**EE 367L Lab Client-Server**

**Due date:  See laulima**

This is part 2 of the lab’s documentation. Here is a quick review:

The objective of this assignment is to gain experience with

* Client server paradigm
* Processes
* System calls (calls to functions in the operating system, which is assumed to be Linux)
* How data is transported over the Internet
* Communication between processes
  + pipes: communication between processes in a single machine
  + sockets: communication between processes over the Internet

The assignment is to write a simple client server system.

There are three parts to this lab:

* Part 1, Processes
  + What is a process
  + Managing processes
  + Launching programs from a process
  + Communicating processes: pipes
  + I/O redirection: dup2
  + More about processes
* Part 2, Internet – Reading, a few exercises in this document, and the Assignment
  + Client server
  + Internet
  + Sockets
  + Simple client-server example
  + Zombie processes
  + TCP port assignments
  + Simple exercises
  + Assignment

**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*.



Hamburger please

Drink please



Server

Client

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>

We will first describe the Internet, and then the code for the client.c and server.c

**Internet**

Clients send and receive data to and from the server. Previously, we used pipes to connect processes, but pipes only work for programs in a single machine. An Internet server must be available to clients in other computers by communicating through the Internet. An important mechanism to connect processes over the Internet is a *socket*. We will discuss the basics of the Internet, and then describe sockets.

The Internet is basically a big switch, as shown below, that is able to transport files to end nodes, which are referred as *hosts*. Hosts connected to the Internet must have Internet addresses (or IP addresses), which are 32-bit numbers (or four bytes). As an example, wiliki has the IP address 127.0.0.1. This address is in the *decimal dot format*, where the four bytes are represented by decimal values separated by dots. The IP address is the Internet’s ID for the host.



**127.0.0.1**

**wiliki.eng.hawaii.edu**

**Internet**

**www.manoa.hawaii.edu**

**128.171.224.104**

**192.168.101.30**



**Host**

**Host**

**Host**



**Happy User**

**Port 22**



**Port 25**

Usually there are multiple software applications running on a host that are accessing the Internet. For example, we could have the following applications running simultaneously: Chrome Internet browser, Netflix video player, Mpg audio player, Skype, Putty, Filezilla, and so on. The applications must share the IP address of the host. In order to multiplex the software applications through the IP address, there are *port numbers*. These port numbers are part of network transport protocols called TCP and UDP. Application software use ports to connect with other application software in the Internet. The port numbers are 16 bits, so they range from 0 to around 64K. Some of the port numbers are reserved, such as port 80 is used for the Internet server for the host, port 22 (basically SSH) is used in Filezilla, port 25 is used in Simple Mail Transfer protocol for email.

Notice that each Internet domain name corresponds to an IP address, e.g., wiliki.eng.hawaii.edu has IP address 127.0.0.1. The IP address is used by the machines, but it’s too difficult to remember, so we have domain names. There is a directory service that translates the domain names to IP addresses and vice versa. This is called the Domain Name System (DNS). There are Linux system calls that use the DNS to translate a domain name to an IP address.

As mentioned earlier there are two network transport protocols: TCP and UDP. TCP is used to transport a stream of bytes, while UDP is used to transport a single file of limited size, at most around 64KB. TCP is known as stream service, and UDP is known as datagram service. TCP is the most popular since it can transport an unlimited amount of bytes. For the rest of discussion, we will assume TCP rather than UDP, though keep in mind UDP is available. Also, there are two Internet protocols (IP), which is IPv4 (version 4) and IPv6 (version 6). We’ll work with IPv4, which is the traditional version.

**Sockets**

Network sockets are communication end points that can be used by processes to set up network connections between them. The connections can be on different computers. The connections can be realized by a TCP or UDP connection. But the details of running a connection are taken care of by the operating system, so the socket is a nice interface for application software.

A socket is identified by its address, which is composed of an IP address and a TCP port number. It is often written as IPAddress:TCPportNumber, e.g., 127.0.0.2:22. When a process sets up a socket, it’s ready to make a TCP connection using its socket address. The IP address is already determined because it corresponds to the machine that the process is running on. For example, if the process is running on wiliki, then the IP address is for wiliki. The TCP port number may be fixed for particular applications. For example, if the process is an Internet server, the port number may have to be 80. However, for many applications, there is no particular port number, so a process can choose one that is available.

The figure below shows four processes, each with a socket (shown are the sockets’ addresses). The processes reference sockets through socket file descriptors (sd), which are analogous to ordinary file descriptors for files. Two of the sockets have a connection between them, allowing transfer of data. Two of the sockets have the same IP address indicating they are in the same computer.

Process

**127.0.0.7:22**

**96.45.0.25:1024**

Sockets

IP address:Port #

**128.33.12.255:24**

**128.33.12.255:10**

Process

Process

Process

**sd**

**sd**

**sd**

**sd**

**connection**

Network sockets and connections are analogous to physical connectors. Note that in physical connections there are “connectors” and cables (or wires). Example male and female connectors are shown below



Male

Male

Female

Female

Sockets are analogous to female connectors. A network connection is analogous to having a cable, with male connectors at its ends, attached to two female connectors. A network connection can connect two sockets, just as a cable can connect two female connectors.

A *socket pair* is a communication connection between two sockets. It is described by a unique pair of (IPAddress:TCPPortNumber). For example, (127.0.0.1:3410, 96.255.56.12:22) is a socket pair. A socket communication connection is bidirectional, so its end points can send and receive data bytes.

A server will use a socket by putting it into the *listening* state. Then the server is waiting for requests from clients to establish a connection. These requests are queued at the server’s socket. A request can be *accepted* by the server, which creates a connection. The connection corresponds to the socket pair (ClientIPAddress:ClientPortNumber), (ServerIPAddress:ServerPortNumber). The following figure shows a server process connected to three client processes. It also shows the socket addresses of the socket pairs.

**127.0.0.7:22**

**96.45.0.25:1024**

**128.33.12.255:24**

**128.33.12.255:10**

Server

Process

**sd**

**sd**

**sd**

**sd**

**connection**

Client

Process

Client

Process

Client

Process

**96.45.0.25:1024, 128.33.12.255:24**

**127.0.0.7:22, 128.33.12.255:24**

**128.33.12.255:10, 128.33.12.255:24**

The following system calls are used to establish and manage network connections and sockets

**getaddrinfo:**

int **getaddrinfo**( char \*node, /\* domain name or IP address \*/

char \*service, /\* Port number or well known name of port number, e.g., “http” \*/

struct addrinfo \* hints, /\* Parameters of the connection set by caller \*/

struct addrinfo \*\* res /\* Information used by the operating system software \*/

):

getaddrinfo converts connection information readable by humans into connection information understandable by the operating system software. Here is the struct addrinfo:

struct addrinfo {

int ai\_flags;

int ai\_family; /\* Type of network; in our case it’s IPv4 (AF\_INET) \*/

int ai\_socktype; /\* Type of socket; in our case it’s SOCK\_STREAM (= 1) \*/

int ai\_protocol; /\* Protocol type; in our case it’s TCP (=6) \*/

socklen\_t ai\_addrlen; /\* Length of IP address; in the case of IPv4, it’s 4 bytes \*/

struct sockaddr \*ai\_addr; /\* IP address and other address information \*/

char \*ai\_canonname; /\* Cannonical name of host, which may be different from \*/

/\* host domain name \*/

struct addrinfo \*ai\_next; /\* Next node in the linked list \*/

};

“hints” is used by the calling function to specify the parameters. “res” is the output of getaddrinfo. It is actually a pointer to a linked list, where each node is a structure. The reason why “res’ is a linked list is that a host can have multiple address configurations, e.g., IPv4 or IPv6 address and configurations. Each node in the linked list corresponds to an address configurations. To free the linked list, call freeaddrinfo( ).

Examples: getaddrinfo(“127.0.0.1”, 3490, &hints, &res); /\* res = addrinfo for 127.0.0.1:3490\*/

getaddrinfo(“wiliki.eng.hawaii.edu”, “http”, &hints, &res);

In the first example, the system will look up the socket addresses of 127.0.0.1:3490. In the second example, the socket address corresponds to “wiliki.eng.hawaii.edu”:”http”. “wiliki.eng.hawaii.edu” is converted to its IP address using the DNS service, and “http” is converted to port number 80.

In ‘struct addrinfo’, there is a member ‘struct sockaddr’, which is the structure for the socket address. This can be for an IPv4 or IPv6 address, which are structured differently. The definition of struct addr is as follows:

struct sockaddr {

unsigned short sa\_family; // address family, AF\_xxx

char sa\_data[14]; // 14 bytes of protocol address

};

The following socket structures are specific to IPv4 and IPv6 protocols.

struct sockaddr\_in { /\* Socket address structure for IPv4 AF\_INET sockets \*/

short sin\_family; // e.g., AF\_INET, AF\_INET6

unsigned short sin\_port; // e.g., TCP port number – htons(3490), 2 bytes

// Here htons() converts the bytes of the value into the proper

// order for use in the network; think big and little endian

struct in\_addr sin\_addr; // This is unsigned long, so it’s 4 bytes

char sin\_zero[8]; // Padding, and you can zero this

};

struct sockaddr\_in6 { /\* Socket address structure for IPv6 AF\_INET6 sockets \*/

u\_int16\_t sin6\_family; // address family, AF\_INET6

u\_int16\_t sin6\_port; // port number, Network Byte Order

u\_int32\_t sin6\_flowinfo; // IPv6 flow information

struct in6\_addr sin6\_addr; // IPv6 address, 16 bytes

u\_int32\_t sin6\_scope\_id; // Scope ID

};

Note that sockaddr, sockaddr\_in and sockaddr\_in6 have the same beginning structure. So you can freely cast a pointer of one structure to another without problems. Thus, a pointer variable for struct socketaddr can be used to point to struct socketaddr\_in or struct socketaddr\_in6. Then getaddrinfo can be used for IPv4 or IPv6. See the following for more information: <https://beej.us/guide/bgnet/html/multi/sockaddr_inman.html>

Now suppose you want to create a sockaddr variable that can be used to carry either sockaddr\_in or sockaddr\_in6 data. Note that a struct sockaddr variable won’t work since it’s too small for data in struct ssockaddr\_in6. Instead you can use the following which has enough space for sockaddr\_in or sockaddr\_in6:

struct sockaddr\_storage{

sa\_family\_t ss\_family; // address family

// all this is padding, implementation specific, ignore it:

char \_\_ss\_pad1[\_SS\_PAD1SIZE];

int64\_t \_\_ss\_align;

char \_\_ss\_pad2[\_SS\_PAD2SIZE];

};

So use struct sockaddr if you want a generic form for a pointer to either struct sockaddr\_in or struct sockaddr\_in6, but if you want to allocate memory space that will work for either sockaddr\_in or sockaddr\_in6 use sockaddr\_storage.

References:

Full description of getaddrinfo(): <https://linux.die.net/man/3/getaddrinfo>

Description from wikipedia: <https://en.wikipedia.org/wiki/Getaddrinfo>

Description of struct addrinfo: <https://msdn.microsoft.com/en-us/library/windows/desktop/ms737530(v=vs.85).aspx>

**socket:**

sockfd = **socket**( int socket\_family, /\* Type of network, e.g., IPv4 or IPv6 \*/

int socket\_type, /\* Socket type, e.g., TCP stream socket (SOCK\_STREAM) \*/

int protocol /\* Network protocol type, e.g., TCP (=6) \*/

);

Creates a socket according to the input parameters and returns a socket file descriptor.

References:

Full description of socket(): <https://linux.die.net/man/7/socket>

**connect:**

int **connect**( int sockfd,

struct sockaddr \* addr,

socklen addrlen /\* Address length \*/

);

Creates a connection between the socket referenced by ‘sockfd’ with the (possibly remote) socket specified by ‘addr’. The connection for the resulting socket pair can be accessed by referencing sockfd. Note that the connection requires a local socket address, which is the IP address of the local machine and, by default, an unused TCP port chosen by the operating system. A particular TCP port can be specified by using the “bind()” system call.

In a client-server system, connect() is used by a client to connect to a server.

References:

Full description of connect(): <https://linux.die.net/man/2/connect>

**bind:**

int **bind**(int sockfd, struct sockaddr \*addr, socklen\_t addrlen);

bind() assigns an address specified in ‘addr’ to the socket corresponding to ‘sockfd’. This is used by a server to specify a socket address to its local socket.

Reference:

Full description of bind(): <https://linux.die.net/man/2/bind>

**listen:**

int **listen**(int sockfd, int backlog);

listen() marks the socket specified by ‘sockfd’ as a passive socket, waiting for incoming connection requests. The requests arrive in a queue, which has a maximum size equal to ‘backlog’. Incoming requests that find a full queue are dropped. listen() is used by servers, who are waiting for requests from clients.

Note that a connection request is just request, and a connection has not been established yet.

Reference:

Full description of listen(): <https://linux.die.net/man/2/listen>

**accept**:

int **accept**(int sockfd, struct sockaddr \*addr, socklen\_t addrlen);

Socket ‘sockfd’ must be a listening socket. accept() will accept a connection request in the queue of requests for ‘sockfd’. Then a connection is established and the socket file descriptor is returned.

The returned socket file descriptor will be different from ‘sockfd’. In this way, the new connection will not interfere with the listening socket ‘sockfd’. If the queue of requests for ‘sockfd’ is empty then accept() is blocking until a request arrives. Also the address of connection requester is put in ‘addr’.

Full description of accept(): <https://linux.die.net/man/2/accept>

**send:**

ssize\_t **send**(int sockfd, const void \*buf, size\_t len, int flags);

Sends data in the buffer ‘buf’ into the socket ‘sockfd’ for a maximum of ‘len’ bytes. The return value is the actual number of bytes sent, which will be at most ‘len’. If flags = 0 then send() is equivalent to write().

Full description of send(): <https://linux.die.net/man/2/send>

**recv:**

ssize\_t **recv**(int sockfd, const void \*buf, size\_t len, int flags);

Receives data in the buffer ‘buf’ from the socket ‘sockfd’ for a maximum of ‘len’ bytes. The returned value is the actual number of bytes received, which will be at most ‘len’. If flags = 0 then recv() is equivalent to read(). By default, recv() is blocking until there is data to receive. You can change the connection to be nonblocking using fcntl() – in particular: fcntl(sockfd, F\_SETFL, O\_NONBLOCK); which sets flags of sockfd to nonblocking.

Full description of recv(): <https://linux.die.net/man/2/recv>

**setsockopt:**

int setsockopt(int sockfd, int level, int optname, void \*optval, socklen\_t \*optionlen);

This will configure a socket sockfd. Configuration can be done at different levels, but for our purposes, the level will be the socket level, specified by SOL\_SOCKET.  *optname* is the option to set. In our case, we want it equal to SO\_REUSEADDR which allows the socket’s port number to be reused. *optval* are the parameters needed to set the option.  *optionlen* is a length of optval.

This option solves a problem in the event that a server terminates and is restarted right away, e.g., server crashes and then restarts quickly. Then sockets using the server port may not have terminated yet, so when the server restarts and tries to bind() to the port, it gets an “Address already in use” error message. By allowing the port to be reused, the restarting server can reuse the busy port.

Description of setsockopt() from Beej’s guide: <https://beej.us/guide/bgnet/html/multi/setsockoptman.html>

Here’s an explanation of SO\_REUSEADDR: <http://www.unixguide.net/network/socketfaq/4.5.shtml>

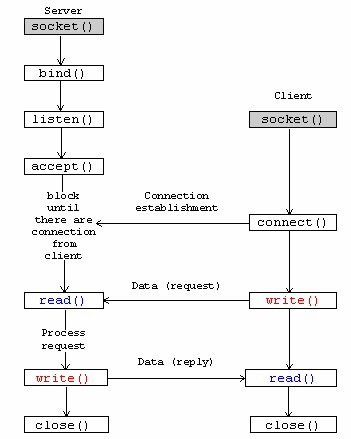
**Simple Client-Server Example**

We will go over a simple client-server example given in <https://beej.us/guide/bgnet/html/multi/index.html>

The web site explains socket programming. The important sections for this lab are Section 5 on System Calls, and Section 6 on Client-Server Background:

* Section 5: <https://beej.us/guide/bgnet/html/multi/syscalls.html>
* Section 6: <https://beej.us/guide/bgnet/html/multi/clientserver.html>

In Section 6, there is a client program client.c and server program server.c. Both of them are attached to this document. We will go over them. First note that in the simplest form, a client-server follows the figure below.



**client.c:**

/\*

\*\* client.c -- a stream socket client demo

\*/

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <errno.h>

#include <string.h>

#include <netdb.h>

#include <sys/types.h>

#include <netinet/in.h>

#include <sys/socket.h>

#include <arpa/inet.h>

#define PORT "3490" // the port client will be connecting to – port of the server

#define MAXDATASIZE 100 // max number of bytes we can get at once

// get sockaddr, IPv4 or IPv6 depending on the protocol

void \*get\_in\_addr(struct sockaddr \*sa)

{

if (sa->sa\_family == AF\_INET) { // Return IPv4 address

return &(((struct sockaddr\_in\*)sa)->sin\_addr);

}

return &(((struct sockaddr\_in6\*)sa)->sin6\_addr); // Return IPv6 address

}

int main(int argc, char \*argv[])

{

if (argc != 2) {

fprintf(stderr,"usage: client hostname\n");

exit(1);

}

// Get address information of server with IP address (or domain name) ‘arg[1]’

// and TCP port number PORT = “3490”. It’s a stream connection which can be

// IPv4 or IPv6

struct addrinfo hints; // hints used in getaddrinfo()

memset(&hints, 0, sizeof hints); // Initialize hints to 0 to avoid problems

// from spurious leftover data in memory

hints.ai\_family = AF\_UNSPEC; // Can be either IPv4 or IPv6

hints.ai\_socktype = SOCK\_STREAM; // TCP stream socket

struct addrinfo \*servinfo; // It will point to the results from getaddrinfo()

int rv;

if ((rv = getaddrinfo(argv[1], PORT, &hints, &servinfo)) != 0) {

fprintf(stderr, "getaddrinfo: %s\n", gai\_strerror(rv));

return 1;

}

// Connect to the server:

// loop through all the results and connect to the first we can

int sockfd;

struct addrinfo \*p;

for(p = servinfo; p != NULL; p = p->ai\_next) {

// Create a socket, if possible

if ((sockfd = socket(p->ai\_family, p->ai\_socktype,

p->ai\_protocol)) == -1) {

perror("client: socket");

continue; // If it doesn’t work then skip the rest of this pass

// and continue with loop

}

// We created a socket, now let’s establish a connection to the server

// with the socket address for ‘p’.

if (connect(sockfd, p->ai\_addr, p->ai\_addrlen) == -1) {

close(sockfd);

perror("client: connect");

continue; // If we can’t establish a connection then skip

// the rest of this pass and continue with loop

}

// We created a socket and established a connection so we’re done

// with the loop

break;

}

if (p == NULL) { // We failed to connect after going through

// the linked list servinfo

fprintf(stderr, "client: failed to connect\n");

return 2;

}

// We established a connection with the server

// Notify the user that we made the connection

char s[INET6\_ADDRSTRLEN]; // Buffer to store char string

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

// Display what’s received from server

int numbytes;

char buf[MAXDATASIZE];

if ((numbytes = recv(sockfd, buf, MAXDATASIZE-1, 0)) == -1) {

// Note that recv() can be replaced by read() since flags = 0

perror("recv");

exit(1);

}

buf[numbytes] = '\0';

printf("client: received '%s'\n",buf);

close(sockfd);

return 0;

}

**server.c:**

/\*

\*\* server.c -- a stream socket server demo

\*/

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <errno.h>

#include <string.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <netdb.h>

#include <arpa/inet.h>

#include <sys/wait.h>

#include <signal.h>

#define PORT "3490" // Port used by server. Clients need to know this port number

#define BACKLOG 10 // how many pending connections queue will hold

// sigchld\_handler is used to clean up zombie processes

// This will be explained later

void sigchld\_handler(int s)

{

while(waitpid(-1, NULL, WNOHANG) > 0);

}

// the following is the same as in client.c, so we just have the prototype.

void \*get\_in\_addr(struct sockaddr \*sa);

int main(void)

{

// Get address information for this server which includes the IP address of

// the local machine, and the TCP port number of the server, in this

// case PORT = “3490”. It’s a stream connection which can be

// IPv4 or IPv6. This is similar to client.c but we want the

// IP address of the local machine

struct addrinfo hints;

memset(&hints, 0, sizeof hints);

hints.ai\_family = AF\_UNSPEC;

hints.ai\_socktype = SOCK\_STREAM;

hints.ai\_flags = AI\_PASSIVE; // use the IP of the local machine

// AI\_PASSIVE indicates that the resulting

// socket will be used for listening by a server

struct addrinfo \*servinfo; // The address results which is a linked list

int rv;

if ((rv = getaddrinfo(NULL, PORT, &hints, &servinfo)) != 0) {

fprintf(stderr, "getaddrinfo: %s\n", gai\_strerror(rv));

return 1;

}

// Loop through all the linked list of addressinfo results

// and create a socket and bind it to the addressinfo.

// Stop when the bind is successful.

int sockfd;

struct addrinfo \*p;

for(p = servinfo; p != NULL; p = p->ai\_next) {

// Create a socket, if possible

if ((sockfd = socket(p->ai\_family, p->ai\_socktype,

p->ai\_protocol)) == -1) {

perror("server: socket");

continue; // If it doesn’t work then skip the rest of this pass

// and continue with loop

}

// setsockopt sets the socket so that it can rebind to its port after crash

int yes=1;

if (setsockopt(sockfd, SOL\_SOCKET, SO\_REUSEADDR, &yes,

sizeof(int)) == -1) {

perror("setsockopt");

exit(1);

}

// Let’s bind the socket with the socket address for ‘p’.

if (bind(sockfd, p->ai\_addr, p->ai\_addrlen) == -1) {

close(sockfd);

perror("server: bind");

continue; // If we can’t bind then skip the rest of this pass

// and continue with the loop

}

// We created a socket and it’s bound to a socket address with

// the IP address of the local machine and TCP port number PORT

// We’re done so we can break out of the loop

break;

}

if (p == NULL) { // We failed to create a socket for the server

fprintf(stderr, "server: failed to bind\n");

return 2;

}

// We now have a socket for the server for listening

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

// Server starts to listen for connection requests from clients.

if (listen(sockfd, BACKLOG) == -1) {

perror("listen");

exit(1);

}

// In what follows, the server will create child processes. It turns out that

// the child processes can become zombie processes. What immediately follows

// is a method to get rid of zombie processes. This will be explained later.

struct sigaction sa;

sa.sa\_handler = sigchld\_handler; // reap all dead processes

sigemptyset(&sa.sa\_mask);

sa.sa\_flags = SA\_RESTART;

if (sigaction(SIGCHLD, &sa, NULL) == -1) {

perror("sigaction");

exit(1);

}

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

// The server runs forever in a while-loop

// In each pass of the loop, the server will accept() a connection

// request from a client. accept() will return the socket fd for

// the connection.

//

// After creating the connection, the server will create a child process

// to reply to the client. In this case, the reply is to simply

// send the message “hello world”.

while(1) { // main accept() loop

struct sockaddr\_storage their\_addr; // connector's addr information

socklen\_t sin\_size = sizeof their\_addr;

int new\_fd = accept(sockfd, (struct sockaddr \*)&their\_addr, &sin\_size);

if (new\_fd == -1) {

perror("accept");

continue;

}

// Display who the client is

char s[INET6\_ADDRSTRLEN];

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

close(sockfd); // child doesn't need the listener

if (send(new\_fd, "Hello, world!", 13, 0) == -1)

perror("send");

close(new\_fd);

exit(0);

}

close(new\_fd); // parent doesn't need this

}

return 0;

}

**Zombie Processes**

When a Linux process is created, it is recorded in a table in the operating system called the *process table*, which has all the active processes. The table can be used by a parent process to check the state of its child. For example, if the parent is executing a wait() for a child, it can check the table.

When a child process exits, it terminates, and all its memory and other resources are deallocated. However, it still has an entry in the process table because it’s needed when the parent calls a wait() for the child. Now the child is a *zombie process* -- there’s nothing left for the child except its entry in the process table.

Here is a description of zombie processes:

<https://en.wikipedia.org/wiki/Zombie_process>

The child’s entry in the process table can be removed when a wait( ) is called by the parent process for the child. So the trick to completely delete a zombie process is to force its parent to call a wait() for it. However, forcing a parent to call wait() may be awkward for the parent, since this may interfere with the parent completing its own task. The solution is to create a *signal handler* to do the waiting.

A *signal* is a way for a process to be notified of an event, and a *signal handler* is code that is run when the signal occurs. When a signal occurs, the process suspends whatever it is running, and goes to the signal handler. When the signal handling is completed, the process resumes running at the point where it was interrupted by the signal. A signal is analogous to an interrupt, and a signal handler is analogous to an interrupt handler.

An example of a signal is when you type Control-C while a process is running in foreground. This signal is known as SIGINT. Its default signal handler will terminate the process.

Now when a child process terminates, it sends a signal called SIGCHLD to its parent process. Suppose the signal handler will wait() for the child process. This will remove the child from the process table, and the child will end being a zombie process.

Let’s look at the signal handler for server.c:

void sigchld\_handler(int s)

{

while(waitpid(-1, NULL, WNOHANG) > 0);

}

This handler calls waitpid() with the WNOHANG option. This means waitpid() won’t wait if there are no terminated processes. Thus, the while-loop will call waitpid() as long as there are zombie processes. This will clean up all the zombie processes. When there are no zombie processes, the while-loop will not wait and the signal handler is done.

The following is how the signal handler sigchld\_handler is registered to SIGCHLD in main():

struct sigaction sa;

sa.sa\_handler = sigchld\_handler; // reap all dead processes

sigemptyset(&sa.sa\_mask);

sa.sa\_flags = SA\_RESTART;

if (sigaction(SIGCHLD, &sa, NULL) == -1) {

perror("sigaction");

exit(1);

}

The registration is done using the system call sigaction().

The following is the definition of struct sigaction:

struct sigaction {

void (\*sa\_handler)(int);

void (\*sa\_sigaction)(int, siginfo\_t \*, void \*);

sigset\_t sa\_mask;

int sa\_flags;

void (\*sa\_restorer)(void);

};

sa\_handler and sa\_sigaction are function pointers to the signal handler. Either can be used but not both. Note that sa\_handler has one input parameter (int), which will be the signal value. sa\_restorer is not intended for application use, so it ignored in our case.

**TCP Port Assignments To Students**

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 two port numbers according to the table below:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Name | Port # | Name | Port # | Name | Port# |
| Au, Jared | 3500  3600 | Leong, Alicia | 3514  3614 | Clemens, Scott | 3529  3629 |
| Butac, Jetro | 3501  3601 | Liang, David | 3515  3615 | Basbarro, Anthony | 3530  3630 |
| Calamasa, Marie Beth | 3502  3602 | Lin, Kylie | 3516  3616 | Tokita, Dylan | 3531  3631 |
| Canaday, Konapiliahi | 3503  3603 | Liu, Kevin | 3517  3617 | Vohra, Meenakshi | 3532  3632 |
| Casipit, Marionne | 3504  3604 | Orias, Jianna | 3518  3618 | You, Yeon Sang | 3533  3633 |
| Ereno, Aljaed | 3505  3605 | Salazar, Andrew | 3519  3619 |  | 3534  3634 |
| Fujimoto, Dean | 3506  3606 | Shimomura, Katrina | 3520  3620 |  | 3535  3635 |
| Fujitani, Alysha | 3507  3607 | Soto, Victoria | 3521  3621 |  | 3536  3636 |
| Ganiron, Ryan | 3508  3608 | Won, Christopher | 3522  3622 |  | 3537  3637 |
| Harris, Joseph | 3509  3609 | Yang, Alvin | 3523  3623 |  | 3538  3638 |
| Hwang, Lance | 3510  3610 | Yang, Thomas | 3524  3624 |  | 3539  3639 |
| Jorg, Nainoa | 3511  3611 | Ye, Japhet | 3525  3625 |  | 3540  3640 |
| Julian, Xandrew | 3512  3612 |  | 3527  3627 |  |  |
| Kwock, Kayla | 3513  3613 |  | 3528  3628 |  |  |

Use these ports for this lab and subsequent labs.

**Simple Exercises**

Do the following simple exercises.

Exercise 1: Attached is the client-server code client.c and server.c. Change the PORT numbers to one of your own. Then run the server in background, and then run the client. Afterwards, kill the server and make sure there are no leftover processes.

Exercise 2: Modify the client.c and server.c code to do the following. The client will send a text word string to the server. This text word will come from the command line.

The server will receive the string and display it on the terminal using something like

printf(“Server: received ‘%s’\n”, string);

Then it will convert all lower-case alphabets to upper-case alphabets, and send the word back to the client. The client will receive the converted word and display it on the terminal using something like

printf(“Client: received ‘%s’\n”, string);

Then the client terminates. You can now kill the server.

Here’s an example run of the client:

./client aloha

Server: received ‘aloha’

Client: received ‘ALOHA’

Hint: Recall that the letters ‘a’ through ‘z’ in ASCII is 0x61 through 0x7a in hexadecimal or 97 through 122 in decimal. The letters ‘A’ through ‘Z’ in ASCII is 0x41 through 0x5A in hexadecimal or 65 through 90 in decimal. See <http://www.asciitable.com/> So the conversion for each char is to subtract 32 if the char is within 97 and 122.

If you’re unfamiliar with inline arguments for main(), here’s some information:

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

void main(); // If nothing is returned by main

void main(void);

int main(int argc, char \*\*argv); // You can replace ‘int’ with ‘void’ as a return data type

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]);

}

}

argc is the number of arguments in the command line when the program was launched.

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

There are number of arguments is 6: ‘./a.out’, ‘-a’, ‘this’, ‘is’, ‘an’, and ‘example’. So argc will be 6. argv[0] is a pointer to a char string of the first argument ‘./a.out’ which is terminated by NULL, arg[1] is a pointer to the second argument, and so on.

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

**Assignment**

For this lab 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.

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

Here’s a suggestion of 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 using ‘send’ or ‘write’. It may include additional parameters, e.g., in the case of Download, it may include the file name to download:
      * 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” or “read”
    - After completing the command, it terminates the socket
* Server
  + The server listens to its socket for any request for connections from client
  + If there is a request, it accepts it and establishes a connection
  + The server creates a child to handle the client
  + The child receives a message (text string) from the client using ‘recv’ or ‘read’
  + 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. One way to do a check is to use Linux’s access()
    - 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.

Hints:

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. This is basically Exercise 1.

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:

Design command messages that a client can send to the server

Modify client so it sends a command message for the server to execute ‘ls’

Modify server so that it parses the command messages

Then it runs ‘ls’

The output is sent back to the client

Modify the client so that it displays what’s received from the server

Stage 4:

Modify client and server so that

Client prompts user for command

Client sends command messages

Server parses and acts on command message

Server sends output back to client

Client displays results from server

The client should have a user interface that accepts the commands to “list” or “quit”.

Stage 5:

The client and server should include the command “check”. Hint: see the Linux function “access”

Stage 6:

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. Hint: see the Linux function “cat”

Stage 7:

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 8:

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.