EE 367L Lab 3

Due date: See laulima

The objective of this assignment is to gain experience with system calls (calls to operating system functions) and multiple processes. The system calls allow communication between processes. They use mechanisms of pipes and sockets.

Contents

A Mechanisms

Processes, pipes and sockets will be described.

A.1 Process

A *process* is a running program, which includes the executable program (or machine program), the data memory that it uses, and the CPU. A process is basically a *virtual computer with a single CPU* running a program. It is actually running on a physical CPU core, perhaps sharing the core with other processes.

A process has its own block of physical memory and consumes a certain amount of CPU processing bandwidth. More processes means more physical memory and CPU bandwidth is used.

An operating system (such as Linux) keeps track of its processes, each with its own ID number. If you go to wiliki and type "ps", it will display the processes running in your account, e.g.,

```
PID    TTY    TIME        CMD
24538  pts/4  00:00:00    bash
24570  pts/4  00:00:00    ps
```

Each row is a running process:

- PID: Process ID. You can use this ID to manage the process. For example, you can kill (terminate) a process with PID 24538 by typing "kill 24538".
- CMD: Program name.

Processes are created over time. The first process has the PID equal to zero, and subsequent processes have higher PIDs. In the example, the processes have PIDs over 24000. This implies that a lot of processes have been created.

The following are system calls that manage processes

- fork( ): This creates a process, which is referred to as a *child* process. The process that creates a child is called the *parent*.
  - When a process executes fork( ), it will create a child which is identical to itself. Both parent and child continue running; however, the returned value of fork( ) is different:
    - For the child, it is zero.
    - For the parent, it is the child's PID.
  - For example, suppose a parent process calls fork( ), which creates a child process with PID 423. Then the parent will receive a returned value of 423, while the child will receive a returned value of 0. In this way, the processes will know whether they are parent or child, e.g., they can check if the returned value is zero.
  - Note that a child process can create its own children, and becomes the parent of its children.

- exit( ): This exits (terminates) a process.

- wait( ): This will cause a parent process to wait until a child (any child) terminates.

- waitpid(int pid): This is similar to wait( )
  - If pid > 0, then it waits for the child process with the PID pid to terminate.
  - If pid = -1, then it waits for any child process to terminate.
  - If pid = 0, then it waits for any child process in the same group as the parent process.

Read the following web site about fork( )
http://linux.about.com/od/commands/l/blcmdl2_fork.htm

Related system calls are wait( ) and waitpid( )
http://linux.about.com/od/commands/l/blcmdl2_wait.htm

Read the following about exit( )
http://linux.die.net/man/2/exit

A common application of fork( ) is

```
main( )
{
pid = fork( );
if (pid < 0) { /* error */
  indicate an error occured
}
if (pid== 0) { /* child process */
   processing for the child
}
else { /* parent process */
   processing for the parent
}

}
```

The child doesn't do the same thing as the parent because

- the child will continue running in "if (pid == 0)..."
- while the parent will continue running in "else..."

The following is a more specific example application of fork( ):

```
while( 1 ) {
  task1( );
  task2( );
}
```

These tasks run in sequence rather than concurrently. By using fork( ), we can run the tasks concurrently.

```
while( 1) {
  if (fork( ) == 0) { /* child */
    task1( );
    exit(EXIT_SUCCESS); /* terminate the child.  You can also use "return" */
  }
  else { /* parent */
    task2( );
    wait( ); /* wait for child to terminate */
  }
}
```

Another application of fork( ) is to launch an executable such as a UNIX command, e.g., "cd", "cat", "rm", "ps", "grep" or "ls". These program can be run using the "exec( )" system call.
http://linux.die.net/man/3/exec

The following is an example of a process creating a child process, which will run “ls” (recall that “ls” is a UNIX command that will list the files and directories in the current directory). This is in an attached file “ls367.c”

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

void error(char *s);
char *data = "Some input data\n";

void main()
{
  int pid;
  int status;

  if ((pid=fork()) == 0) {   /* Child process */
    execl("/usr/bin/ls", "ls", (char *)NULL);
    error("Could not exec 'ls'");
  }

  /*  The following is in the parent process */
  wait(&status);
  printf("Spawned 'ls' is a child process at pid %d\n", pid);
  exit(0);
}

void error(char *s)
{
  perror(s);
  exit(1);
}
```

Note that the “execl” system call is executed in the child process. “execl” will call a Linux command, in this case “ls”. The operating system will completely replace the child process with a copy of “ls”, and then subsequently execute it. The “ls” program will output the contents of the current directory as a text string, which is output to the console.

The input parameters of execl are what you would normally type in command line, e.g.,

```
ls –l
```

or

```
cat filename
```

The first input parameter is a path to the system call, in this case “`/usr/bin/ls`”. Subsequent parameters are what you would normally type on the console, e.g., “ls”, -l”. The last parameter is a Null value, to indicate the end of the input parameters. For example

execl(/usr/bin/cat, “cat”, “filename”, (char *) NULL);

Note: there are other versions of execl such as exec and execlp. Look them up.

Also, note that when “ls” is running, its output goes to the console, where no further processing occurs. Instead, the output should go to the parent process for further processing. We can use pipes to do this, which are described next.

A.2 Pipes

Recall that processes behave like virtual computers that are running independently of each other. There are many applications that where it is necessary for the processes to cooperate. Then they must be able to communicate, i.e., exchange data.

A pipe is a memory buffer in the computer that multiple processes can access. Data is entered and retrieved from the pipe by using write( ) and read( ) system calls, respectively.

The following will create a pipe. Before you do this, you must create an int variable array to store the “file descriptor” of the pipe, which is the ID of the pipe.

```
int fd[2];   // You must create a 2-dimension int array of size 2
```

The following creates the pipe

```
pipe(fd);
```

Now

- fd[0] is the file descriptor of the read-end of the pipe
- fd[1] is the file descriptor of the write-end of the pipe

You can close the read-end using “close(fd[0])”, and close the write-end using “close(fd[1])”.

In many applications, a process will only use the write-end while another process will only use the read-end. Then these processes will close the end that it is not using.

The following comes from http://tldp.org/LDP/lpg/node11.html, which shows an example of a child process sending a character string to the parent process over a pipe.

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

int main(void)
{
        int     fd[2], nbytes;
        pid_t   childpid;
        char    string[] = "Hello, world!\n";
        char    readbuffer[80];

        pipe(fd);

        if((childpid = fork()) == -1)
        {
                perror("fork");
                exit(1);
        }

        if(childpid == 0)
        {
                /* Child process closes up input side of pipe */
                close(fd[0]);

                /* Send "string" through the output side of pipe */
                write(fd[1], string, (strlen(string)+1));
                exit(0);
        }
        else
        {
                /* Parent process closes up output side of pipe */
                close(fd[1]);

                /* Read in a string from the pipe */
                nbytes = read(fd[0], readbuffer, sizeof(readbuffer));
                printf("Received string: %s", readbuffer);
        }
        return(0);
}
```

Note that both “write” and “read” have three input parameters:

- File descriptor
- A reference to a byte array, such as a character array
- An integer value, which is the maximum length of bytes to be sent or received.

The returned valued is the actual number of bytes that were sent or received. This value is important since “write” or “read” may not transfer the maximum number of bytes. Also, note that the pipe was created before the fork( ) so that both parent and child have access to it.

To see how a pipe can be used with execl, see the the attached program pipe.c as an example. (pipe.c was found at http://stackoverflow.com/questions/70842/execute-program-from-within-a-c-program)

A.3 Socket

Pipes are good for processes in the same physical computer, but not for processes in different computers, say one in Honolulu and the other in New York. Then you must use network sockets.

A socket is more complicated since it goes through the Internet. The Internet uses a network protocol called TCP/IP. To identify a machine (computer) in the Internet, you need an IP address, a 32 bit number. The address is usually represented in the decimal-dot notation, e.g., 255.23.63.10 represent four bytes, each byte is in decimal notation. Within a machine there are tens of thousands of TCP "ports". Some of port numbers are well known, e.g., 80 is the port for web servers.

A network socket is a connection end-point. It is the combination IPAddress/Port#, where IPAddress is the IP address of the machine, and Port# is a TCP port number. Sockets have file descriptors just like pipes.

System calls "`send`" and "`recv`" are used to send and receive strings of bytes to a socket.

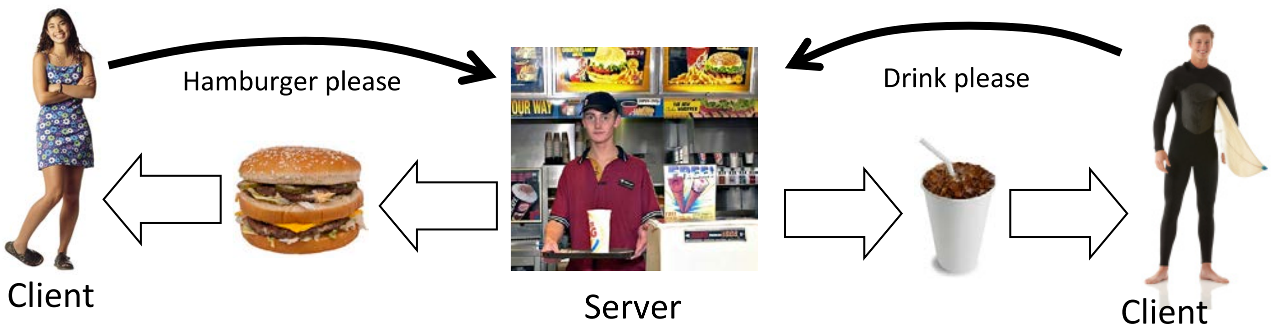
Example: `send(new_fd, "Hello, world!", 13, 0)` will send the string "Hello, world!" using the file descriptor "new_fd". The maximum length of the string is "13", and the flags are set to 0, i.e., turned off. The system call will return -1 if there is an error; otherwise it returns the number of bytes sent.

Example: `recv(sockfd, buf, MAXDATASIZE, 0))` will receive a string and store it in the buffer "buf", using the file decriptor "new_fd". The maximum length of the string is "MAXDATASIZE", and the flags are set to 0, i.e., turned off. The system call will return -1 if there is an error; otherwise it returns the number of bytes received.

In many cases, a user does not have the IP address but knows the text-oriented Internet, domain name, e.g., www.wiliki.eng.hawaii.edu. To check the IP address of wiliki, type "traceroute wiliki", which should give 127.0.0.1. A program can find the IP address from the domain name by using the Domain Name System (DNS) service. The DNS just translates domain names to IP addresses.

B Client-Server

In this project, you will implement a client and server. A *server* is an entity that waits for requests (or commands). When a request arrives, it processes the request and sends back a reply. The entity that sends requests is a *client*.



Attached are two programs: a client program client.c, and a server program server.c. This is from Beej's guide to network programming: http://beej.us/guide/bgnet/ More specifically it comes from the following which explains client, server, and sockets:
http://beej.us/guide/bgnet/output/html/multipage/clientserver.html

The server will have an IP address that is for the machine that it's running on, which for this lab is wiliki. It also needs a TCP port number. For this semester, you are assigned two TCP ports for your use. The assignments are in Section C. Pick one and use it for your server. In server.c and client.c, you will find the definition

```
#define PORT "3490"
```

Change 3490 to your port number.

The server will wait for requests to come through its TCP port, and the client will contact the server through the port.

You can compile both programs. Suppose the executables are named "server" and "client".

Run the server

```
./server
```

At this point, the console is attached to the server and you can't use it for anything else. This means the server is running in the "foreground". To free up the console, put the server in background as follows:

- Suspend the server by typing control-z.
- You now have control over the console. Type "bg" This resumes the server to run in background

Now type

```
./client wiliki.eng.hawaii.edu
```

Then the client will request a socket connection to the server. The server will accept the request and reply by sending "Hello World" through the socket connection. The client will receive the message and display it on the console.

The server is still running in background. You can kill it by first finding its process ID (PID):

```
   Type ps
or Type ps –a
```

This will list all the processes and their PIDs. If the PID for your process is 1234, you can terminate it by typing

```
kill 1234.
```

Another way to terminate the server is to bring it to the foreground by typing

```
fg
```

Then kill the process by typing control-c

If there is more than one process in the background then type

```
jobs
```

This will list out the job and a number. Suppose your job is 3. Then type

```
fg  %3
```

to bring the job into the foreground. Note that if you want to launch your server in background you can type

```
./server &
```

Warning: "Zombie" processes are those that are not terminated, e.g., they were not exited or killed. They continue running even after you logout. Zombie processes can accumulate. Since processes use up CPU cycles and memory space, zombie processes can completely use up your resources on wiliki where you cannot even login. You should periodically check if you have zombie processes by using "ps". Then kill them.

B.1 Client

The following is a shortened and simplified version of client.c. The error checking has been removed so that the code fits on one page, though this is not good programming style. Most of the function calls are from libraries which you can look them up by googling.

```
#define PORT "3490" // the port client will be connecting to
#define MAXDATASIZE 100 // max number of bytes we can get at once

int main(int argc, char *argv[])
{
      int sockfd, numbytes;
      char buf[MAXDATASIZE];
      struct addrinfo hints, *servinfo, *p;
      int rv;
      char s[INET6_ADDRSTRLEN];

      // Initialize variables to set up a socket
      memset(&hints, 0, sizeof hints);
      hints.ai_family = AF_UNSPEC;
      hints.ai_socktype = SOCK_STREAM;
      //    Get the IP address from the domain name in argv[1] using the DNS
      //          "getaddrinfo" is given below
      rv = getaddrinfo(argv[1], PORT, &hints, &servinfo));
      p = servinfo;

      // Get a file descriptor for the socket
      sockfd = socket(p->ai_family, p->ai_socktype, p->ai_protocol);

      // Send a request to connect to the server
      connect(sockfd, p->ai_addr, p->ai_addrlen);

      // Convert the IP address in the data structure
      //    into a character string, which is stored in char array "s"
      inet_ntop(p->ai_family, get_in_addr((struct sockaddr *)p->ai_addr),s, sizeof s);
      printf("client: connecting to %s\n", s);

      freeaddrinfo(servinfo); // all done with this structure

      // Receive a char string from the server and then display
      numbytes = recv(sockfd, buf, MAXDATASIZE-1, 0));n
      buf[numbytes] = '\0';
      printf("client: received '%s'\n",buf);

      close(sockfd); // close socket
      return 0;
}

// get sockaddr, IPv4 or IPv6:
void *get_in_addr(struct sockaddr *sa)
{     if (sa->sa_family == AF_INET) return &(((struct sockaddr_in*)sa)->sin_addr);
      else                          return &(((struct sockaddr_in6*)sa)->sin6_addr);
}
```

B.2 Server

The following is a shortened and simplified version of server.c.

```
#define PORT "3490"  // the port users will be connecting to
#define BACKLOG 10   // how many pending connections queue will hold

int main(void)
{     int sockfd, new_fd;  // listen on sock_fd, new connection on new_fd
      struct addrinfo hints, *servinfo, *p;
      struct sockaddr_storage their_addr; // connector's address information
      socklen_t sin_size;
      struct sigaction sa;
      int yes=1;
      char s[INET6_ADDRSTRLEN];
      int rv;

      memset(&hints, 0, sizeof hints); // Similar to client
      hints.ai_family = AF_UNSPEC;
      hints.ai_socktype = SOCK_STREAM;
      hints.ai_flags = AI_PASSIVE; // use my IP since I am the server

      rv = getaddrinfo(NULL, PORT, &hints, &servinfo);
      p = servinfo;
      sockfd = socket(p->ai_family, p->ai_socktype, p->ai_protocol);
      setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int));
      bind(sockfd, p->ai_addr, p->ai_addrlen);
      freeaddrinfo(servinfo); // all done with this structure

      listen(sockfd, BACKLOG); // Mark this socket as “passive” to just listen

      sa.sa_handler = sigchld_handler; // reap all dead processes -- housekeeping
      sigemptyset(&sa.sa_mask);
      sa.sa_flags = SA_RESTART;
      sigaction(SIGCHLD, &sa, NULL) == -1); // Note callback function is sa.sa_handler

      printf("server: waiting for connections...\n");

      while(1) {  // main accept() loop – constantly processing requests from clients
            sin_size = sizeof their_addr;
            //   Accept a new connection request from the client through the socket
            //      Create a new connection to communicate with the client
            //      new_fd is the file descriptor of the new socket
            new_fd = accept(sockfd, (struct sockaddr *)&their_addr, &sin_size);
            inet_ntop(their_addr.ss_family, get_in_addr((struct sockaddr *)&their_addr),
                  s, sizeof s);
            printf("server: got connection from %s\n", s);

            if (!fork()) { // this is the child process
                           //  Note: this is the same as “if (fork() == 0)”
                  close(sockfd); // child doesn't need the listener
                  send(new_fd, "Hello, world!", 13, 0);
                  close(new_fd);
                  exit(0);
            }
            close(new_fd);  // parent doesn't need this
      }
}  // ---- There’s more code on the next page.
```

```
void sigchld_handler(int s)
{
      while(waitpid(-1, NULL, WNOHANG) > 0);
}

// get sockaddr, IPv4 or IPv6: -- same as in client.c
void *get_in_addr(struct sockaddr *sa)
{
      if (sa->sa_family == AF_INET) return &(((struct sockaddr_in*)sa)->sin_addr);
      else return &(((struct sockaddr_in6*)sa)->sin6_addr);
}
```

In this example, server has a primary port number 3490. It creates a socket for this port with a file descriptor "sockfd". Functions "bind" and "socketopt" set appropriate parameters, and "listen" will establish the socket as passive. This socket is dedicated to only receiving connection requests from clients. It is not used to transfer data between the client and server.

Next, the server goes to an infinite-loop. The server waits for connection requests from clients via "accept". When a request arrives from a client, "accept" will create another socket specifically for the client which has the file descriptor "new_fd". The server will use "new_fd" to communicate with the client.

At this point, the server creates a child process to deal with the client. In this way, the parent is free to resume dealing with new connection requests from clients.

C Assignment

For this assignment, you will work in groups of two, with the exception that there may be one group of three. As a group, you will develop two programs, server367 and client367, which are described later in this section. Your programs should run on wiliki. The most straightforward way to do this is to modify server.c and client.c. Hints are given in Section D.

You are to work with your partner by doing "pair programming":
https://en.wikipedia.org/wiki/Pair_programming
https://www.youtube.com/watch?v=YhV4TaZaB84

Remember to switch roles frequently, e.g., every 30 minutes with a short break of a few minutes.

Here is show the client and server will work. Suppose server367 is running. A user should be able to use client367 to do the following. The client will continually accept commands until the user quits. The user interface is

Command (type 'h' for help):

The commands are single text characters. Here are the commands:

- **l: List**: List the contents of the directory of the server.
- **c: Check <file name>:** Check if the server has the file named <file name>.
  - If it exists then the client will output "File '<file name>' exists" on the console.
  - Otherwise, the client will output "File '<file name>' not found".
- **p: Display <file name>:** Check if the server has the file named <file name>.
  - If it exists then the client will display the contents of the file on the client's console.
  - Otherwise, the client will output "File '<file name>' not found".
- **d: Download <file name>:**
  - Check if the client has a file in its directory with the same file name
    - If it does then it will query the user if it would like to overwrite it
    - If the user does not want to overwrite then the client will discontinue processing this command
  - Check if the server has the file.
    - If it does then the client will download the file using the same file name.
    - Otherwise, the client will output "File '<file name>' not found"
- **q: Quit:** This is to terminate the client program. Otherwise the client continues.

- **h: Help:** Lists all the commands, e.g.,
  - **l: List**
  - **c: Check <file name>**
  - **p: Display <file name>**
  - **d: Download <file name>**
  - **q: Quit**
  - **h: Help**

The user starts client367 by typing

```
./client367    <host name>
```

For example,

```
./client367   wiliki.eng.hawaii.edu
```

You will need a port number for your server. The usable ports, called the ephemeral ports, are from 2000 to 5000. Each student will be assigned one port number according to the table below:

Name	Port #	Name	Port #	Name	Port#
Badke, David R.	3500 3600	Kahakui, Richard K.	3514 3614	Shimabukuro, Jared M.	3527 3627
Barbour, Elena M.	3501 3601	Kurano, Andrew L.	3515 3615	Tokita, Dylan K.	3528 3628
Cabael, Kevin P.	3502 3602	Lam, Joshua C.	3516 3616	Wu, Mengyuan	3529 3629
Cho, Kevin J.	3503 3603	Lam, Nathan C.	3517 3617	Yuu, Ian I.	3530 3630
Chong, Wendy W.	3504 3604	Lemus, Paulo P.	3518 3618		3531 3631
Desmond, Leilani M.	3505 3605	Levesque, Kyle A.	3519 3619		3532 3632
Godinet, Kayla A.	3506 3606	Mehta, Suraj R.	3520 3620		3533 3633
Grazziotin, Jessica B.	3507 3607	Moriyasu, Thomas A.	3521 3621		3534 3634
Hagi, Kasey K.	3508 3608	Mukai, Reyn H.	3521 3621		3535 3635
Hester, Joshua J.	3509 3609	Mynatt, Micah K.	3522 3622		3536 3636
Higa, Ross T.	3510 3610	Nakahodo, Dylan K.	3523 3623		3537 3637
Ibara, Casey I.	3511 3611	Nakamatsu, Keith T.	3524 3624		
Ines, BeeJay I.	3512 3612	Nakanishi, Kyle A.	3525 3625		
Izumigawa, Christianne G.	3513 3613	Robles, Keanu D.	3526 3626		

Included with this project are the following files:

- server.c, client.c: The simple stream server and client from Beej's web site
- ls367.c: This is an example of using execl
- pipe.c: The example code for running a program from a C program and using execl and pipes
- arguments.c: This is an example used in Appendix D
- file1.txt, file2.txt, file3.txt: Sample data files that can be downloaded or displayed.

Submission Instructions:

Even though you work in groups, each student must submit the following. In a wiliki directory named "Lab3" include

- Your source code for your client and server, i.e., client367.c and server367.c
- Your makefile
- The data files file1.txt, file2.txt, and file3.txt
- README file that includes
    - Your name
    - EE 367L Lab 3
    - Instructions to compile. If you have a make file, these instructions could simple be "enter makefile"
    - Instructions to run the programs.

Tar the directory: e.g., tar cvf Lab3.tar Lab3

Gzip the tar'd directory. Upload the gzipped directory into laulima under the Assignment for the project.

D Hints

Here are hints to complete the project.

D.1 Software Development

This is what the client and server should do

- Client
  - The client is basically a loop that does the following
  - The client gets a command from the user through its user interface
    - "Command (enter 'h' for help): "
  - It responds to the command. For commands that require connecting to the server, it does the following
    - It encodes the command. For example, it could encode a command into single text character: "C" = check, "D" = download, "L" = list, and "P" = display.
    - It sets up a socket to the server, and then sends the command to the server. It may include additional parameters:
      - check = "C <file name>"
      - download = D <file name>"
      - list = "L"
      - display = "P <file name>"
    - It waits for a reply from the server by using "recv".
    - After completing the command, it terminates the socket
- Server
  - The server waits to accept connections
  - After a connection is established, the server creates a child to handle the client
  - The child receives a message text string from the client
  - It parses the message to determine the command
    - If the command is "C", "D", or "P" it further parses the message to find the file name
    - If the command is "C" then it checks if the file exists and sends a response back to the client
    - If the command is "D" or "P" then the file is opened and the contents are sent to the client
    - If the command is "L" then the child will create its own child, which executes "ls" using execl( ). The output is sent to the client.
  - After the command is completed, the child closes the socket and terminates.

Build the client and server in stages, each stage is an improvement over the previous stage. Here is a suggestion for a sequence of stages.

Stage 0:
Run client.c and server.c using your own port numbers.

Stage 1:
Have the server execute "ls" whenever a client tries to connect to it. The "ls" should output to the console, i.e., the default output.

Stage 2:
This is the same as Stage 1 but have the output of "ls" is sent back to the client, where it is displayed.

Stage 3:
The client should have a user interface that accepts the commands to "list" or "quit".

Stage 4:
The client and server should include the command "check".

Stage 5:
The client and server should include the command "display". To implement "display", first have the server display the file directly to the console. Then have the server send the file back to the client.

Stage 6:
The client and server should include the command "download". Note that "download" is different from "display" because the client will store the file rather than display it on the console.

Stage 7:
Complete the assignment.

You can break these stages down even further. It is important to split up your progress so that it can be debugged. As much as possible, deal with one bug at a time.

D.2 System Calls

This lab will use a number of system and library calls. Two of the most common are "write" and "read", which transfer strings of bytes. The system calls "send" and "recv" are similar.

You can look up system calls by googling them. For example, googling "Linux write()" will return

http://man7.org/linux/man-pages/man2/write.2.html

that has a description of the operation, input parameters, return value, and any libraries that must be included, e.g., unistd.h.

The following is for read():

http://man7.org/linux/man-pages/man2/read.2.html

Note that both write() and read() have parameters with data types size_t and ssize_t.   You can assume it's the data type int.

D.3 Arguments in main()

Function prototypes of the main function have the following forms (main can also return void)

```
int main(void);
int main();   // This is the same as the line above

int main(int argc, char **argv);
int main(int argc, char *argv[]);  // This is the same as the line above
```

The following is an example:

```
void main(int argc, char *argv[])
{
int k;

printf("Number of arguments = %d\n");
for (k=0; k,argc; k++) {
    printf("argv[%d] -> %s\n", k, argv[k]);
}
}
```

argv[ ] is known as the argument vector. Suppose you compiled the program and then launched it as follows:

```
./a.out   -a  this   is  an  example
```

What will appear on the console is

```
Number of arguments = 6
argv[0] -> ./a.out
argv[1] -> -a
argv[2] -> this
argv[3] -> is
argv[4] -> an
argv[5] -> example
```

A copy this program is attached and labeled "argument.c"