN    1

int             turn            = GUESSER_TURN;
int             guess           = -1;
int             shouldContinue  = 1;
pthread_mutex_t lock;
pthread_cond_t  guessersTurn;
pthread_cond_t  answerersTurn;


void*           answerer        (void*  vPtr
                                )
```

```
{
  int   answer  = rand() % 32;

  printf("(Don't tell, but the answer is %d)\n",answer);

  while  (1)
  {
    (a) YOUR CODE HERE

    while  (turn != ANSWERER_TURN)
    {
      (b) YOUR CODE HERE
    }

    if  (guess == answer)
    {
      printf("Congratulations!  The answer is %d\n",answer);
      shouldContinue    = 0;
      turn              = GUESSER_TURN;
      (c) YOUR CODE HERE
      (d) YOUR CODE HERE
      break;
    }
    else
      printf("Sorry, the answer is not %d\n",guess);

    turn        = GUESSER_TURN;
    (e) YOUR CODE HERE
    (f) YOUR CODE HERE
  }

  return(NULL);
}


void*           guesser         (void*  vPtr
                                )
{
  while  (1)
  {
    (g) YOUR CODE HERE

    while  (turn != GUESSER_TURN)
    {
      (h) YOUR CODE HERE
    }

    if  ( !shouldContinue )
      break;

    guess       = rand() % 32;
    printf("Is the answer %d?\n",guess);
    turn        = ANSWERER_TURN;

    (i) YOUR CODE HERE
```

```
    (j) YOUR CODE HERE
  }

  return(NULL);
}


int             main            ()
{
  pthread_t     answererId;
  pthread_t     guesserId;

  pthread_mutex_init(&lock,NULL);
  pthread_cond_init(&answerersTurn,NULL);
  pthread_cond_init(&guessersTurn,NULL);

  pthread_create(&answererId,NULL,answerer,NULL);
  pthread_create(&guesserId, NULL,guesser, NULL);

  pthread_join(answererId,NULL);
  pthread_join(guesserId,NULL);

  pthread_mutex_destroy(&lock);
  pthread_cond_destroy(&answerersTurn);
  pthread_cond_destroy(&guessersTurn);
}
```

5. **Practical C Programming**

   a. (4 Points) Why should we use `snprintf()` instead of `sprintf()`, `strncpy()` instead of `strcpy()`, *etc.*? Seriously, how bad can using `sprintf()`, `strcpy()`, *etc.* be?

   b. (4 Points) What does `extern` mean?
   What does it tell the compiler to do?

   c. (8 Points) The program below will compile well but run poorly. Please make it *do error checking* and fix it to make it proper:

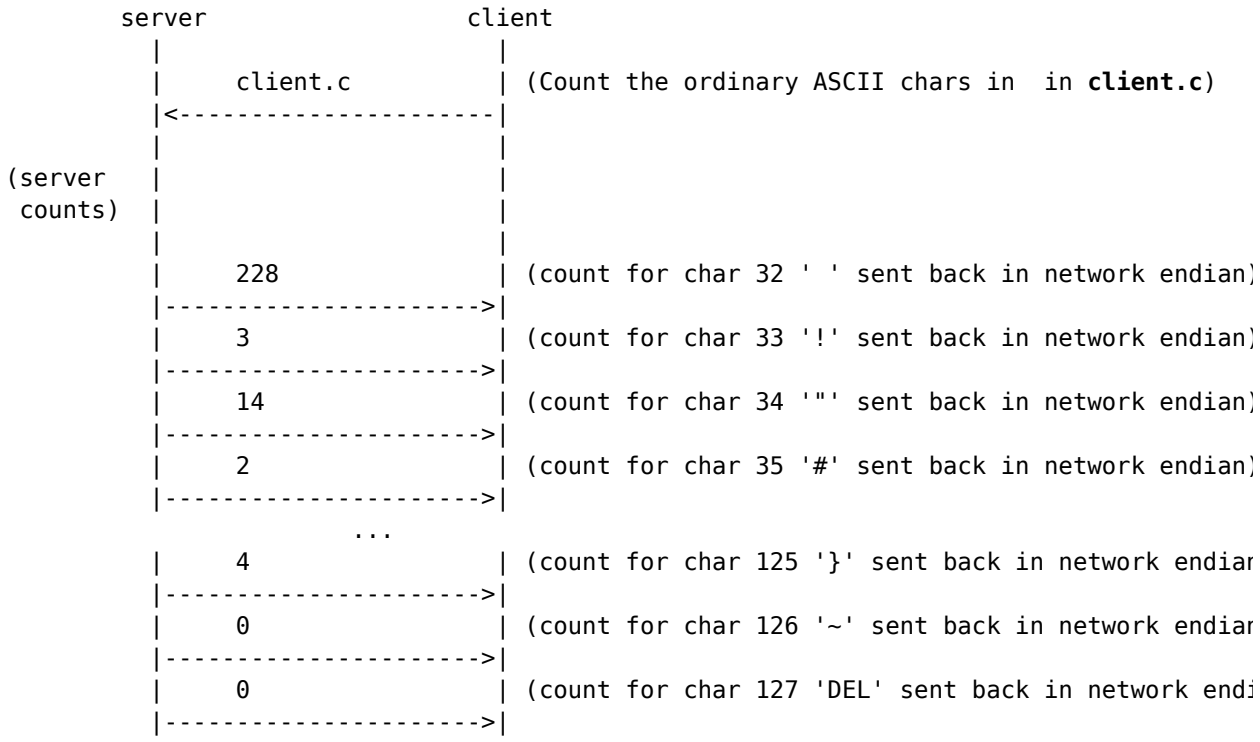
```
#include        <stdlib.h>
#include        <stdio.h>
#include        <string.h>

const int       LINE_LEN        = 1024;

int     main    (int    argc,
                 char*  argv[]
                )
{
  const char*   filename        = argv[1];
  const char*   limitNumText    = argv[2];
  FILE*         fp              = fopen(filename,"r");
  int           limit           = strtol(limitNumText,NULL,10);
  int           haveReachedEnd  = 0;
  char*         line;
  int           counter;

  while  (1)
  {
    for  (counter = 0;  counter < limit;  counter++)
    {
      if  (fgets(line,LINE_LEN,fp) == NULL)
      {
        haveReachedEnd  = 1;
        break;
      }

      printf(line);
    }

    if  (haveReachedEnd)
      break;

    printf("Press enter to see the next %d lines:",limit);
    gets(line);
  }

  return(EXIT_SUCCESS);
}
```

6. **Sockets and Files**

Finish the server function below which counts how many times ordinary ASCII characters (with values from 32 to 127) occur in a given file. The server should do one `read()` from the client into a buffer to get a filename from the client.

Protocol:

(Character ' ' appears in `client.c` 228 times. '**!**' appears 3 times. '**"**' appears 14 times. *etc.*)



Sample Output:

```
$ ./client
Machine name (e.g. localhost)? localhost
Port number? 2000
Please enter a filename: client.c
 :      228
!:      3
":      14
#:      2
...
}:      4
~:      0
^?:     0
```

The function `handleClient(void* vPtr)` is run in its own thread. It receives `vPtr` which points an integer file descriptor for talking to the client. It should:

A. Cast `vPtr` to type `int*` and set `clientFd` to the integer passed
B. `free()` pointer `vPtr`.
C. Get a buffer of text from the client and put it into `buffer[]`.
D. `open()` for reading the file whose name is in `buffer[]` and put the file descriptor in `fileFd`.
E. Set the histogram counts to 0. (I did this for you.)
F. In a loop, `read()` from `fileFd` into `buffer[]`. Make `numChars` equal to the number of chars read into `buffer[]`.
G. Count the chars. (I did this for you.)
H. Close `fileFd`.
I. Send all `HISTOGRAM_LEN` values of `histogram[]` back to the client *in network endianness!*
J. Close `clientFd`.
K. Return `NULL`.

Do not worry about error checking!

```
#define BUFFER_LEN      256
#define HISTOGRAM_LEN   96

void*   handleClient    (void*  vPtr)
{
  char  buffer[BUFFER_LEN];
  int   histogram[HISTOGRAM_LEN];
  int   clientFd        = 0;    // (a) <-- change that 0
  int   fileFd;
  int   numChars;
  int   i;

  // (b)

  // (c)

  // (d)


  //  (e) Already done
  for  (i = 0;  i < HISTOGRAM_LEN;  i++)
    histogram[i]        = 0;

  while  ( /* (f)                                        */ )
  {
    for  (i = 0;  i < numChars;  i++)
    {
      //  (g) Already done
```

```
      char      c = buffer[i];

      if  ( (c >= 32) && (c <= 127) )
        histogram[c-32]++;
    }

  }

  // (h)


  for  (i = 0;  i < HISTOGRAM_LEN;  i++)
  {
    // (i)


  }

  // (j)

  // (k)

}
```

Useful C/Linux Functions in CSC 374 Computer Systems 2

Last modified 2017 June 1

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`.
`char* strstr (const char* haystack, size_t needle)`	Finds the first occurrence of the substring `needle` in the string `haystack`. Returns address of first match if found, or `NULL` otherwise. The ending '\0' is not considered.
`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: ***0x***40 — Hexadecimal 40 (= 64 decimal) ***0***40 — 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 `arg` 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;

memset(&action,'\0',sizeof(action));
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

Useful flags include:

Flag	Meaning
`SA_NOCLDSTOP`	(For `SIGCHLD`) only do the child handler when the child ends (not when it pauses)

	<table><tr><td>SA_RESTART</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.)

<pre>struct sigaction   action;
 	
memset(&action,'\0',sizeof(action));
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>
	Tells the OS that when this process receives signal `signalNum` it is to do function `advancedHandler`. `advancedHandler` should have form:
	
	`void advancedHandler (int sigNum, siginfo_t* infoPtr, void* dataPtr`
	
	`infoPtr` gives all kinds of info. Perhaps among the most useful is `infoPtr->si_pid` which tells the process id of who sent the signal (or maybe `0` if coming from the OS or hardware).
	
	`dataPtr` is not used so much.
	
	See above for the descriptions of the `signalNum` and flags.

<pre>pid_t wait(int* ptr)</pre>	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 `*ptr` equal to the status returned by the child and returns the process id of the child. If child ended normally then `WIFEXITED(childStatus)` return non-zero. If the child crashed then `WIFEXITED(childStatus) == 0` If the child end normally then the portion of the status that

	was `return()`ed by child's `main()`, or which the child `exit()`ed, is obtained by `WEXITSTATUS(childStatus)` If there are no children for which to wait() then return 0.
`pid_t waitpid(pid_t pid, int* statusPtr, int options)`	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: Value: Meaning 0: Act just like `wait()` WNOHANG: Return immediately if no child has exited
`void exit(int status)`	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)`

Threading-related functions

Be sure to:

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

Function	What it does

`extern void*  someFnc (void* vPtr);`
`pthread_t  threadId;`
`Whatever  someArgs;`
 	
`pthread_create(&threadId,NULL,someFnc,(void*)&someArgs);`
	Makes a thread in the space pointed to by `threadPtr`. The thread runs the function `void* fncName(void* vPtr)` and passes `arg` to it. Just leave `attr` as `NULL` for a generic thread, but do the following for a detached thread:
	
	`pthread_attr_t  threadAttr;`

	`pthread_attr_init(&threadAttr);`
	`pthread_attr_setdetachstate(&threadAttr, PTHREAD_CREATE_DETACHED);`
	`pthread_create(..,&threadAttr,..,..);`
	`pthread_attr_destroy(&threadAttr);`

`int pthread_join`
`(/* Which thread to wait for */`
`pthread_t thread,`
 	
`/* Pointer to pointer to receive pointer`
`returned by exiting thread's function.`
`*/`
`void** valuePtrsPtr`
 	
`)`
	Waits for thread `thread` to finish. When it does `valuePtr` (the thing that `valuePtrsPtr` points to) is set to the thread's 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 erro

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 entrie 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 `strea` 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:

<table>
<tr>
<td>Open a file</td>
<td>
<code>int open(const char* pathname, int flags, mode_t mode)</code> Returns a file descriptor (an integer <code>0</code> or greater) on success, or <code>-1</code> on failure.
<code>filename</code> is the path of the file.
<code>flags</code> tells how to open the file.

One of these flags must be given:
<table>
<tr><td><code>O_RDONLY</code></td><td>Read-only</td></tr>
<tr><td><code>O_WRONLY</code></td><td>Write-only</td></tr>
<tr><td><code>O_RDWR</code></td><td>Read and Write</td></tr>
</table>
Additionally, for <code>O_WRONLY</code> one or more of these flags are commonly bitwise-ORed together.
<table>
<tr><td><code>O_APPEND</code></td><td>Append to end of file</td></tr>
<tr><td><code>O_CREAT</code></td><td>Create file if it does not already exist</td></tr>
<tr><td><code>O_TRUNC</code></td><td>Truncate (erase and write over) the file if it already exists (as opposed to appending)</td></tr>
</table>
</td>
</tr>
<tr>
<td>Open a pipe</td>
<td>
<pre>int pipeFd[2]; // Requires an array of 2 integers

int pipe(pipeFd);</pre>
Creates a one-way data channel. If the <code>pipe()</code> call succeeds (if it returns <code>0</code>) then

<code>pipeFd[0]</code> is set to an input file descriptor (<code>read()</code>-ing end) of the channel
<code>pipeFd[1]</code> is set to an output file descriptor (<code>write()</code>-ing end) of the channel

</td>
</tr>
</table>

<table>
<tr><td>Get a file descriptor for a socket</td><td>

```
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

</td></tr>
<tr><td>Server monopolize a port.</td><td>

```
  struct sockaddr_in socketInfo;

  // Fill socketInfo with 0's
  memset(&socketInfo,'\0',sizeof(socketInfo));

  socketInfo.sin_family = AF_INET;         // Use std TCP/IP
  socketInfo.sin_port = htons(port);       // Tell port in
                                           //   network endian
  socketInfo.sin_addr.s_addr = INADDR_ANY; // Allow machine
                                           //   to connect

  // Try to bind socket with port and other specifications
  int status = bind(socketDescriptor, // from socket()
                    (struct sockaddr*)&socketInfo,
                    sizeof(socketInfo)
                   );
```

In the example above:

- Specifies (safer but slower) TCP/IP
- Tries to monopolize port `port`
- Tells the OS "*Let anyone connect*"

Returns `0` on success, or `-1` otherwise.

</td></tr>
<tr><td>Tell server max. number of waiting clients</td><td>

```
int listen(int serverSocketFD, int maxNumWaitingClients)
```

Tells OS that the server socket file descriptor `serverSocketFD` should have a maximum of `maxNumWaitingClients` clients waitin

</td></tr>
</table>

	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.

Client lookup server by name with DNS and try to connect	`//  Look up server named machineName:`
	`struct addrinfo* hostPtr;`
	`int status = getaddrinfo(machineName,NULL,NULL,&hostPtr);`
	
	`if (status != 0)`
	`{`
	`  fprintf(stderr,gai_strerror(status));`
	`  exit(EXIT_FAILURE);`
	`}`
	
	`//  Connect to server on port port:`
	`struct sockaddr_in server;`
	`memset(&server, 0, sizeof(server));// Clear datastruct`
	`server.sin_family = AF_INET;       // Use TCP/IP`
	`server.sin_port = htons(port);     // Tell port to connect`
	`//  Tell IP address of server`
	`server.sin_addr.s_addr =`
	`  ((struct sockaddr_in*)hostPtr->ai_addr)->sin_addr.s_addr;`
	`//  Attempt to connect using socket file descriptor socketFd`
	`//  (socketFd came from an earlier call to socket().)`
	`status = connect(socketFd,`
	`                 (struct sockaddr*)&server,`
	`                 sizeof(server)`
	`                );`
	
	`if  (status < 0)`
	`{`
	`  fprintf(stderr,"Could not connect %s:%d\n",machineName,port);`
	`  return(EXIT_FAILURE);`
	`}`

	`getaddinfo()` returns `0` on success or something else otherwise. `connect()` returns `0` on success or `-1` otherwise.
Close file, socket, pipe, *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 ir *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()`