CSCB09 – Software Tools and Systems Programming

University of Toronto Scarborough, Fall 2016

Principles of Software Tools

Write small programs that do one thing well.
Expect the output of every program to become the input to another, as yet unknown, program.
Make programs' input formats easy to generate or type.
Use programs to write programs.

Shell Commands

sort – sort –f (case insensitive)–k2(by field 2) –n(accepts numerals) faculty (filename) –r(reverse)
sed – (stream editor) e.g. sed s/the/five/g (changes all the->file on a global basis, not just first of a line)
sed 's/\(.*\) \(.*\)/\2,\1/' (assigns first value, second value to second value, first value)
sort -f -k2 faculty | sed 's/\(.*\) \(.*\)/\2, \1/'(combined with sort, returns sorted list of people by their Lastname, FirstName)
processes line-by-line, unless global command is given
who – who (returns User Terminal LoginTime)
grep – (globally search a regular expression and print) –i(ignores case) –l(el only filenames) –n(line #) –I (eye - files where str DNE)
grep ajr (finds and prints lines of all occurences of ajr in input)
* (matches any number of any character e.g. *.c)
? (matches any one character)
a[12345].pdf (matches any of 1,2,3,4,5)
a[1-5].pdf (same as above)
special treatment of . at BOF, needs to be matched explicitly
tr – (translate) tr '\015'(translates \015) '\012' (to \012, space) e.g. tr ab cd changes all a->c, b->d, tr –d 238 deletes all 2, 3, 8, not 238
find [options] -> find –name (regex)
ls -> lists all things in directory, or prints filename. –l(long list format) –i(inode number) –R(recursively) –a (hidden files)
head/tail – tail -40 (prints the last 40 lines, n=10 by def) filename (file name)
uniq – uniq –c(prefix lines by num occur) –u(only unique) –d(only duplicate) filename (collapses adjacent same lines)
echo – echo -n(terminates autogenerated endline) faculty (string to return) (returns User Terminal LoginTime)
wc – (word count) wc -c(prints bytes count, remember the \n) -l(prints newline count)) -m(prints characters count)
rm - (remove)
mv - (rename)
cat - (show file contents)
cp -> copies source to targ (cp file1 file2), -r(copies recursively) **mv** -> same, deletes original sourcefile **rm ->** -r (subdirectories)
cmp - (compare files byte by byte)
diff - (shows different lines between files)
comm - (shows common lines between files)
join - (prints common lines between two files)
>>file (appends) > (redirects stdout to file) < (makes stdin from file) >&(redirects stderr) | (pipes) |&(pipes stdout and stderr)
File descriptor numbers: 0 is standard input (stdin) 1 is standard output (stdout) 2 is standard error (stderr)
Exit(0) on success, Exit(1) on failure. Remember to perror() when things don't return as expected

Shell Programming

Single quotes block interpretation of most things, double quotes block most things except blackslash, dollar sign, backquote `
Variable declaration -> X=3 (variable declaration, no spaces)
expr -> evaluates expressions (e.g. x=x+1 in Shell is x=`expr $x + 1`)
fprintf -> printf with stream specification: ([stream specification e.g. file, stdout, stdin], "printfstatement %s", printFValue));
test -> evaluates test commands (lt = less than, gt = greater than, eq = numerical equity, '=' is string compare) || && are same as java

backslashes – prevents the interpretation of immediately following symbol e.g. />
single quotes – prevents the interpretation of everything inside them except other sqs, useful for LITERAL literals
double quotes – prevents interpretation of all cept $, backquotes, and backslashes. This helps us do echo "your name is $name"
$n – The nth input argument. $0 is the filename $1 is first, etc… all the way to $9. For $9+, use $* to get an array of cmd arguments
#whatever – comment notation in shell
#!/bin/sh – PATH notation /dev/null -> if sent to a file, makes it blank (doesn't remove it)

Syntax for Iterative Loops and Conditionals in Shell

```
if condition
then
  something
else
  something >&2
  exit 1
fi
```

```
i=0
while test $i –lt 10
do
        i = `expr $i +1
        echo $i
done
```

```
for x in hellogoodbye
do
        something
done
```

C Programming

External functions ->

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

```
int main()
{
    int i;
    extern int gcd(int x, int y);

    for (i = 0; i < 20; i++)
        printf("gcd of 12 and %d is %d\n", i, gcd(12, i));
    return(0);
}
int gcd(int x, int y)
{
    int t;

    while (y) {
        t = x;
        x = y;
        y = t % y;
    }
    return(x);
}
```

Using the extern keyword we can link a function not yet declared. We could also just put extern into gcd.h, have the function gdc into gdc.c, and use the keyword #include "gcd.h", to be compiled with `gcc -Wall -o testgcd testgcd.c gcd.c`

Bytes -> 8 bits, Word -> 4 Bytes, or 32 bits. Thus, integers are -2^31 to 2^31-1 inclusive

Pointers -> high level versions of addresses

```
int i;
int *p;    -> declare p to be of type pointer-to-int

i = 3;
p = &i;   -> assign p as to point to address-of-index(i)
printf("%d\n", *p);   -> "dereference" -- follow a pointer
```

Thus, p+3 yields the address which is 3 integers later, or address(p) + (3*4)

Arrays in C -> not objects, just concatenations of n of the array elements (e.g. int = 4 bytes, array of 10 ints = 40 bytes) – can't tell size

Array Functions -> use int *a, not int a[], because arrays are not objects, and there are no objects in C. Instead, use pointer-to-element

Note the idiom "sizeof x / sizeof x[0]" to find the number of items in an array

The Unix FileSystem

Major features of the unix filesystem

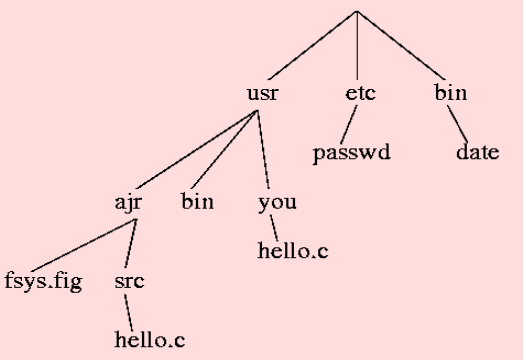
- completely uniform -- organization of system files is all done using one simple hierarchy of directories
  - so users can make their own subdirectories just the same as system ones
  - Yields a directory "tree"; specify a file name in the tree with a "path name", which is how to get there from the root
  - Directories are just a special kind of file. If you want a directory listing, you just read the file which is that directory.
- No restrictions on or system interpretation of file names. File names can be any sequence of up to fourteen characters (now 225)); periods are not special; upper and lower case are considered different, only not allowed is the zero byte (i.e '\0'),
- All of the available disks are integrated into one uniform file system. A disk, or a disk partition, is "mounted" at a certain point in the filesystem.
- Thus it's easy to specify other people's files.
- No difference between "random access files" versus "sequential files", etc. A file is just a list of bytes; the operating system has no involvement on file format(s)

i	mode	contents
2	D	usr 7, etc 4, bin 6
3		*contents of password file*
4	D	passwd 3
5	D	fsys.fig 10, src 11
6	D	date 9
7	D	bin 8, ajr 5, you 12
8	D	...
9		*date program*
10		...
11	D	hello.c 13
12	D	hello.c 14
13		...
14		...
...		

Directory

Filename

Inode num



Symlinks -> soft links, only points to file, does not have contents of file itself (think shortcuts) use the ln –s to create symlinks

Hard links -> have equal status, remove one, other remains. Have contents of file itself, use ln to create hard links

ln -> link command, used like ln /src/filename newfilename, e.g. renaming date in /bin is like `ln (-s) /bin/date d`

CONVENTION: you have a .. entry for every inode table to refer to the parent directory and a . entry for the current directory (each w/#)

Inode -> index node, every single file on a disk has an inode

- "mode" -- the rwxrwxrwx stuff, also includes the 'd' bit for directories
- owner (uid -- "user id") (the owner can change the mode of the file; can affect interpretation of mode)
- group (gid) (cannot change anything, but also changes interpretation of mode)
- times: last modification time (mtime), last access time (atime), and last inode change time (ctime)
- NOT the file name. E.g. a file linked-in twice can have two different names.
- location on disk: list of block numbers, etc

File permissions -> u(owner), g(group), o/a(all users), +/- are used to tell system add or remove permissions

r- read, w – write, x – execute. E.g. _rw_rw_rw means that u, g, o/a have permission to read and write

chmod a-rw file1 (changes mode of file1 to have all users only able to read/write)

Structs in C

```
struct point {
    int x, y;
} a;
Struct point a;
```

accessing values -> a.x. However, we usually declare as pointer-to-struct, so we'd do *b = &a, then b->x or (*b).x
assigning values -> can't initialize values or have defaults, must assign each individually
is the data -> if b=a, and a's value changes, b's values are still the same

➔ **operator deferences and selects at the same time. E.g. (*a).b == a->b**

Files

Three files are open when process starts, stdin, stdout, and stderr, and all files are closed when process exits
Getchar() – gets one character from stdin, returns an int, -1 for EOF
Putchar() – puts one character to stdout
FILE *fp -> abstract file type. Fopen takes fopen(filename, permission) e.g. fopen("a4.c", "r");
Stashes error if null. Find error by calling if((fp = fopen("a4.c", "r")) == NULL){ perror("a4.c"); }
Fgets() – reads a line of input, non-\n characters followed by a \n. fgets(storagevaraible, length, input) e.g. fgets(a, 20, stdin);
Fprintf() – printf but for fiels
Sprintf() – printf but stored
Sscanf() – fscanf but from strings instead of I/o channels
Another cat example, which takes command-line arguments which are file names

```
#include <stdio.h>
int main(int argc, char **argv)
{
    int status = 0;
    for (argc--, argv++; argc > 0; argc--, argv++) {
        FILE *fp;
        if ((fp = fopen(*argv, "r")) == NULL) {
            perror(*argv);
            status = 1;
        } else {
            int c;
            while ((c = getc(fp)) != EOF)
                putchar(c);
            fclose(fp);
        }
    }
    return(status);
}
```

Strings

```
void mystrcpy(char *a, char *b)
{
    while ((*a = *b) != '\0') {
          a++;
          b++;
    }
}
```

strcpy(a, b) -- copy string b to a
strcat(c, d) -- concatenate string d onto the end of string c (modifying c)
strlen(e) -- length of string e (not counting the terminating \0)
strcmp(f, g) -- difference between strings f and g (zero for equal)
strcasecmp(f, g) -- like strcmp but it's a case-insensitive comparison
strchr(h, x) -- find first occurrence of character x in string h (NULL for not found)
strrchr(h, x) -- find last occurrence of character x in string h
strstr(i, j) -- find first occurrence of STRING j in string i
strcasestr(i, j) -- case-insensitive version of strstr
strtok -- break up a string into tokens (words). Limited applicability and strange interface, but very useful when it happens to be suitable.
The char pointers b, c, d, e, f, g, h, i, and j (i.e. all of the string arguments except for 'a') must point to properly zero-terminated strings.
The char pointers a and c must point into an array with enough space to store the result.

Argv and Argc, and in-line commands

```
int main(int argc, char **argv)
```

argc -> argument count, number of elements in argv
argv -> argument vector
getopt -> get the options or limiters. c: below implies that c takes arguments, gotten by optarg

```
34      while ((opt = getopt (argc, argv, "vic:")) != -1){
35          opts++;
36          switch(opt){
37              case 'v':
38                  //v;
39                  ech = 1;
40                  break;
```

```
41              case 'i':
42                  //i;
43                  tty = 1;
44                  break;
45              case 'c':
46                  //c;
47                  nxt = 1;
48                  cmd = optarg;
49                  break;
50          }
51      }
```

Printf and Scanf formatting codes

d-> int ld->long s->string c->character f->float

```
Modifiers appear between the '%' and the key letter.
a number is a field width
'.' and a number is a "precision"
Example: printf("%6.3f", 2.8) yields _2.800 (with a space before the '2')
Note that that "6" includes the 3 decimal places and the 1 '.' -- 6 characters total. Thus 6-3-1 = 2 characters to the
left of the decimal point.
0 (the digit zero) means pad with zeroes to field width (usually used only with integers)
l (the letter) means "long", e.g. %ld to format a long int in decimal
```

etc

The C Preprocessor

#define i 3 never i=3;

```
top = malloc(sizeof(struct list));
if(top == NULL){
        fprintf(stderr, “out of memory! \n”);
        exit(1);
}
Or
Int *y;
y = malloc(z * sizeof(int))          ==      int y[z];
```

Free(things) after you're done using them! Otherwise stack overflow

When casting, remove variable and put parentheses, e.g. char **x → (char **)

Unix Processes

Processes have characteristics: pid (process id), uid (user id that process belongs to), and ppid (parent process id) tty(terminal type)

Execve -> executes a command, used by execve(commandLocation, commandArray, environ), e.g. execve(“/bin/cat”, x, environ);

Environ -> a char ** environ object that points to the first of an array, terminated by the NULL object

Newline standards -> \r and \n are newlines, but you should accept \r\n both as newlines, combined or separate

Interprocess Communication

Basic Idea -> cooperating processes

Big Endian -> Store the most significant byte in the smallest address e.g. $90AB12CD_{16}$ becomes 90->1000, AB->1001, etc… (hex)

Little Endian -> Stores the least significant byte in the smallest address e.g. ^ becomes CD -> 1000, 12 -> 1001, etc. (same as above)

NOTE ABOVE SAYS … SIGNIFICANT BYTE (8 BITS!!!)

Htons/htonl -> host to network long/short (32bits/16bits) ntohs/ntohl same idea

If you want your program to be portable, then any time you send an integer greater than 1 byte in size over the network, you must first convert it to network byte order using htons or htonl, and the receiving computer must convert it to host byte order using ntohs or ntohl. Otherwise, computers with different Endian-ness will mess up the interpretation

"Network byte order" is Big Endian, and protocols such as TCP use this for integer fields (e.g. port numbers). Functions such as htons and ntohs can be used to do conversion.

Child processes -> close(fd[0]), input side of pipe Parent processes -> close(fd[1]), output sides of pipe

dup2 doesn't switch the file descriptors, it makes them equivalent. After dup2(f1, 0), whatever file was opened on f1 is now f0

Multiprocess Implementation

The "MMU" maps addresses, a "page" at a time. It sits between the CPU and the main memory unit, conceptually; these days typically built in to the CPU.

presents a uniform address space; no need to write relocatable code anymore

"virtual address" vs "physical address"

"locality of reference"

Code Examples

Implement File redirection

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
```

```
int main(){
    int x = fork();
    if (x == -1) {
        perror("fork");
        return(1);
    } else if (x == 0) {
        /* child */
        close(1);
        if (open("file", O_WRONLY|O_CREAT|O_TRUNC, 0666) < 0) {
            perror("file");
            return(1);
        }
        execl("/bin/ls", "ls", (char *)NULL);
        perror("/bin/ls");
        return(1);
    } else {
        /* parent */
        int status, pid;
        pid = wait(&status);
        printf("pid %d exit status %d\n", pid, status >> 8);
        return(0);
    }
}
```

Implement Piping

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
int main(){
    int pid, status;
    extern void docommand();
    printf("Executing 'ls | tr e f'\n");
    fflush(stdout);
    switch ((pid = fork())) {
    case -1:
        perror("fork");
        break;
    case 0:
        /* child */
        docommand();
        break;  /* not reached */
    default:
        printf("fork() returns child pid of %d\n", pid);
        pid = wait(&status);
        printf("wait() returns child pid of %d\n", pid);
        printf("Child exit status was %d\n", status >> 8);
    }
    return(0);
}
void docommand()  /* does not return, under any circumstances */
    int pipefd[2];

    /* get a pipe (buffer and fd pair) from the OS */
    if (pipe(pipefd)) {
        perror("pipe");
        exit(127);
    }

    /* We are the child process, but since we have TWO commands to exec we
     * need to have two disposable processes, so fork again */
    switch (fork()) {
    case -1:
        perror("fork");
        exit(127);
    case 0:
        /* child */
        /* do redirections and close the wrong side of the pipe */
        close(pipefd[0]);  /* the other side of the pipe */
        dup2(pipefd[1], 1);  /* automatically closes previous fd 1 */
        close(pipefd[1]);  /* cleanup */
        /* exec ls */
        execl("/bin/ls", "ls", (char *)NULL);
        /* return value from execl() can be ignored because if execl returns
         * at all, the return value must have been -1, meaning error; and the
         * reason for the error is stashed in errno */
        perror("/bin/ls");
        exit(126);
    default:
```

```
        /* parent */
        /*
         * It is important that the last command in the pipeline is execd
         * by the parent, because that is the process we want the shell to
         * wait on.  That is, the shell should not loop and print the next
         * prompt, etc, until the LAST process in the pipeline terminates.
         * Normally this will mean that the other ones have terminated as
         * well, because otherwise their sides of the pipes won't be closed
         * so the later-on processes will be waiting for more input still.
         */
        /* do redirections and close the wrong side of the pipe */
        close(pipefd[1]);  /* the other side of the pipe */
        dup2(pipefd[0], 0);  /* automatically closes previous fd 0 */
        close(pipefd[0]);  /* cleanup */
        /* exec tr */
        execl("/usr/bin/tr", "tr", "e", "f", (char *)NULL);
        perror("/usr/bin/tr");
        exit(125);
    }

    /*
     * When the exec'd processes exit, all of their file descriptors are closed.
     * Thus the "ls" command's side of the pipe will be closed, and thus the
     * "tr" command will get eof on stdin.  But if we didn't have the
     * close(pipefd[1]) for 'tr' (in the default: case), the incoming side
     * of the pipe would NOT be closed (fully), the "tr" command would still
     * have it open, and so tr itself would not get eof!  Try it!
     */
}
```

Server Select Code – UNIX Domain

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
int main(){
    int fd, clientfd;
    socklen_t len;
    char buf[80];
    struct sockaddr_un r, q;

    /*
     * create a "socket", like an outlet on the wall -- an endpoint for a
     * connection
     */
    if ((fd = socket(AF_UNIX, SOCK_STREAM, 0)) < 0) {
        perror("socket");  /* (there's no file name to put in as argument
                            * to perror... so for lack of anything better,
                            * the syscall name is the usual choice here) */
        return(1);
    }
    /* "unix domain" -- rendezvous is via a name in the unix filesystem */
    memset(&r, '\0', sizeof r);
    r.sun_family = AF_UNIX;
    strcpy(r.sun_path, "/tmp/something");
    /*
     * "binding" involves creating the rendezvous resource, in this case the
     * socket inode (a new kind of "special file"); then the client can
     * "connect" to that.
     */
    if (bind(fd, (struct sockaddr *)&r, sizeof r)) {
        perror("bind");
        return(1);
    }
    /*
     * The "listen" syscall is required.  It says the length of the queue for
     * incoming connections which have not yet been "accepted".
     * 5 is a suitable value for your assignment four.
     * It is not a limit on the number of people you are talking to; it's just
     * how many can do a connect() before you accept() them.
     */
    if (listen(fd, 5)) {
          perror("listen");
          return(1);
    }
    /*
     * The accept() syscall accepts a connection and gives us a new socket
     * file descriptor for talking to that client.  We can read and write the
```

```
     * socket.  Other than that it functions much like a pipe.
     */
    len = sizeof q;
    if ((clientfd = accept(fd, (struct sockaddr *)&q, &len)) < 0) {
        perror("accept");
        return(1);
    }
    /*
     * Usually we'd have a more complex protocol than the following, but
     * in this case we're just reading one line or so and outputting it.
     */
    if ((len = read(clientfd, buf, sizeof buf - 1)) < 0) {
        perror("read");
        return(1);
    }
    /* The read is raw bytes.  This turns it into a C string. */
    buf[len] = '\0';
    printf("The other side said: %s\n", buf);
    /*
     * Closing the socket makes the other side see that the connection is
     * dropped.  It's how you "hang up".
     */
    close(clientfd);
    close(fd);
    unlink("/tmp/something");
    return(0);
}
```

Client Connect Code – UNIX domain

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
int main(){
    int fd, len;
    struct sockaddr_un r;
    /*
     * create a "socket", like an outlet on the wall -- an endpoint for a
     * connection
     */
    if ((fd = socket(AF_UNIX, SOCK_STREAM, 0)) < 0) {
        perror("socket");  /* (there's no file name to put in as argument to
                            *  perror... so for lack of anything better, the
                            *  syscall name is the usual choice here) */
        return(1);
    }
    /* "unix domain" -- rendezvous is via a name in the unix filesystem */
    memset(&r, '\0', sizeof r);
    r.sun_family = AF_UNIX;
    strcpy(r.sun_path, "/tmp/something");  /* and this is the name */
    /* The server does an "accept"; the client does a "connect": */
    if (connect(fd, (struct sockaddr *)&r, sizeof r)) {
        perror("connect");
        return(1);
    }
    /* at this point we have connected to the socket successfully */

    /* send the transmission */
    if ((len = write(fd, "Hello", 5)) != 5) {
        perror("write");
        return(1);
    }
    /*
     * exiting reclaims all resources, including open files, open pipes,
     * open sockets
     */
    return(0);
}
```

Past Exam(s) Code

Which of the following quotes are unnecessary?

```
foo &
foopid="$!"                                     1
while kill -0 "$foopid"                         2
do
    message="`foostatus`"                       3
    if test "$message" = "Terminated"           4         5
    then
        exit 0
    else
        echo foo status: $message
    fi
done
exit 0
```

ANSWER:1, 2, 5

Version of Head - This program takes zero or more file name arguments; (0 = process stdin)

```
#include <unistd.h>
int main(int argc, char **argv){
    int c, status = 0, n = 10;
    FILE *fp;
    extern void process(FILE *fp, int lines);

    while ((c = getopt(argc, argv, "n:")) != EOF) {
        if (c == 'n') {
            /* (for this exam question, calling atoi() was fine) */
            char tmp;
            if (sscanf(optarg, "%d%c", &n, &tmp) != 1)
                status = 1;
        } else {
            status = 1;
        }
    }
    if (status) {
        fprintf(stderr, "usage: %s [-n num] [file ...]\n", argv[0]);
        return(1);
    }
    if (optind == argc) {
        process(stdin, n);
    } else {
        for (; optind < argc; optind++) {
            if ((fp = fopen(argv[optind], "r")) == NULL) {
                perror(argv[optind]);
                status = 1;
            } else {
                process(fp, n);
                fclose(fp);
            }
        }
    }
    return(status);
}
void process(FILE *fp, int lines){
    char buf[500];
    for (; lines > 0 && fgets(buf, sizeof buf, fp); lines--)
        printf("%s", buf);
}
```

The C library function "system" executes a command by passing it to the shell. For example, if a C program wants to put you in the 'vi' editor on a file named "file", it could say
status = system("vi file");
The system() function does not return until the subprocess has exited. It returns the exit status of the command as the return value of system().It runs, basically
/bin/sh -c 'vi file'
except, of course, that it does it via fork()/execl()/wait(). (If the fork() fails, system() returns -1.)
Write the system() C library function.

ANSWER:

```
int system(char *cmd)
{
    int pid = fork();
    if (pid < 0)
        return(-1);
    if (pid == 0) {
        execl("/bin/sh", "sh", "-c", cmd, (char *)NULL); // the last command is the environ
        perror("/bin/sh");
        exit(127);
    } else {
        int status;
        wait(&status);
```

```
        return(status >> 8);
    }
}
```

The current directory contains files named 1,2,3. Write a program to listen on TCP/IP port 1234, and each time there is a connection, it outputs the contents of one of these files to the connection and closes. Looping

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
int main(){
    int listenfd, clientfd, len;
    struct sockaddr_in r, q;
    static char *files[3] = { "1", "2", "3" };
    int whichfile = 0;
    FILE *fp;
    int c;
    if ((listenfd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        perror("socket");
        return(1);
    }
    memset(&r, '\0', sizeof r);
    r.sin_family = AF_INET;
    r.sin_addr.s_addr = INADDR_ANY;
    r.sin_port = htons(1234);
    if (bind(listenfd, (struct sockaddr *)&r, sizeof r) < 0) {
        perror("bind");
        return(1);
    }
    if (listen(listenfd, 5)) {
        perror("listen");
        return(1);
    }
    while (1) {
        len = sizeof q;
        if ((clientfd = accept(listenfd, (struct sockaddr *)&q, &len)) < 0) {
            perror("accept");
            return(1);
        }
        if ((fp = fopen(files[whichfile], "r")) == NULL) {
            perror(files[whichfile]);
        } else {
            while ((c = getc(fp)) != EOF) {
                if (c == '\n') {
                    write(clientfd, "\r\n", 2);
                } else {
                    char cc = c;
                    write(clientfd, &cc, 1);
                }
            }
            fclose(fp);
            whichfile = (whichfile + 1) % 3;
        }
        close(clientfd);
    }
}
```

SHELL PROGRAMMING

Write a for loop to be executed in sh which goes through your directory "/usr/wilma/letters" (you are Wilma) and for each file in the directory, executes the command "grep Fred file". (Don't use xargs or find.)

ANSWER:

```
for i in usr/wilma/letters/*
do
         grep Fred "$i"
done
```

a) Delete all files in /tmp with an 'x' in their name.

```
rm /tmp/*x*
```

b) Delete all files in /tmp with an 'x' in the file (the contents of the file, as opposed to the file name).

```
find /tmp -type f | while read filename
                    do
                            if grep -q x "$filename"
                            then
                                rm "$filename"
                            fi
                  done
```

c) Make subdirectories named "even" and "odd", and move all other files into if bytes is even or odd.

```
mkdir even odd
for filename in *
```

```
do
    case "$filename" in
         even|odd)
             ;;
         *)
             case `wc -c <"$filename"` in
                  *[02468])
                      mv "$filename" even
                      ;;
                  *[13579])
                      mv "$filename" odd
                      ;;
             esac
             ;;
    esac
done
```

Remove files in /home/ajr/q6 if they have exactly 3 lines

```
for i in /home/ajr/q6/*
do
    if test -f "$i"
    then
        if test `wc -l <"$i"` -eq 3
        then
            rm "$i"
        fi
    fi
done
```

There is a plagiarism-checking program called "cheating", where "cheating file1 file2" outputs the probability, as a percentage value between 0 and 100 inclusive, that file1 and file2 are programs of students cheating off of each other. Print first 10 most likely cheaters

```
cd /u/submit/cscb09/a1
for i in *
do
    for j in *
    do
        echo `cheating "$i" "$j"` "$i" "$j"
    done
done | sort -nr | head
```

Write a shell script (in the "sh" programming language) which takes one or more file name arguments, and outputs the name of the file which has the most lines in the opinion of "wc -l". (In case of a tie, any tiename is okay)

```
if test $# -eq 0
then
    echo usage: $0 file ... >&2
    exit 1
fi

max=-1
for i
do
    this=`wc -l <"$i"`
    if test $this -gt $max
    then
         max=$this
         maxname="$i"
    fi
done
echo "$maxname"
```

Write a C program which is a simplified version of the "test" command. Your program will support only -f (test if a file exists and is a plain file), the two numeric relational operators -lt and -eq (the other four are very similar, after all), and the two string relational operators = and !=. So argc needs to be 3 for -f and 4 else

```
#include <stdio.h> #include <stdlib.h> #include <string.h> #include <sys/types.h> #include <sys/stat.h>
int main(int argc, char **argv){
    extern int fexists(char *file), usage();
    if (argc == 3 && strcmp(argv[1], "-f") == 0)
         return(fexists(argv[2]));
    else if (argc != 4)
         return(usage());
    else if (strcmp(argv[2], "-lt") == 0)
         return(!(atoi(argv[1]) < atoi(argv[3])));
    else if (strcmp(argv[2], "-eq") == 0)
         return(!(atoi(argv[1]) == atoi(argv[3])));
    else if (strcmp(argv[2], "!=") == 0)
         return(!strcmp(argv[1], argv[3]));
    else if (strcmp(argv[2], "=") == 0)
         return(!!strcmp(argv[1], argv[3]));
    else
         return(usage());
}
```

```
int fexists(char *file)
{
    struct stat statbuf;
    return(lstat(file, &statbuf) < 0 || !S_ISREG(statbuf.st_mode));
}
int usage()
{
    fprintf(stderr, "usage: test { -f file | item1 {-lt|-gt|=|!=} item2 }\n");
    return(2);  /* for the exam, it's ok if you used exit status of 1 here */
}
```

Write fgets in terms of getc()

```
char *myfgets(char *buf, int size, FILE *fp){
    int pos, c;
    pos = 0;
    while (size - pos > 1) {
          if ((c = getc(fp)) == EOF) {
              if (pos)
                    break;
              return(NULL);
          }
          buf[pos++] = c;
          if (c == '\n')
              break;
    }
    buf[pos] = '\0';
    return(buf);
}
```

Recursive 'squid' file finding

```
void rfind(int depth){    DIR *dp;  struct dirent *r;   struct stat statbuf;
    if ((dp = opendir(".")) == NULL) {
          perror("opendir");
          exit(1);
    }
    while ((r = readdir(dp))) {
          if (strcmp(r->d_name, "squid") == 0) {
              printf("%d\n", depth);
              exit(0);
          }
          if (lstat(r->d_name, &statbuf)) {
              perror(r->d_name);
              exit(1);
          }
          if (S_ISDIR(statbuf.st_mode)
                    && strcmp(r->d_name, ".")
                    && strcmp(r->d_name, "..")) {
              if (chdir(r->d_name)) {
                    perror(r->d_name);
                    exit(1);
              }
              rfind(depth+1);
              if (chdir("..")) {
                    perror("..");
                    exit(1);
              }
          }
    }
    closedir(dp);
}
```

Write a program which creates two child processes using fork(), each connected with a pipe so they can send data to the parent. One of the children writes the number "12" to the parent over its pipe. The other child writes the number "49" to the parent over its pipe. The parent reads both of these and adds them and outputs the sum to stdout.

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

int main()
{
    int x, y;
    extern void dofork(int i, int *fd);
    extern int readint(int fd);
    dofork(12, &x);
    dofork(49, &y);
    printf("%d\n", readint(x) + readint(y));
    return(0);
}

void dofork(int i, int *fd)
```

```
{
    int pipefd[2];
    if (pipe(pipefd)) {
         perror("pipe");
         exit(1);
    }
    switch (fork()) {
    case -1:
         perror("fork");
         exit(1);
    case 0:
         if (write(pipefd[1], &i, sizeof i) != sizeof i) {
             perror("write");
             exit(1);
         }
         exit(0);
    default:
         *fd = pipefd[0];
    }
}

int readint(int fd)
{
    int x;
    if (read(fd, &x, sizeof x) != sizeof x) {
         fprintf(stderr, "read problems\n");  /* good enough for a test... */
         exit(1);
    }
    return(x);
}
```

Translate Euclid's greatest common divisor algorithm

```
While (y>0){
         Int t = x;
         X = y;
         Y = t % y;
}
PATH=/bin:/usr/bin
if test $# -ne 2
then
    echo usage: gcd x y >&2
    exit 1
fi
x=$1
y=$2
while test $y -gt 0
do
    t=$x
    x=$y
    y=`expr $t % $y`
done
echo $x
```

Binary Search Guessing Algorithm

```
low=0
high=101

while test $low -lt `expr $high - 1`
do
    guess=`expr $low + $high`
    guess=`expr $guess / 2`
    echo "Is your number >=$guess or <$guess?  Enter 'g' or 'l'."
    read input
    case "$input" in
         l)
             high=$guess
             ;;
         g)
             low=$guess
             ;;
    esac
done
echo Your number must be $low.
```

Write a C program which listens on port number 1234 and receives a series of connections. It will handle only one connection at a time. Once a user connects, it will run a "chat" between the user connecting over the network (e.g. with "telnet") and the user running the program (and with access to its stdin and stdout). When the remote user disconnects, your program will wait for the next user, until it is killed.

```
#include <stdio.h> #include <stdlib.h>
#include <string.h> #include <unistd.h> #include <sys/types.h>
#include <sys/socket.h>
```

```
#include <netinet/in.h>
int main(){
    struct sockaddr_in r, q;
    int listenfd, clientfd;

    if ((listenfd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        perror("socket");
        exit(1);
    }
    memset(&r, '\0', sizeof r);
    r.sin_family = AF_INET;
    r.sin_addr.s_addr = INADDR_ANY;
    r.sin_port = htons(1234);
    if (bind(listenfd, (struct sockaddr *)&r, sizeof r)) {
        perror("bind");
        exit(1);
    }
    if (listen(listenfd, 5)) {
        perror("listen");
        exit(1);
    }
    while (1) {
          char buf[30];
          socklen_t len = sizeof q;
          if ((clientfd = accept(listenfd, (struct sockaddr *)&q, &len)) < 0) {
              perror("accept");
              return(1);
          }
          while (1) {
              fd_set fdlist;
              int from, to, n;
              FD_ZERO(&fdlist);
              FD_SET(clientfd, &fdlist);
              FD_SET(0, &fdlist);
              if (select(clientfd+1, &fdlist, NULL, NULL, NULL) < 0) {
                    perror("select");
                    exit(1);
              }
              if (FD_ISSET(0, &fdlist)) {
                    from = 0;
                    to = clientfd;
              } else {
                    from = clientfd;
                    to = 0;
              }
              if ((n = read(from, buf, sizeof buf)) < 0) {
                    perror("read");
                    exit(1);
              }
              if (n == 0)
                    break;
              if (write(to, buf, n) != n) {
                    perror("write");
                    exit(1);
              }
          }
          close(clientfd);
    }
}
```

Write a program in C which interactively builds up a list of i/o redirections in accordance with the user's instructions, then executes a command with those i/o redirections in place, then loops around and repeats.
Sample interaction:

```
Which file descriptor do you want to redirect?  -1 to end the list.
1
Where do you want to redirect file descriptor 1 to/from?
blahblah
Is blahblah to be opened for read or write?  ('r' or 'w')
w
Which file descriptor do you want to redirect?  -1 to end the list.
0
Where do you want to redirect file descriptor 0 to/from?
/dev/null
Is /dev/null to be opened for read or write?  ('r' or 'w')
r
Which file descriptor do you want to redirect?  -1 to end the list.
-1
What program do you want to execute?
ls something
Executing:
```

```
          something: No such file or directory
          exit status 1
          Which file descriptor do you want to redirect?  -1 to end the list.
          38
          Where do you want to redirect file descriptor 38 to/from?
          ^D
```

The user typed the lines in bold. At the end, the user pressed ^D signalling end-of-file, and in response the program exited. This can happen at any time.

I recommend using fgets() to read the user input (in all cases). You can use atoi() to convert the string into an integer. You can use strsave() from assignment three if you like. You can assume that all user inputs are correct.

Your program will ignore argv (and need not declare argc or argv in main's parameter list).

Since the user can type any number of redirections, you will use a linked list to store them all. Then after the fork(), the child will implement them all in a loop, and the parent will free all of the malloc'd memory before looping around to solicit the next command setup.

To open a file for read, the second argument of open() should be O_RDONLY. To open a file for write, the second argument of open() should be O_WRONLY|O_CREAT|O_TRUNC.

You can execute the user's command by passing it to "sh -c" as system() does.

This is basically three exam questions. Grading will be divided roughly as follows:
8a [10 marks] user interaction
8b [10 marks] build the linked list of redirections; free it afterwards
8c [10 marks] fork/exec/wait and i/o redirections

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/wait.h>
struct list {
    int fd;
    char *filename;
    int mode;
    struct list *next;
} *top;

char *prog;

extern void buildlist(), getprog(), executeit(), freelist(struct list *p);
extern char *mygetline(), *estrsave(char *s);
int main(){
    while (1) {
          buildlist();
          getprog();
          executeit();
          freelist(top);
    }
}
void buildlist(){
    int fd;
    top = NULL;
    while (printf(
"Which file descriptor do you want to redirect?  -1 to end the list.\n")
              , (fd = atoi(mygetline())) >= 0) {
          struct list *p = malloc(sizeof(struct list));
          if (p == NULL) {
              fprintf(stderr, "out of memory!\n");
              exit(1);
          }
          p->next = top;
          top = p;
          p->fd = fd;
          printf("Where do you want to redirect file descriptor %d to/from?\n", p->fd);
          p->filename = estrsave(mygetline());
          printf("Is %s to be opened for read or write?  ('r' or 'w')\n", p->filename);
          p->mode = (mygetline()[0] == 'r') ? O_RDONLY : (O_WRONLY|O_CREAT|O_TRUNC);
    }
}
void getprog(){
    printf("What program do you want to execute?\n");
    prog = mygetline();
}
```

```
void executeit(){
    int pid;
    printf("Executing:\n");
    fflush(stdout);

    if ((pid = fork()) < 0) {
          perror("fork");
          exit(1);
    } else if (pid == 0) {
          for (; top; top = top->next) {
              int fd;
              if ((fd = open(top->filename, top->mode, 0666)) < 0) {
                    perror(top->filename);
                    exit(126);
              }
              if (fd != top->fd) {
                    dup2(fd, top->fd);
                    close(fd);
              }
          }
          execl("/bin/sh", "sh", "-c", prog, (char *)NULL);
          perror("/bin/sh");
          exit(125);
    } else {
          int status;
          if (wait(&status) < 0) {
              perror("wait");
              exit(1);
          }
          printf("exit status %d\n", status >> 8);
    }
}
void freelist(struct list *p){
    if (p) {
          free(p->filename);
          freelist(p->next);
          free(p);
    }
}
char *mygetline(){
    static char buf[500];
    char *p;
    if (fgets(buf, sizeof buf, stdin) == NULL)
          exit(0);
    if ((p = strchr(buf, '\n')))
          *p = '\0';
    return(buf);
}
char *estrsave(char *s){
    char *q;
    if (!s)
          return(NULL);
    if ((q = malloc(strlen(s) + 1)) == NULL) {
          fprintf(stderr, "out of memory\n");
          exit(1);
    }
    strcpy(q, s);
    return(q);
}
```

Write a simplified version of man in C.
Your man command will take zero or more arguments, and loop through them. For each argument, it searches all eight possible 'man' directories, in order, for a file name beginning with the supplied name and a dot. For the first one it finds, it executes /usr/bin/less (using fork() and execl()), specifying the absolute path name to the appropriate file as argv[1].
The man directories are /usr/share/catman/man1, /usr/share/catman/man2, and so on through /usr/share/catman/man8 (inclusive).
A sample possible file name matching "man cat" is /usr/share/catman/man1/cat.1 Another sample possible file name matching "man cat" is /usr/share/catman/man1/cat.ln However, the file name /usr/share/catman/man1/catch.1 must not be deemed to match "man cat" -- you need to check for the dot as well.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <dirent.h>
#include <sys/types.h>
#include <sys/wait.h>
#define CATMAN "/usr/share/catman"
#define LESS "/usr/bin/less"
```

```
extern char *find(int manno, char *name);
extern void man(int manno, char *filename);
int main(int argc, char **argv){
    int i;
    char *p;

    for (argc--, argv++; argc > 0; argc--, argv++) {
        for (i = 1; i <= 8 && (p = find(i, *argv)) == NULL; i++)
            ;
        if (p)
            man(i, p);
        else
            fprintf(stderr, "No manual entry for %s\n", *argv);
    }
    return(0);
}

char *find(int manno, char *name)
{
    char dirname[100];
    DIR *dp;
    struct dirent *r;
    extern int ismatch(char *name, char *filename);

    sprintf(dirname, "%s/man%d", CATMAN, manno);
    if ((dp = opendir(dirname)) == NULL) {
        perror(dirname);
        exit(1);
    }
    while ((r = readdir(dp)) && !ismatch(name, r->d_name))
        ;
    closedir(dp);
    return(r ? r->d_name : NULL);
}

int ismatch(char *name, char *filename)
{
    int len = strlen(name);
    return(strncmp(name, filename, len) == 0 && filename[len] == '.');
}

void man(int manno, char *filename)
{
    /* max dirname size is <100, and max filename size in bsd fsys is 256 */
    char fullpath[356];
    int pid = fork();
    if (pid == -1) {
        perror("fork");
        exit(1);
    } else if (pid == 0) {
        sprintf(fullpath, "%s/man%d/%s", CATMAN, manno, filename);
        execl(LESS, "less", fullpath, (char *)NULL);
        perror(LESS);
        exit(1);
    } else {
        if (wait((int *)NULL) < 0)
            perror("wait");
    }
}
```

Write a C program which listens on port number 1234 and tells each caller how many people have connected so far. You could connect to it by typing "telnet hostname 1234" (for the appropriate hostname), and it would output "1" and close the connection. The next person would be told "2", etc. The program does not exit (until it is killed).

You do not have to handle multiple simultaneous incoming connections. Remember to use the network newline convention -- end the output with a newline. You can omit the #includes, and you can cite course web pages rather than copying out bits of code if you can do so unambiguously.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>


int main()
{
```

```
    int count = 0;
    struct sockaddr_in r, q;
    int listenfd, clientfd;

    if ((listenfd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        perror("socket");
        exit(1);
    }

    memset(&r, '\0', sizeof r);
    r.sin_family = AF_INET;
    r.sin_addr.s_addr = INADDR_ANY;
    r.sin_port = htons(1234);

    if (bind(listenfd, (struct sockaddr *)&r, sizeof r)) {
        perror("bind");
        exit(1);
    }

    if (listen(listenfd, 5)) {
        perror("listen");
        exit(1);
    }

    while (1) {
        char buf[30];
        socklen_t len = sizeof q;
        if ((clientfd = accept(listenfd, (struct sockaddr *)&q, &len)) < 0) {
            perror("accept");
            return(1);
        }
        sprintf(buf, "%d\r\n", ++count);
        write(clientfd, buf, strlen(buf));
        close(clientfd);
    }
}
```

ASSIGNMENTS CODE

Assignment 1

Write a shell script which produces output in three categories:

the list of letters which are repeated in the 'from' letter list (column 1)
the list of letters which are missing from the 'from' letter list
the list of letters which are repeated in the 'to' letter list (column 2)

```
 1  #! bin/bash
 2
 3  # If there are the required (1) arguments
 4  if test $# -eq 1
 5  then
 6      echo Letters repeated in the from list:
 7      # Cuts the first column of the line and pipes it to
 8      # Sort which sorts all  of the column alphabetically
 9      # Piped into uniq -d which only prints the duplicate adjacent chars
10          cut -c1  $1 | sort | uniq -d
11      echo Letters missing from the from list:
12          # loops through alphabet
13          for i in `cat /cmshome/ajr/b09/a1/atoz`; do
```

```
            # checks if every element exists in key file
            exist=false
            for j in `cut -c1 $1 | sort | uniq`; do
                if [ $i = $j ]
                then
                    exist=true
                fi
            done
            # If it doesnt exist, then print out the alphabet element that DNE in the key file
            if [ $exist = false ]
            then
                echo $i
            fi
        done
    echo Letters repeated in the to list:
    # Outlines second element after delimiter
    # Piped into sort and uniq -d which only prints duplicate adjacent chars
        cut -d' ' -f 2 $1 | sort | uniq -d
# Otherwise prints an error message and exits
else
    echo usage: checkkey file
fi
```

tr is a relatively simple C program, but not simple enough for assignment one, especially since we haven't done strings in C yet.
Your very simple tr program will be called "vstr.c" and will translate only one character to only one other character, such as in commands like "tr e f", and won't do backslash escapes.

```
#include <stdio.h>

int main(int argc, char **argv)
{
    int from, to;
    if (argc != 3) {
        fprintf(stderr, "usage: vstr fromchar tochar\n");
        return(1);
    }
    from = argv[1][0];
    to = argv[2][0];

    int c;

    while ((c = getchar()) != EOF) {
        if(c == from){
            putchar(to);
        }
        else{
            putchar(c);
        }
    }
    return(0);
}


```

Next, write a C program called "undigit.c" which behaves like "tr -d 0-9". It takes no command-line arguments and won't even declare the argc and argv arguments to main(). It reads from its standard input and writes to its standard output, copying everything except omitting digits.
Note: you might be tempted to do comparisons such as "c >= 48" and "c <= 57" to see if c is a digit. This will work but is less clear to the reader than writing things like "c >= '0'". Remember that char literals such as '0' in C are of type int -- in the ASCII character set (such as we have in unix or linux), '0' means 48, but in other character sets it would have different values... and most of all, '0' is more obvious to the reader. The reader might not have the ASCII character set memorized, but even if they do, it's clearer why you are saying 48 if you write it as '0'. (And don't confuse '0' with '\0' -- definitely do ask if you have the slightest confusion on this matter; they're completely different.)

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

int main(){
    char c;
    while ((c = getchar()) != EOF) {
        if(!((c>='0') && (c<='9'))){
            putchar(c);
        }
    }
    return 0;
}
```

Finally, write "subst2tr", as a shell script. It takes one mandatory argument, which is the decryption key file name. It needs to process all of this file, reading through with a loop such as "while read from to",

accumulating the 'from' and 'to' letter lists, and outputting the two strings which are suitable arguments to tr. You can compare the behaviour of /cmshome/ajr/b09/tut/02/subst2tr

```
#! bin/bash

if test $# -eq 1
then
    # Reads lines from file and prints all the first column (from)
    cat $1 | while read LINE
    do
        from=`echo $LINE|cut -c1`
        echo -n $from
    done
    echo -n  " "
    # Reads lines from file and prints all the last column (to)
    cat $1 | while read LINE
    do
        to=`echo $LINE|cut -d' ' -f 2`
        echo -n $to
    done
        echo ""
else
    echo usage: subst2tr file
fi
```

Assignment 2

Write a standard unix tool in C which is a simplified fgrep: its only command-line options are -l, -h, and -m (use getopt() to parse the command-line options). Like most standard unix tools, it takes zero or more file names on the command-line in the usual way.
Call your program "myfgrep".

You can check whether a string occurs in another string with strstr(). Since you don't care where in the string the search string occurs, all you need to check is whether strstr() returns NULL.

If there are more than one file names on the command line, each matching line is prefaced by the file name and a colon (and nothing else; no space). However, the -h option suppresses this.

The -l option means that instead of outputting the matching lines, we output the file names which contain matches. Note that each command-line file name is output a maximum of once even if multiple matches occur in the file. If the standard input matches, output the string "stdin". If -l is specified, -h has no effect.

The -m option takes a count, and stops processing after that many lines have matched. The count is not reset between files.

You must parse the command-line options with getopt() so as to accommodate all standard allowable command-line variations. There is an example getopt()-using program in the "Course notes" section of the CSC B09 web site.

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

typedef int bool;
#define true 1
#define false 0

int main(int argc, char **argv){
    bool hideFilenames = false;
    bool onlyFilenames = false;
    bool specificCounter = false;
    int counter = 0;
    int linesMatched = 0;
    if(argc > 1){
        int opt;
        int optionsTaken = 1;
        while ((opt = getopt (argc, argv, "lhm:")) != -1){
            optionsTaken++;
            switch(opt){
                case 'l':
                    onlyFilenames = true;
                    break;
                case 'h':
                    hideFilenames = true;
                    break;
                case 'm':
                    specificCounter = true;
                    counter = atoi(optarg);
                    break;
            }
```

```
        }
        int i;
        for(i = optionsTaken+1; i < argc; i++){
            FILE *fp;
            fp =fopen(argv[i],"r");
            if (!fp){
                printf("%s: No such file or directory\n", argv[i]);
                return 2;
            }
            char inputLine[1000];
            while (fgets(inputLine, 1000, fp)!=NULL){
                // If nothing is specified, or something is specified and there is matches<allowed
                if(!specificCounter || (specificCounter&&(linesMatched < counter))){
                    if(strstr(inputLine, argv[optionsTaken])!= NULL){
                        if(onlyFilenames){
                            printf("%s\n", argv[i]);
                            linesMatched++;
                            break;
                        }
                        else{
                            if(argc > 2 && !hideFilenames)
                                printf("%s:", argv[i]);
                            printf("%s", inputLine);
                            linesMatched++;
                        }
                    }
                    else{
                        continue;
                    }
                }
            }
        }
    }
    else{
        printf("usage: myfgrep [-lh] [-m count] searchstring [file ...]\n");
        return 1;
    }
    return 0;
}
```

Write a standard unix tool in C which is like "find –name": it recursively traverses one or more directories, looking for a file with the exact specified name (not a substring). It outputs an appropriate path name for each match.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <dirent.h>
#include <sys/stat.h>

char* concat(char *str1, char *str2)
{
    char *result = malloc(strlen(str1)+strlen(str2)+1);
    strcpy(result, str1);
    strcat(result, str2);
    return result;
}

int combDirectoryForFile(char *path, char *dirName, char *fileName, int occur){
    char *pathname= path;
    int hasBeenFound = 0;
    DIR *d;
    struct dirent *dir;
    d = opendir(dirName);
    if (d){
        while ((dir = readdir(d)) != NULL){
            // Skips the . and .. cases
            if(strcmp(dir->d_name, ".")==0 || strcmp(dir->d_name, "..")==0){
                continue;
            }
            // If the type is a file
            if (dir->d_type == DT_REG && strcmp(dir->d_name, fileName)==0){
                printf("%s/%s\n", pathname,fileName);
                hasBeenFound = 1;
            }
            // If its a directory
            else if(dir->d_type == DT_DIR){
                if(strcmp(dir->d_name, fileName)==0){
                    printf("%s/%s\n", pathname,fileName);
                    hasBeenFound = 1;
                }
```

```
                //printf("%s\n", dir->d_name);
                char *newPath = concat(path, "/");
                newPath = concat(newPath, dir->d_name);
                //printf("%s\n", newPath);
                //Recursively goes into the directory and opens it, search for fileName
                int result = combDirectoryForFile(newPath, newPath, fileName, 1);
                result--;
            }
        }
        closedir(d);
    }
    else{
        //directory DNE, returns failure
        if(occur == 0){
            printf("%s: No such file or directory\n", dirName);
        }
        return(2);
    }
    return(hasBeenFound);
}

int main(int argc, char **argv){
    // Needs 0 = functionName, 1 = fileToFind, 2+ = directoriesToBeSearched
    int result = 0;
    if(argc > 2){
        int i = 0;
        for(i = 2; i < argc; i++){
            result = combDirectoryForFile(argv[i], argv[i], argv[1], 0);
        }
    }
    else{
        printf("usage: findname name dir ...\n");
        return(1);
    }
    return(result);
}

```

Assignment 3

```
/*
 * fsh.c - the Feeble SHell.
 */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/stat.h>
#include "fsh.h"
#include "parse.h"
#include "error.h"
#include <string.h>

int showprompt = 1;
int laststatus = 0;  /* set by anything which runs a command */
extern char **environ;

int main(int argc, char **argv)
{
    char buf[1000];
    struct parsed_line *p;
    extern void execute(struct parsed_line *p);
    // Options taken
    int opts = 0;
    int opt = 0;
    int tty = 0; //Is from terminal (i)
    int ech = 0; // Is to be echoed (v)
    int nxt = 0; // Is taking the next input instead of from stdin (c)
    const size_t line_size = 1000;
    char *cmd = malloc(line_size);
    while ((opt = getopt (argc, argv, "vic:")) != -1){
        opts++;
        switch(opt){
            case 'v':
                //v;
                ech = 1;
                break;
            case 'i':
```

```
 42                  //i;
 43                  tty = 1;
 44                  break;
 45              case 'c':
 46                  //c;
 47                  nxt = 1;
 48                  cmd = optarg;
 49                  break;
 50          }
 51      }
 52      if(nxt){
 53          // Process the next string as the command (-c flag)
 54          // Execute 'cmd'
 55          if ((p = parse(cmd))) {
 56              execute(p);
 57              freeparse(p);
 58          }
 59          return(laststatus);
 60      }
 61      // If a filename is given as an argument
 62      else if(argc == 2+opts){
 63          FILE *fp;
 64          fp = fopen(argv[opts+1],"r");
 65          if (fp == NULL){
 66              printf("%s: No such file or directory\n", argv[opts+1]);
 67              exit(1);
 68          }
 69          else{
 70              // Get lines from file and run them in a while loop
 71              char* line = malloc(line_size);
 72              while (fgets(line, line_size, fp) != NULL)  {
 73                  if (showprompt){
 74                  // If it is terminal input or if the -i option is specified (-i f
lag)
 75                      if(tty == 1){
 76                          printf("$ ");
 77                      }
 78                  }
 79                  if(ech){
 80                      // Echo each line as they are processed (-v flag)
 81                      printf("%s", line);
 82                  }
 83                  if ((p = parse(line))) {
 84                      execute(p);
 85                      freeparse(p);
 86                  }
 87              }
 88              free(line);
 89          }
 90          fclose(fp);
 91          return(laststatus);
 92      }
 93      if(argc < (2+opts)){
 94          while (1) {
 95              if (showprompt){
 96                  // If it is terminal input or if the -i option is specified (-i f
lag)
 97                  if(tty == 1 || isatty(fileno(stdin))){
 98                      printf("$ ");
 99                  }
100              }
101              if (fgets(buf, sizeof buf, stdin) == NULL)
102                  break;
103              if(ech){
104                  // Echo each line as they are processed (-v flag)
105                  printf("%s", buf);
106              }
107              if ((p = parse(buf))) {
108                  execute(p);
109                  freeparse(p);
110              }
111          }
112      }
113      else{
114          fprintf(stderr, "usage: ./fsh [-i] [-v] [{file | -c command}]\n");
115          exit(1);
116      }
117      return(laststatus);
118  }
```

```
119
120
121  void execute(struct parsed_line *p)
122  {
123      int status;
124      extern void execute_one_subcommand(struct parsed_line *p);
125
126      fflush(stdout);
127      switch (fork()) {
128      case -1:
129          perror("fork");
130          laststatus = 127;
131          break;
132      case 0:
133          /* child */
134          execute_one_subcommand(p);
135          break;
136      default:
137          /* parent */
138          wait(&status);
139          laststatus = status >> 8;
140      }
141  }
142
143
144  /*
145   * execute_one_subcommand():
146   * Do file redirections if applicable, then [you can fill this in...]
147   * Does not return, so you want to fork() before calling me.
148   */
149  void execute_one_subcommand(struct parsed_line *p)
150  {
151      // If piping and a next exists (length of 2)
152      if(p->pl && p->pl->next){
153          int pipefd[2];
154
155          /* get a pipe (buffer and fd pair) from the OS */
156          if (pipe(pipefd)) {
157              perror("pipe");
158              exit(127);
159          }
160
161          /* We are the child process, but since we have TWO commands to exec we
162           * need to have two disposable processes, so fork again */
163          switch (fork()) {
164          case -1:
165              perror("fork");
166              exit(127);
167          case 0:
168              /* child */
169              /* do redirections and close the wrong side of the pipe */
170              if (p->inputfile) {
171                  close(0);
172                  if (open(p->inputfile, O_RDONLY, 0) < 0) {
173                      perror(p->inputfile);
174                      exit(1);
175                  }
176              }
177              if (p->outputfile) {
178                  close(1);
179                  if (open(p->outputfile, O_WRONLY|O_CREAT|O_TRUNC, 0666) < 0) {
180                      perror(p->outputfile);
181                      exit(1);
182                  }
183                  if(p->output_is_double){
184                      dup2(1,2);
185                  }
186              }
187              close(pipefd[0]);  /* the other side of the pipe */
188              dup2(pipefd[1], 1);  /* automatically closes previous fd 1 */
189              close(pipefd[1]);  /* cleanup */
190              /* exec ls */
191              // Process step 5, command execution and stat() calling on file locat
ions
192              if (p->pl){
193                  struct stat *buf;
194                  buf = malloc(sizeof(struct stat));
195                  // Checks if stderr needs to be redirected to stdout to be piped
into parent
```

```
196                 if(p->pl->next->isdouble){
197                     dup2(1,2);
198                 }
199                 if(strchr(p->pl->argv[0], '/') == NULL){
200                     char *bin, *usr, *cdir;
201                     bin = malloc (sizeof (char) * 1000);
202                     usr = malloc (sizeof (char) * 1000);
203                     cdir = malloc(sizeof (char) * 1000);
204                     strcpy (bin, "/bin/");
205                     strcpy (usr, "/usr/bin/");
206                     strcpy(cdir, "./");
207                     strcat(bin, p->pl->argv[0]);
208                     strcat(usr, p->pl->argv[0]);
209                     strcat(cdir, p->pl->argv[0]);
210                     //printf("%s:\n", p->pl->argv[0]);
211                     if(stat(bin, buf) == 0){
212                         laststatus = execve(bin, p->pl->argv, environ);
213                         if(laststatus == -1){
214                             fprintf(stderr, "%s\n", bin);
215                         }
216                     }
217                     else if(stat(usr, buf) == 0){
218                         laststatus = execve(usr, p->pl->argv, environ);
219                         if(laststatus == -1){
220                             fprintf(stderr,"%s\n",usr);
221                         }
222                     }
223                     else if(stat(cdir, buf) == 0){
224                         laststatus = execve(cdir, p->pl->argv, environ);
225                         if(laststatus == -1){
226                             fprintf(stderr,"%s\n",cdir);
227                         }
228                     }
229                     else{
230                         printf("%s: Command not found\n", p->pl->argv[0]);
231                         exit(1);
232                     }
233                 }
234                 else{
235                     if(stat(p->pl->argv[0], buf) == 0){
236                         laststatus = execve(p->pl->argv[0], p->pl->argv, environ)
;
237                         if(laststatus == -1){
238                             fprintf(stderr,"%s\n",p->pl->argv[0]);
239                         }
240                     }
241                 }
242             }
243             else{
244                 perror("p->pl is null");
245                 exit(1);
246             }
247             exit(126);
248         default:
249             /* parent */
250             /* do redirections and close the wrong side of the pipe */
251             if (p->inputfile) {
252                 close(0);
253                 if (open(p->inputfile, O_RDONLY, 0) < 0) {
254                     perror(p->inputfile);
255                     exit(1);
256                 }
257             }
258             if (p->outputfile) {
259                 close(1);
260                 if (open(p->outputfile, O_WRONLY|O_CREAT|O_TRUNC, 0666) < 0) {
261                     perror(p->outputfile);
262                     exit(1);
263                 }
264                 if(p->output_is_double){
265                     dup2(1,2);
266                 }
267             }
268             close(pipefd[1]);  /* the other side of the pipe */
269             dup2(pipefd[0], 0);  /* automatically closes previous fd 0 */
270             close(pipefd[0]);  /* cleanup */
271             /* exec tr */
272             if (p->pl->next){
273                 struct stat *buf;
```

```
274                 buf = malloc(sizeof(struct stat));
275                 if(strchr(p->pl->next->argv[0], '/') == NULL){
276                     char *bin, *usr, *cdir;
277                     bin = malloc (sizeof (char) * 1000);
278                     usr = malloc (sizeof (char) * 1000);
279                     cdir = malloc(sizeof (char) * 1000);
280                     strcpy (bin, "/bin/");
281                     strcpy (usr, "/usr/bin/");
282                     strcpy(cdir, "./");
283                     strcat(bin, p->pl->next->argv[0]);
284                     strcat(usr, p->pl->next->argv[0]);
285                     strcat(cdir, p->pl->next->argv[0]);
286                     //printf("%s:\n", p->pl->argv[0]);
287                     if(stat(bin, buf) == 0){
288                         laststatus = execve(bin, p->pl->next->argv, environ);
289                         if(laststatus == -1){
290                             fprintf(stderr,"%s\n",bin);
291                         }
292                     }
293                     else if(stat(usr, buf) == 0){
294                         laststatus = execve(usr, p->pl->next->argv, environ);
295                         if(laststatus == -1){
296                             fprintf(stderr,"%s\n",usr);
297                         }
298                     }
299                     else if(stat(cdir, buf) == 0){
300                         laststatus = execve(cdir, p->pl->next->argv, environ);
301                         if(laststatus == -1){
302                             fprintf(stderr,"%s\n",cdir);
303                         }
304                     }
305                     else{
306                         printf("%s: Command not found\n", p->pl->next->argv[0]);
307                         exit(1);
308                     }
309                 }
310                 else{
311                     if(stat(p->pl->next->argv[0], buf) == 0){
312                         laststatus = execve(p->pl->next->argv[0], p->pl->next->ar
gv, environ);
313                         if(laststatus == -1){
314                             fprintf(stderr,"%s\n",p->pl->next->argv[0]);
315                         }
316                     }
317                 }
318             }
319             exit(125);
320         }
321     }
322     // Else if not piping
323     else{
324         if (p->inputfile) {
325             close(0);
326             if (open(p->inputfile, O_RDONLY, 0) < 0) {
327                 perror(p->inputfile);
328                 exit(1);
329             }
330         }
331         if (p->outputfile) {
332             close(1);
333             if (open(p->outputfile, O_WRONLY|O_CREAT|O_TRUNC, 0666) < 0) {
334                 perror(p->outputfile);
335                 exit(1);
336             }
337             if(p->output_is_double){
338                 dup2(1,2);
339             }
340         }
341         // Process step 5, command execution and stat() calling on file locations
342         if (p->pl){
343             struct stat *buf;
344             buf = malloc(sizeof(struct stat));
346            if(strchr(p->pl->argv[0], '/') == NULL){
347                 char *bin, *usr, *cdir;
348                 bin = malloc (sizeof (char) * 1000);
349                 usr = malloc (sizeof (char) * 1000);
350                 cdir = malloc(sizeof (char) * 1000);
351                 strcpy (bin, "/bin/");
352                 strcpy (usr, "/usr/bin/");
```

```
 353                 strcpy(cdir, "./");
 354                 strcat(bin, p->pl->argv[0]);
 355                 strcat(usr, p->pl->argv[0]);
 356                 strcat(cdir, p->pl->argv[0]);
 357                 //printf("%s:\n", p->pl->argv[0]);
 358                 if(stat(bin, buf) == 0){
 359                     laststatus = execve(bin, p->pl->argv, environ);
 360                     if(laststatus == -1){
 361                         fprintf(stderr,"%s\n",bin);
 362                     }
 363                 }
 364                 else if(stat(usr, buf) == 0){
 365                     laststatus = execve(usr, p->pl->argv, environ);
 366                     if(laststatus == -1){
 367                         fprintf(stderr,"%s\n",usr);
 368                     }
 369                 }
 370                 else if(stat(cdir, buf) == 0){
 371                     laststatus = execve(cdir, p->pl->argv, environ);
 372                     if(laststatus == -1){
 373                         fprintf(stderr,"%s\n",cdir);
 374                     }
 375                 }
 376                 else{
 377                     printf("%s: Command not found\n", p->pl->argv[0]);
 378                     exit(1);
 379                 }
 380             }
 381             else{
 382                 if(stat(p->pl->argv[0], buf) == 0){
 383                     laststatus = execve(p->pl->argv[0], p->pl->argv, environ);
 384                     if(laststatus == -1){
 385                         fprintf(stderr,"%s\n",p->pl->argv[0]);
 386                     }
 387                 }
 388             }
 389         }
 390         else{
 391             fprintf(stderr,"p->pl is null");
 392             exit(1);
 393         }
 394     }
 395  }
```

ASSIGNMENT 4

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include "protocol.h"
#include "mud.h"
#include "lang.h"
#include "map.h"
#include "things.h"
#include "util.h"
static char *host = "localhost";
static int port = DEFAULT_PORT;
int serverfd;
char buf[200];  /* communication from server */
int bytes_in_buf = 0;
int mylocation;
int saw_inventory_something;
static struct namelist {
    char *name;
    int id;
    struct namelist *next;
} *names = NULL;
static void parseargs(int argc, char **argv);
static void connect_to_server();
static void gethandle();
static void do_say();
static void do_something();
static void do_something_server(char *wherenewline);
static void docmd(char **cmd);
static void call_with_arg(void (*f)(int), char *arg, char *expln, char *cmdname);
```

```
static void get(int obj);
static void drop(int obj);
static void poke(int obj);
static void go(int dir);
static void help();
static void loc(int place);
static void here(int id);
static void arrived(int id);
static void departed(int id);
static void pokedby(int id);
static void startup_checks();
static void storename(int id, char *name);
static void removename(int id);
static char *find_name(int id);
static char **explode(char *s);
static int parsenumber(char *s);
int main(int argc, char **argv){
    startup_checks();
    parseargs(argc, argv);
    connect_to_server();
    gethandle();
    while (1)
        do_something();
}
static void parseargs(int argc, char **argv)
{
    int status = 0;
    if (argc > 1)
        host = argv[1];
    if (argc > 2)
        if ((port = atoi(argv[2])) < 0)
            status = 1;
    if (argc > 3 || status) {
        fprintf(stderr, "usage: %s [hostname [portnum]]\n", argv[0]);
        exit(1);
    }
}
static void connect_to_server(){
    struct hostent *hp;
    struct sockaddr_in r;
    char *q;
    int len, server_protocol, server_nplaces, server_nthings;

    if ((hp = gethostbyname(host)) == NULL) {
        fprintf(stderr, "%s: no such host\n", host);
        exit(1);
    }
    if (hp->h_addr_list[0] == NULL || hp->h_addrtype != AF_INET) {
        fprintf(stderr, "%s: not an internet protocol host name\n", host);
        exit(1);
    }
    if ((serverfd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        perror("socket");
        exit(1);
    }
    memset(&r, '\0', sizeof r);
    r.sin_family = AF_INET;
    memcpy(&r.sin_addr, hp->h_addr_list[0], hp->h_length);
    r.sin_port = htons(port);

    if (connect(serverfd, (struct sockaddr *)&r, sizeof r) < 0) {
        perror("connect");
        exit(1);
    }
    /* read banner line */
    while (!(q = memnewline(buf, bytes_in_buf))) {
        if ((len = read(serverfd, buf+bytes_in_buf, sizeof buf - bytes_in_buf))
                == 0) {
            printf("server dropped the connection\n");
            exit(0);
        } else if (len < 0) {
            perror("read");
            exit(1);
        }
        bytes_in_buf += len;
    }
    *q = '\0';
    if (sscanf(buf, "%d%d%d",
                    &server_protocol, &server_nplaces, &server_nthings)
```

```
            != 3) {
        fprintf(stderr, "can't parse server banner line\n");
        exit(0);
    }
    if (server_protocol != PROTOCOL_VERSION) {
        fprintf(stderr, "protocol version mismatch\n");
        exit(0);
    }
    if (server_nplaces != nplaces || server_nthings != lang_nthings) {
        fprintf(stderr, "server has a different map than we do\n");
        exit(0);
    }
    /* remove the banner line from the already-read buffer */
    len = q - buf + 1;
    if (bytes_in_buf > len && (buf[len] == '\r' || buf[len] == '\n'))
        len++;
    bytes_in_buf -= len;
    memmove(buf, buf + len, bytes_in_buf);
}
static void send_string(char *s){
    int len = strlen(s);
    if (write(serverfd, s, len) != len)
        perror("write");
}
static void gethandle(){
    char buf[MAXHANDLE + 3], *p;
    do {
        printf("%s: ", lang_handle_request);
        if (fgets(buf, MAXHANDLE+1, stdin) == NULL)
            exit(0);
        if ((p = strchr(buf, '\n')))
            *p = '\0';
    } while (buf[0] == '\0');
    strcat(buf, "\r\n");
    send_string(buf);
}
static void do_say(char *buf){
    char *sendmsg, *saidID, *msg;
    sendmsg = malloc(sizeof(char) * 5);
    saidID = malloc(sizeof(char) * 100);
    msg = malloc(sizeof(char) * 1000);
    strcpy(msg, "");
    char *space = " ";
    sendmsg = strtok(buf, space);
    // Useless, but make it be  placeholder I guess...?
    saidID = sendmsg;
    saidID = strtok(NULL, space);
    // From the FD onwards, is the message
    char *token;
    int counter = 0;
    while( token != NULL ){
        token = strtok(NULL, space);
        if(token != NULL){
            // Re-inserts the space that was parsed out
            if(counter != 0){
                strcat(msg, " ");
            }
            else{
                counter = 1;
            }
            // Appends each token into it
            strcat(msg, token);
        }
    }
    int userSaidID = atoi(saidID);
    char *userName;
    userName = malloc(sizeof(char) * 1000);
    userName = find_name(userSaidID);
    // Writes something to the effect of USERNAME says:
    printf(lang_says_format, userName);
    // Writes the message
    printf(" %s\n", msg);
}
static void do_something()
{
    char *q;
    if ((q = memnewline(buf, bytes_in_buf))) {
        do_something_server(q);
    } else {
```

```
        fd_set fdlist;
        FD_ZERO(&fdlist);
        FD_SET(0, &fdlist);
        FD_SET(serverfd, &fdlist);
        if (select(serverfd+1, &fdlist, NULL, NULL, NULL) < 0) {
            perror("select");
        } else {
            if (FD_ISSET(serverfd, &fdlist)) {
                int n = read(serverfd, buf+bytes_in_buf, sizeof buf - bytes_in_buf);
                if (n == 0) {
                    printf("\nserver dropped the connection\n");
                    exit(0);
                } else if (n < 0) {
                    perror("read");
                    exit(1);
                } else {
                    bytes_in_buf += n;
                    if ((q = memnewline(buf, bytes_in_buf)))
                        do_something_server(q);
                }
            }
            if (FD_ISSET(0, &fdlist)) {
                char buf[200];
                if (fgets(buf, sizeof buf, stdin) == NULL)
                    exit(0);
                // Checks if someone said something and sends the text to the server
                int len = strlen(lang_say);

                char *totmsg;
                totmsg = malloc(sizeof(char) * 1000);
                strcat(totmsg, "say ");
                strcat(totmsg, buf+len+1);
                if(strncmp(buf, lang_say, len) == 0){
                    if(buf[strlen(lang_say)+1] == 0){
                        printf(lang_say_explain, lang_say);
                    }
                    else
                        send_string(totmsg);
                }
                else{
                    docmd(explode(buf));
                }
            }
        }
    }
}
static void do_something_server(char *whererenewline)
{    int n;
    *whererenewline = '\0';
    if (match_arg(buf, "loc", &n)) {
        loc(n);
    } else if (match_arg(buf, "here", &n)) {
        here(n);
    } else if (match_arg(buf, "arr", &n)) {
        arrived(n);
    } else if (match_arg(buf, "dep", &n)) {
        departed(n);
    } else if (match_arg(buf, "poked", &n)) {
        pokedby(n);
    } else if (strcmp(buf, "ib") == 0) {
        saw_inventory_something = 0;
        printf("%s\n", lang_inv_heading);
    } else if (match_arg(buf, "i", &n)) {
        saw_inventory_something = 1;
        printf("    %s (#%d)\n", lang_thing[n], n);
    } else if (strcmp(buf, "ie") == 0) {
        if (!saw_inventory_something)
            printf("%s\n", lang_inv_nothing);
    } else if (strncmp(buf, "said", 4) == 0) {
        do_say(buf);
    } else if (strcmp(buf, "ok") == 0) {
        printf("%s\n", lang_ok);
    } else if (strcmp(buf, "ng") == 0) {
        printf("%s\n", lang_get_nosuch);
    } else if (strcmp(buf, "nd") == 0) {
        printf("%s\n", lang_drop_nosuch);
    } else if (strcmp(buf, "np") == 0) {
        printf("%s\n", lang_get_nosuch);
    } else if (match_arg(buf, "name", &n)) {
```

```
        char *p;
        if ((p = strchr(buf, ' ')) == NULL
                || (p = strchr(p + 1, ' ')) == NULL)
            fprintf(stderr, "error: malformed 'name' from server\n");
        else
            storename(n, p + 1);
    } else if (match_arg(buf, "quit", &n)) {
        removename(n);
    } else if (strncmp(buf, "error ", 6) == 0) {
        printf("error from server: %s\n", buf + 6);
    } else {
        fprintf(stderr, "unexpected data from server: %s\n", buf);
    }
    n = wherenewline - buf;
    n++;
    if (bytes_in_buf > n && (buf[n] == '\r' || buf[n] == '\n'))
        n++;
    bytes_in_buf -= n;
    memmove(buf, buf + n, bytes_in_buf);
}
static void docmd(char **cmd){
    int i;
    if (cmd[0] == NULL) {
        help();
        return;
    }
    if (cmd[1] && cmd[2]) {
        printf("%s\n", lang_toolong);
        help();
        return;
    }
    if (strcmp(cmd[0], lang_look[0]) == 0
            || strcmp(cmd[0], lang_look[1]) == 0) {
        send_string("descr\r\n");
        return;
    }
    if (strcmp(cmd[0], lang_inv[0]) == 0
            || strcmp(cmd[0], lang_inv[1]) == 0) {
        send_string("inv\r\n");
        return;
    }
    if (strcmp(cmd[0], lang_get) == 0) {
        call_with_arg(get, cmd[1], lang_getdrop_explain, lang_get);
        return;
    }
    if (strcmp(cmd[0], lang_drop) == 0) {
        call_with_arg(drop, cmd[1], lang_getdrop_explain, lang_drop);
        return;
    }
    if (strcmp(cmd[0], lang_poke) == 0) {
        if (cmd[1] && cmd[1][0] == '-')
            cmd[1]++;
        call_with_arg(poke, cmd[1], lang_poke_explain, lang_poke);
        return;
    }
    for (i = 0; i < 6; i++) {
        if (strcmp(cmd[0], lang_directions[i][0]) == 0
                || strcmp(cmd[0], lang_directions[i][1]) == 0) {
            go(i);
            return;
        }
    }
    /* accept standard command "l" in any language, unless it is assigned
     * another meaning */
    if (strcmp(cmd[0], "l") == 0) {
        send_string("descr\r\n");
        return;
    }
    printf("%s\n", lang_huh);
    help();
}
static void call_with_arg(void (*f)(int), char *arg, char *expln, char *cmdname){
    int argnum;
    if (arg == NULL)
        printf(expln, cmdname);
    else if ((argnum = parsenumber(arg)) >= 0)
        (*f)(argnum);
}
static void get(int obj){
```

```
   if (obj >= 0 && obj < n_thing_place) {
        char buf[40];
        sprintf(buf, "get %d\r\n", obj);
        send_string(buf);
    } else {
        printf("%s\n", lang_get_nosuch);
    }
}


static void drop(int obj){
    if (obj >= 0 && obj < n_thing_place) {
        char buf[40];
        sprintf(buf, "drop %d\r\n", obj);
        send_string(buf);
    } else {
        printf("%s\n", lang_drop_nosuch);
    }
}
static void poke(int obj){
    char buf[40];
    sprintf(buf, "poke %d\r\n", obj);
    send_string(buf);
}

static void go(int dir){
    if (places[mylocation].exit_loc[dir] >= 0) {
        char buf[30];
        sprintf(buf, "go %d\r\n", places[mylocation].exit_loc[dir]);
        send_string(buf);
    } else {
        printf("%s\n", lang_nosuchexit);
    }
}
static void help(){
    int i;
    printf("%s %s %s %s %s %s", lang_commandlist, lang_get, lang_drop, lang_poke, lang_say, lang_inv[0]);
    for (i = 0; i < 6; i++)
        printf(" %s", lang_directions[i][0]);
    printf("\n");
}
static void loc(int place){
    int i;
    mylocation = place;
    printf("\n%s %s.\n", lang_youat, lang_place_title[place]);
    if (lang_place_detail[place])
        printf("%s\n", lang_place_detail[place]);
    printf("%s:\n", lang_youcango);
    for (i = 0; i < NDIRECTIONS; i++)
        if (places[place].exit_loc[i] >= 0)
            printf("    %s %s: %s\n", lang_directions[i][0], lang_go_to,
                    lang_place_title[places[place].exit_loc[i]]);
}
static void here(int id)
{
    if (id >= 0) {
        printf(lang_thereis_format, lang_thing[id]);
    } else {
        char *p = find_name(id);
        if (p)
            printf(lang_thereis_format, p);
        else
            printf("error: unidentified id");
    }
    printf(" (#%d)\n", id);
}
static void arrived(int id){
    if (id >= 0) {
        printf("%s %s", lang_thing[id], lang_arrived);
    } else {
        char *p = find_name(id);
        if (p)
            printf("%s %s", p, lang_arrived);
        else
            printf("error: unidentified id");
    }
    printf(" (#%d)\n", id);
}
static void departed(int id){
```

```
    if (id >= 0) {
        printf("%s %s", lang_thing[id], lang_departed);
    } else {
        char *p = find_name(id);
        if (p)
            printf("%s %s", p, lang_departed);
        else
            printf("error: unidentified id");
    }
    printf(" (#%d)\n", id);
}
static void pokedby(int id){
    if (id >= 0) {
        printf("%s %s", lang_poked_by, lang_thing[id]);
    } else {
        char *p = find_name(id);
        if (p)
            printf("%s %s", lang_poked_by, p);
        else
            printf("error: unidentified id");
    }
    printf(" (#%d)\n", id);
}
static void startup_checks(){
    map_setup();
    if (lang_nplaces != nplaces) {
        fprintf(stderr, "mismatch in 'place' lists\n");
        exit(1);
    }
    if (lang_nthings != n_thing_place) {
        fprintf(stderr, "mismatch in 'thing' lists\n");
        exit(1);
    }
}
static void storename(int id, char *name){
    struct namelist *p;
    if ((p = malloc(sizeof(struct namelist))) == NULL
            || (p->name = malloc(strlen(name) + 1)) == NULL) {
        fprintf(stderr, "out of memory!\n");
        exit(1);
    }
    strcpy(p->name, name);
    p->id = id;
    p->next = names;
    names = p;
}
static void removename(int id){
    struct namelist **p;
    for (p = &names; *p && (*p)->id != id; p = &((*p)->next))
        ;
    if (*p) {
        struct namelist *old = *p;
        free((*p)->name);
        *p = (*p)->next;
        free(old);
    }
}
static char *find_name(int id){
    struct namelist *p;
    for (p = names; p && p->id !=id; p = p->next)
        ;
    return(p ? p->name : NULL);
}
static char **explode(char *s){
#   define MAXTOKENS 5
    static char *retval[MAXTOKENS + 1];
    int i;
    static char sep[] = " \t\n\r";

    if ((retval[0] = strtok(s, sep)) == NULL)
        return(retval);
    for (i = 1; i < MAXTOKENS && (retval[i] = strtok(NULL, sep)); i++) {
        if (lang_use_tolower) {
            char *p;
            for (p = retval[i]; *p; p++)
                if (isascii(*p) && isupper(*p))
                    *p = tolower(*p);
        }
    }
```

```
    retval[i] = NULL;
    return(retval);
}
static int parsenumber(char *s){
    if (!isalldigits(s)) {
        printf("%s\n", lang_req_obj_number);
        return(-1);
    }
    return(atoi(s));
}
-------------- SERVER.C-------------------------
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include "protocol.h"
#include "nplaces.h"
#include "things.h"
#define nthings n_thing_place
#include "util.h"
static int port = DEFAULT_PORT;
static int listenfd;
static char hello[100];
static int hellolen;
struct client {
    int fd;
    int id;  /* unique negative integer */
    struct in_addr ipaddr;
    int loc;  /* where this person is */
    char handle[MAXHANDLE + 1];  /* zero-terminated */
    char buf[200];  /* command-line or command-line in progress */
    int bytes_in_buf;  /* how many data bytes already read in buf */
    struct client *next;  /* next item in linked list of clients */
} *clientlist = NULL;
static void parseargs(int argc, char **argv);
static void newclient(int fd, struct sockaddr_in *r);
static void removeclient(struct client *p);
static void describe(struct client *p);
static void do_set_handle(struct client *p, int len);
static void do_inv(struct client *p);
static void do_go(struct client *p, int place);
static void do_get(struct client *p, int id);
static void do_drop(struct client *p, int id);
static void do_poke(struct client *p, int id);
static void do_say(struct client *p, int len);
static void do_something(struct client *p, char *wherenewline);
static void send_arrived(int loc, int thing, struct client *donttell);
static void send_departed(int loc, int thing, struct client *donttell);
static void send_string(int fd, char *s);
int main(int argc, char **argv){
    // Parses the initial arguments
    parseargs(argc, argv);
    // Server socket declaration
    int clientfd, maxfd;
    socklen_t len;
    struct sockaddr_in r, q;
    fd_set fdlist;
    if ((listenfd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        perror("socket");
        return(1);
    }
    memset(&r, '\0', sizeof r);
    r.sin_family = AF_INET;
    r.sin_addr.s_addr = INADDR_ANY;
    // Gives the default port or -p argument
    r.sin_port = htons(port);
    if (bind(listenfd, (struct sockaddr *)&r, sizeof r) < 0) {
        perror("bind");
        return(1);
    }
    if (listen(listenfd, 5)) {
        perror("listen");
        return(1);
```

```
}
// Select() loop
FD_ZERO(&fdlist);
FD_SET(listenfd, &fdlist);
maxfd = listenfd;
while(1){
    // Resets FDlist and adds the server back to FDList
    FD_ZERO(&fdlist);
    FD_SET(listenfd, &fdlist);
    // Adds all the clientFDs into FDlist
    struct client *curr = clientlist;
    while(curr != NULL){
        clientfd = curr->fd;
        FD_SET(clientfd, &fdlist);
        curr = curr->next;
    }
    switch (select(maxfd + 1, &fdlist, NULL, NULL, NULL)) {
        case 0:
            printf("timeout happened\n");
            break;
        case -1:
            perror("select");
            break;
        default:
            // Accepting new clients on the listening port
            if (FD_ISSET(listenfd, &fdlist)){
                len = sizeof q;
                // Accepts the connection
                if ((clientfd = accept(listenfd, (struct sockaddr *)&q, &len)) < 0) {
                    perror("accept");
                    return(1);
                }
                // Adds the clientfd into the list of filedescriptors
                FD_SET(clientfd, &fdlist);
                // Increments how many filedescriptors to loop through
                if (clientfd > maxfd)
                    maxfd = clientfd;
                // Commences the newclient protocol
                newclient(clientfd, &q);
                struct client *traverse = clientlist;
                for(traverse = clientlist; traverse; traverse = traverse->next){
                    if(clientfd == traverse->fd){
                        // puts the banner to the new connection, initializing it
                        char *buf;
                        buf = malloc(sizeof(struct client) * 1000);
                        sprintf(buf, "%d %d %d\r\n", PROTOCOL_VERSION, NPLACES, n_thing_place);
                        send_string(clientfd, buf);
                    }
                }
            }
            else{
                // Processsing commands from connected peers
                struct client *traverse = clientlist;
                // If the connected peer exists in the fd_list
                for(traverse = clientlist; traverse; traverse = traverse->next){
                    if(FD_ISSET(traverse->fd, &fdlist)){
                        if (!(memnewline(traverse->buf, traverse->bytes_in_buf))) {
                            int len = 0;
                            if ((len = read(traverse->fd, traverse->buf+traverse->bytes_in_buf,
                                            sizeof (traverse->buf-traverse->bytes_in_buf))) == 0) {
                                printf("client dropped the connection\n");
                                exit(0);
                            } else if (len < 0) {
                                perror("read");
                                exit(1);
                            }
                            traverse->bytes_in_buf += len;
                        }
                        if (memnewline(traverse->buf, traverse->bytes_in_buf)){
                         do_something(traverse, memnewline(traverse->buf,traverse->bytes_in_buf));
                        }
                    }
                }
            }
    }
}
// The Banner message from the server
//sprintf(buf, "%d %d %d\r\n", PROTOCOL_VERSION, NPLACES, n_thing_place);
/*
```

```
    //...
    do_something(p, s);
        -> where 'p' is a pointer to struct client,
           and s is the return value of memnewline()
    //...
    */
    return 0;
}
static void parseargs(int argc, char **argv){
    int c, status = 0;
    while ((c = getopt(argc, argv, "p:")) != EOF) {
        switch (c) {
        case 'p':
            port = atoi(optarg);
            break;
        default:
            status++;
        }
    }
    if (status || optind != argc) {
        fprintf(stderr, "usage: %s [-p port]\n", argv[0]);
        exit(1);
    }
}
static void newclient(int fd, struct sockaddr_in *r){
    struct client *p;
    static int lastid = 0;
    printf("connection from %s\n", inet_ntoa(r->sin_addr));
    if (write(fd, hello, hellolen) != hellolen) {
        close(fd);
        printf("%s didn't even listen to the banner!\n",
                inet_ntoa(r->sin_addr));
        return;
    }
    if ((p = malloc(sizeof(struct client))) == NULL) {
        /* very unlikely */
        fprintf(stderr, "out of memory in adding new client!\n");
        close(fd);
        return;
    }
    p->fd = fd;
    p->id = --lastid;
    p->ipaddr = r->sin_addr;
    p->bytes_in_buf = 0;
    p->loc = INITIAL_LOC;
    p->handle[0] = '\0';  /* indicates that it hasn't been input yet */
    p->next = clientlist;
    clientlist = p;
}
static void removeclient(struct client *p){
    struct client **pp;
    int oldid, oldloc, i;
    char buf[30];
    printf("disconnecting client %s\n", inet_ntoa(p->ipaddr));
    close(p->fd);
    oldid = p->id;
    oldloc = p->loc;
    /* remove */
    for (pp = &clientlist; *pp && *pp != p; pp = &((*pp)->next))
        ;
    if (*pp == NULL) {
        fprintf(stderr, "very odd -- I can't find that client\n");
    } else {
        *pp = (*pp)->next;
        free(p);
    }
    /* drop all possessions */
    for (i = 0; i < nthings; i++) {
        if (thing_place[i] == oldid) {
            thing_place[i] = oldloc;
            send_arrived(oldloc, i, NULL);
        }
    }
    /* tell everyone this person has quit */
    send_departed(oldloc, oldid, NULL);
    sprintf(buf, "quit %d\r\n", oldid);
    for (p = clientlist; p; p = p->next)
        send_string(p->fd, buf);
}
```

```
static void describe(struct client *p)
{
    int i;
    char buf[40];
    struct client *q;

    sprintf(buf, "loc %d\r\n", p->loc);
    send_string(p->fd, buf);
    for (i = 0; i < nthings; i++) {
        if (thing_place[i] == p->loc) {
            sprintf(buf, "here %d\r\n", i);
            send_string(p->fd, buf);
        }
    }
    for (q = clientlist; q; q = q->next) {
        if (q != p && q->loc == p->loc) {
            sprintf(buf, "here %d\r\n", q->id);
            send_string(p->fd, buf);
        }
    }
}
static void do_say(struct client*p, int len){
    struct client *traverse = clientlist;
    for(traverse = clientlist; traverse; traverse = traverse->next){
        if(traverse->loc == p->loc){
            char *buf;
            char *curid;
            buf = malloc(sizeof(char) * 1000);
            curid = malloc(sizeof(char) * 1000);
            // casts the id to int for concat-ing to string
            sprintf(curid, "%d", p->id);
            strcat(buf, "said ");
            strcat(buf, curid);
            strcat(buf, " ");
            // Removes the say_ part before the text
            strcat(buf, p->buf+4);
            strcat(buf, "\r\n");
            printf("%s\n", buf);
            if(strlen(buf) == 4){
                printf("NO\n");
            }
            // Sends the information to the client to be processed
            send_string(traverse->fd, buf);
        }
    }
}
static void do_set_handle(struct client *p, int len)
{
    struct client *q;
    char buf[MAXHANDLE + 50];
    int i, c;

    if (len > MAXHANDLE) {
        fprintf(stderr, "handle is %d chars long, max %d\n", len, MAXHANDLE);
        removeclient(p);
        return;
    }
    /* copy characters, check for bad ones */
    for (i = 0; i < len; i++) {
        c = (p->buf[i] & 255);
        if (c < ' ' || (c >= 127 && c < 160)) {
            fprintf(stderr, "handle contains illegal character %d\n", c);
            removeclient(p);
            return;
        }
        p->handle[i] = c;
    }
    p->handle[len] = '\0';

    printf("set handle of fd %d to %s\n", p->fd, p->handle);

    /* tell everyone else about the newcomer */
    sprintf(buf, "name %d %s\r\n", p->id, p->handle);
    for (q = clientlist; q; q = q->next)
        send_string(q->fd, buf);
    /* tell the newcomer about everyone else */
    for (q = clientlist; q; q = q->next) {
        if (q != p) {
            sprintf(buf, "name %d %s\r\n", q->id, q->handle);
```

```
            send_string(p->fd, buf);
        }
    }
    /* move the newcomer to the initial location */
    send_arrived(p->loc, p->id, p);
    describe(p);
}
static void do_inv(struct client *p){
    int i;
    char buf[30];

    send_string(p->fd, "ib\r\n");
    for (i = 0; i < nthings; i++) {
        if (thing_place[i] == p->id) {
            sprintf(buf, "i %d\r\n", i);
            send_string(p->fd, buf);
        }
    }
    send_string(p->fd, "ie\r\n");
}
static void do_go(struct client *p, int place){
    if (place < 0 || place >= NPLACES) {
        fprintf(stderr, "invalid place %d from client fd %d\n",
                place, p->fd);
        send_string(p->fd, "invalid place number\r\n");
        return;
    }
    send_departed(p->loc, p->id, p);
    p->loc = place;
    send_arrived(p->loc, p->id, p);
    describe(p);
}
static void do_get(struct client *p, int id){
    if (id < 0 || id >= nthings) {
        fprintf(stderr, "invalid thing %d from client fd %d\n",
                id, p->fd);
        send_string(p->fd, "invalid thing number\r\n");
        return;
    }
    if (thing_place[id] != p->loc) {
        send_string(p->fd, "ng\r\n");
        return;
    }
    thing_place[id] = p->id;
    send_departed(p->loc, id, p);
    send_string(p->fd, "ok\r\n");
}
static void do_drop(struct client *p, int id){
    if (id < 0 || id >= nthings) {
        fprintf(stderr, "invalid thing %d from client fd %d\n",
                id, p->fd);
        send_string(p->fd, "invalid thing number\r\n");
        return;
    }
    if (thing_place[id] != p->id) {
        send_string(p->fd, "nd\r\n");
        return;
    }
    thing_place[id] = p->loc;
    send_arrived(p->loc, id, p);
    send_string(p->fd, "ok\r\n");
}
static void do_poke(struct client *p, int id){
    char buf[30];
    struct client *q;

    if (id > 0)
        id = -id;
    for (q = clientlist; q && q->id != id; q = q->next)
        ;
    if (!q || p->loc != q->loc) {
        send_string(p->fd, "np\r\n");
        return;
    } else {
        sprintf(buf, "poked %d\r\n", p->id);
        send_string(q->fd, buf);
        send_string(p->fd, "ok\r\n");
    }
}
```

```
/* there is a command in the buffer; do it */
static void do_something(struct client *p, char *whererenewline){
    int len, n;
    len = whererenewline - p->buf;
    *whererenewline = '\0';
    if (len == 0) {
        /* ignore blank lines */
    } else if (p->handle[0] == '\0') {
        do_set_handle(p, len);
    } else if(strncmp(p->buf, "say ", 4) == 0){
        do_say(p, len);
    } else if (strcmp(p->buf, "inv") == 0) {
        do_inv(p);
    } else if (strcmp(p->buf, "descr") == 0) {
        describe(p);
    } else if (match_arg(p->buf, "go", &n)) {
        do_go(p, n);
    } else if (match_arg(p->buf, "get", &n)) {
        do_get(p, n);
    } else if (match_arg(p->buf, "drop", &n)) {
        do_drop(p, n);
    } else if (match_arg(p->buf, "poke", &n)) {
        do_poke(p, n);
    } else {
        char buf[100];
        fprintf(stderr, "invalid command from fd %d: %.*s\n",
                p->fd, len, p->buf);
        sprintf(buf, "error invalid: %.*s\r\n", len, p->buf);
        send_string(p->fd, buf);
    }
    /* p->buf[len] was either \r or \n.  How about p->buf[len+1]? */
    len++;
    if (len < p->bytes_in_buf && (p->buf[len] == '\r' || p->buf[len] == '\n'))
        len++;
    p->bytes_in_buf -= len;
    memmove(p->buf, p->buf + len, p->bytes_in_buf);
}
static void send_arrived(int loc, int thing, struct client *donttell){
    struct client *q, *n;
    for (q = clientlist; q; q = n) {
        n = q->next;
        if (q != donttell && q->loc == loc) {
            char buf[100];
            sprintf(buf, "arr %d\r\n", thing);
            send_string(q->fd, buf);
        }
    }
}
static void send_departed(int loc, int thing, struct client *donttell){
    struct client *q, *n;
    for (q = clientlist; q; q = n) {
        n = q->next;
        if (q != donttell && q->loc == loc) {
            char buf[100];
            sprintf(buf, "dep %d\r\n", thing);
            send_string(q->fd, buf);
        }
    }
}
static void send_string(int fd, char *s){
    int len = strlen(s);
    if (write(fd, s, len) != len)
        perror("write");
}
```