2019330021 김영우 Subject 53 - Project 1 Description

1. Execution environment and how to build the programs

Execution environment – Ubuntu 20.04.2 LTS, gcc 9.3.0

```
user@user-VirtualBox:~/53proj$ lsb_release -a
No LSB modules are available.
Distributor ID: Ubuntu
Description:    Ubuntu 20.04.2 LTS
Release:        20.04
Codename:       focal
```

```
user@user-VirtualBox:~/53proj$ gcc --version
gcc (Ubuntu 9.3.0-17ubuntu1~20.04) 9.3.0
Copyright (C) 2019 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

How to build:

server.c – `gcc -o server server.c`

client.c – `gcc -o client client.c`

or

`make`

How to execute

server - `./server <port number>`

client - `./client <ip address> <port number>`

```
user@user-VirtualBox:~/53proj/proj2$ make
gcc -o client client.c
gcc -o server server.c
user@user-VirtualBox:~/53proj/proj2$ ./server 2000
```

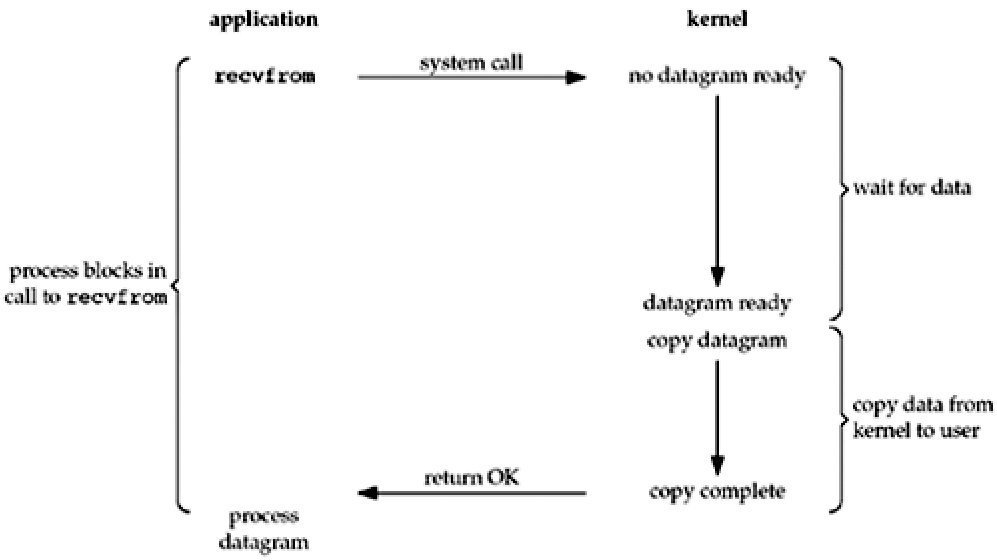
```
user@user-VirtualBox:~/53proj/proj2$ make
gcc -o client client.c
gcc -o server server.c
user@user-VirtualBox:~/53proj/proj2$ ./client 127.0.0.1 2000
```

2. Comparison of the I/O models

-blocking I/O, non-blocking I/O, I/O multiplexing

1) Blocking I/O Model

The most prevalent model for I/O is the *blocking I/O model*, which we have used for all our examples so far in the text. By default, all sockets are blocking. Using a datagram socket for our examples, we have the scenario shown in Figure 1.



<Figure 1> blocking I/O model

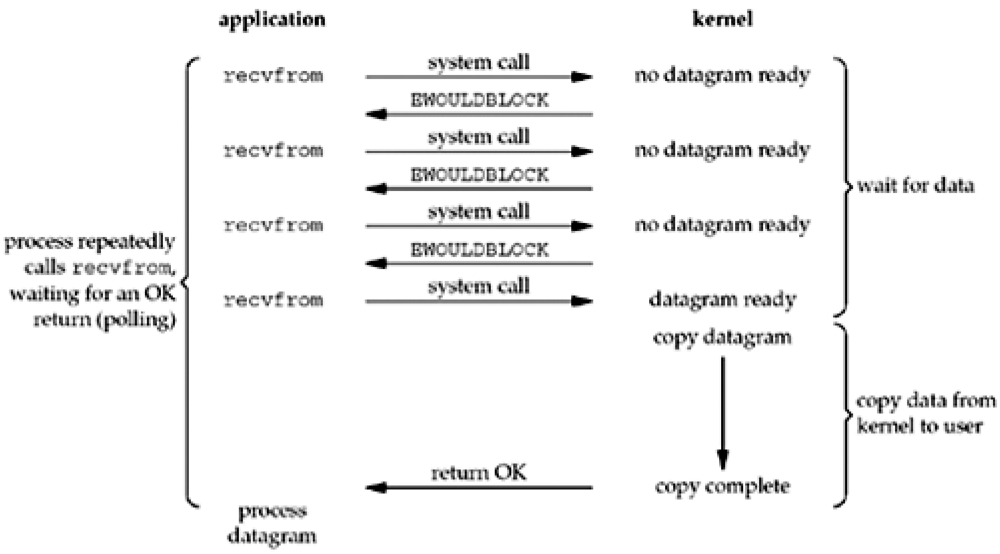
We use UDP for this example instead of TCP because with UDP, the concept of data being "ready" to read is simple: either an entire datagram has been received or it has not. With TCP it gets more complicated, as additional variables such as the socket's low-water mark come into play.

In the examples in this section, we also refer to recvfrom as a system call because we are differentiating between our application and the kernel. Regardless of how recvfrom is implemented (as a system call on a Berkeley-derived kernel or as a function that invokes the getmsg system call on a System V kernel), there is normally a switch from running in the application to running in the kernel, followed at some time later by a return to the application.

In Figure 1, the process calls recvfrom and the system call does not return until the datagram arrives and is copied into our application buffer, or an error occurs. The most common error is the system call being interrupted by a signal. We say that our process is *blocked* the entire time from when it calls recvfrom until it returns. When recvfrom returns successfully, our application processes the datagram.

2) Nonblocking I/O Model

When we set a socket to be nonblocking, we are telling the kernel "when an I/O operation that I request cannot be completed without putting the process to sleep, do not put the process to sleep, but return an error instead.". Figure 2 shows a summary of the example we are considering.



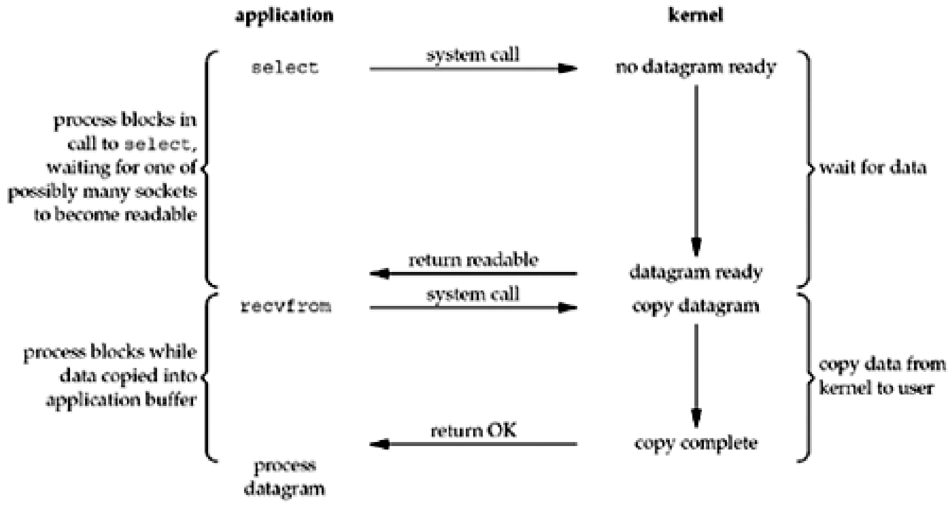
<Figure 2> non-blocking I/O model

The first three times that we call recvfrom, there is no data to return, so the kernel immediately returns an error of EWOULDBLOCK instead. The fourth time we call recvfrom, a datagram is ready, it is copied into our application buffer, and recvfrom returns successfully. We then process the data.

When an application sits in a loop calling recvfrom on a nonblocking descriptor like this, it is called *polling*. The application is continually polling the kernel to see if some operation is ready. This is often a waste of CPU time, but this model is occasionally encountered, normally on systems dedicated to one function.

3) I/O Multiplexing Model

With *I/O multiplexing*, we call select or poll and block in one of these two system calls, instead of blocking in the actual I/O system call. Figure 3 is a summary of the I/O multiplexing model.

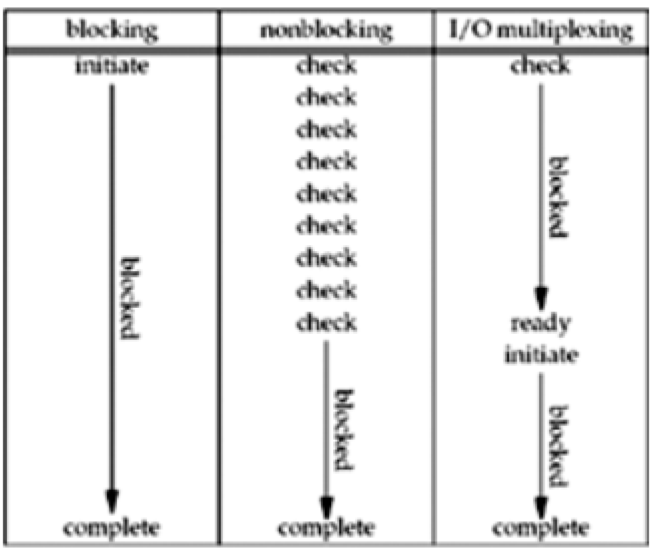


<Figure 3> I/O multiplexing model

We block in a call to select, waiting for the datagram socket to be readable. When select returns that the socket is readable, we then call recvfrom to copy the datagram into our application buffer.

Comparing Figure 3 to Figure 1, there does not appear to be any advantage, and in fact, there is a slight disadvantage because using select requires two system calls instead of one. But the advantage in using select, which we will see later in this chapter, is that we can wait for more than one descriptor to be ready.

Another closely related I/O model is to use multithreading with blocking I/O. That model very closely resembles the model described above, except that instead of using select to block on multiple file descriptors, the program uses multiple threads (one per file descriptor), and each thread is then free to call blocking system calls like recvfrom.



<Figure 4> comparison of blocking model, nonblocking model, multiplexing I/O model.

3. Comments for each line of the essential part of my code

1)server.c

```
#include <stdio.h>
#include <unistd.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <arpa/inet.h>
#include <stdlib.h>
#include <string.h>
#include <sys/select.h>
#define MAX 1024 //max client number(=FD_SETSIZE)
#define BUF_LEN 1024 //max buffer length
int main(int argc, char *argv[]) {
	//argv error handling
	if(argc!=2){
		printf("usage : %s <port> \n" , argv[0]);
		exit(1);
	}
	//create IPv4, TCP socket
	int sockfd = socket(AF_INET,SOCK_STREAM,0);
	if(sockfd == -1) { //socket create error handler
		printf("create socket fail\n");
		return -1;
	}
	else {
		printf("creat socket success , sockfd = %d\n",sockfd);
	}
	//initialize server address information
	struct sockaddr_in seraddr,cliaddr;
	seraddr.sin_family = AF_INET;
	seraddr.sin_port = htons(atoi(argv[1]));
	seraddr.sin_addr.s_addr = htonl(INADDR_ANY);
	memset(seraddr.sin_zero,0,8);

	//allocate server address
	socklen_t len = sizeof(struct sockaddr);
	int bindret = bind(sockfd,(struct sockaddr *)&seraddr,len);
	if(bindret == -1) { //bind error handler
		printf("bind fail\n");
		close(sockfd);
		return -2;
	}
	else {
		printf("bind success\n");
	}

	//listen for client(max number 1024-FD_SETSIZE)
	int listenret = listen(sockfd,MAX);
	if(listenret == -1) { //listen error handler
		printf("listen fail\n");
		close(sockfd);
		return -3;
	}
	else {
		printf("listen success\n");
	}
	//fd_set, current max file descriptor
	fd_set read,readset;
	int maxfd = sockfd;
	//initialize readset&set sockfd to 1
	FD_ZERO(&readset);
	FD_SET(sockfd,&readset);
```

```c
	while(1) {
		//clientfd
		int confd;
		read = readset;
		//select which file descriptor has changed
		int selectret = select(maxfd+1,&read,NULL,NULL,NULL);
		if(selectret == -1) { //select error handler
			printf("select fail\n");
			close(sockfd);
			return -4;
		}

		//if socket for tcp request has changed
		if(FD_ISSET(sockfd,&read)) {
			//accept client&allocate confd to client
			confd = accept(sockfd,(struct sockaddr *)&cliaddr,&len);
			printf("connect client(%s) success\nconfd
= %d\n",inet_ntoa(cliaddr.sin_addr),sockfd);
			//update maxfd
			if(maxfd < confd) {
				maxfd = confd;
			}
			//set confd in readset
			FD_SET(confd,&readset);

			//if sockfd is the only file descriptor changed
			if(selectret == 1) {
				continue;
			}
		}
		//message from client
		char buf[BUF_LEN];
		int i;

		//for the rest of file descriptors(client's file descriptor)
		for(i = sockfd+1;i<=maxfd;i++) {
			memset(buf,0,BUF_LEN);

			//if client's file descriptor has changed
			if(FD_ISSET(i,&read)) {
				if(recv(i,buf,BUF_LEN,0) == 0) { //if end of data
					FD_CLR(i,&readset); //clear file descriptor
					printf("----------client(%s) quit---------
\n",inet_ntoa(cliaddr.sin_addr));

					continue;
				}
				else {
					printf("client(%d) send message\n",i);
					send(i,buf,BUF_LEN,MSG_CONFIRM); //echo buf
to client

					if(strncmp(buf,"end",3) == 0) { //end of
connection

						FD_CLR(i,&readset); //clear file
descriptor

						printf("----------client(%s) quit--
-------\n",inet_ntoa(cliaddr.sin_addr));

						continue;
					}
				}
			}
		}
	}
	return 0;
}
```

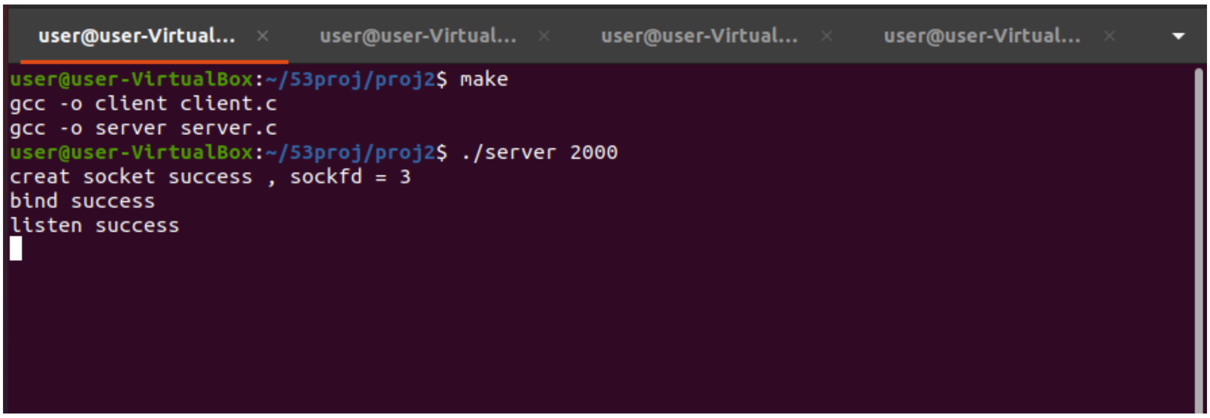
2)client.c

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <arpa/inet.h>
#include <string.h>
#include <stdlib.h>
#include <sys/select.h>
#define BUF_LEN 1024 // max buffer length
int main(int argc, char *argv[])
{
	//argv error handling
	if (argc != 3) {
	printf("usage : %s <ip> <port> \n" , argv[0]);
	exit(1);
	}

	//create IPv4, TCP socket
	int sockfd = socket(AF_INET,SOCK_STREAM,0);
	if(sockfd < 0) { // socket creation error handler
		printf("creat socket fail\n");
		return -1;
	}
	else {
		printf("creat socket success , sockfd = %d\n",sockfd);
	}
	//initialize server address information
	struct sockaddr_in cliaddr;
	cliaddr.sin_family = AF_INET;
	cliaddr.sin_port = htons(atoi(argv[2]));
	cliaddr.sin_addr.s_addr = inet_addr(argv[1]);
	memset(cliaddr.sin_zero,0,8);
	//connect request
	int confd = connect(sockfd,(struct sockaddr *)&cliaddr,sizeof(struct sockaddr));
	if(confd < 0) { //connection error handler
		printf("connect fail\n");
		close(sockfd);
		return -2;
	}
	else {
		printf("----------connect success-----------\nsockfd = %d\n",sockfd);
	}

	// message buffer
	char buf[BUF_LEN];
	int str_len,input_len;

	//fd_set&initialize readset&add stdin/sockfd in readset
	fd_set reads, readset;
	FD_ZERO(&readset);
	FD_SET(sockfd,&readset);
	FD_SET(0,&readset);

	while(1) {
		reads=readset;

		//select which file descriptor has changed
		if(select(sockfd+1,&reads,NULL,NULL,NULL)==-1) { //select error handler
			printf("select() : error\n");
			exit(-1);
		}

		//if stdin has changed
```

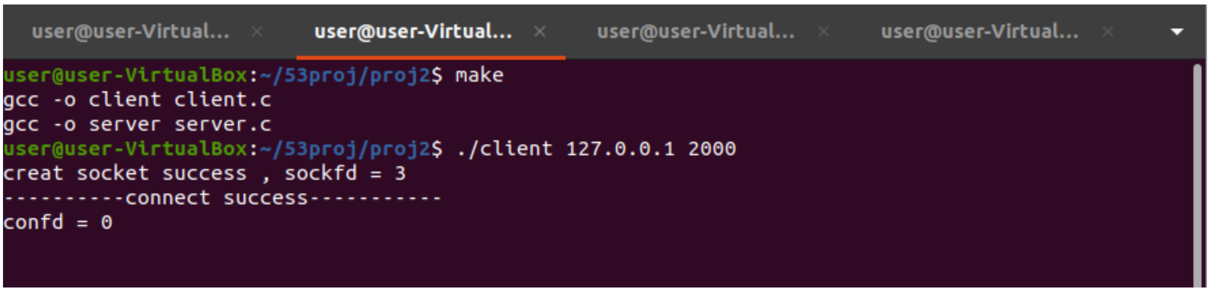
```
			if(FD_ISSET(0,&reads)) {
				memset(buf,0,BUF_LEN);
				input_len=read(0,buf,BUF_LEN); //read buffer from stdin
				send(sockfd,buf,strlen(buf),0); //send it to the server
			}

			//if sockfd has changed(recv from server)
			if(FD_ISSET(sockfd,&reads)) {
				if(str_len=recv(sockfd,buf,BUF_LEN,0)) { //received buffer from server
					buf[str_len]='\0';
					printf("echo - ");
					fputs(buf, stdout); // echo it
				}
				else { // no response from server
					FD_CLR(sockfd,&readset); //clear sockfd from readset
					printf("server closed\n");
					close(sockfd);
					break;
				}
			}
		}
		close(sockfd);
		return 0;
}
```

4. Execution result analysis

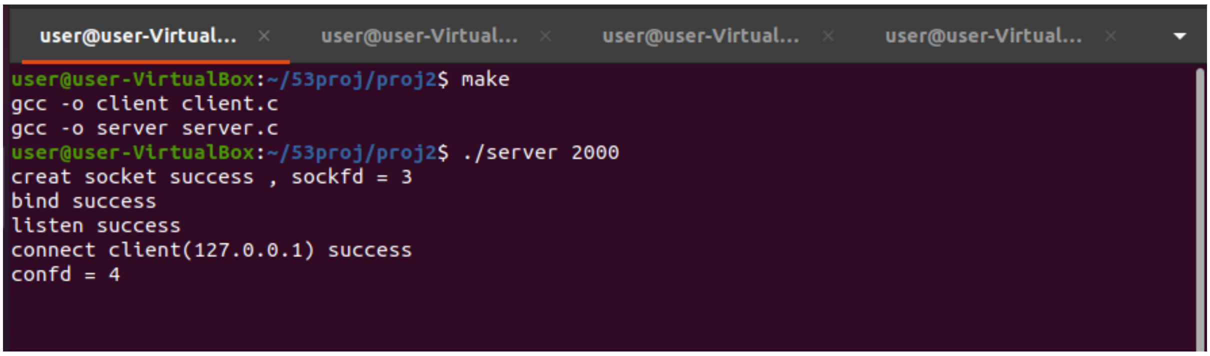
portnum=2000



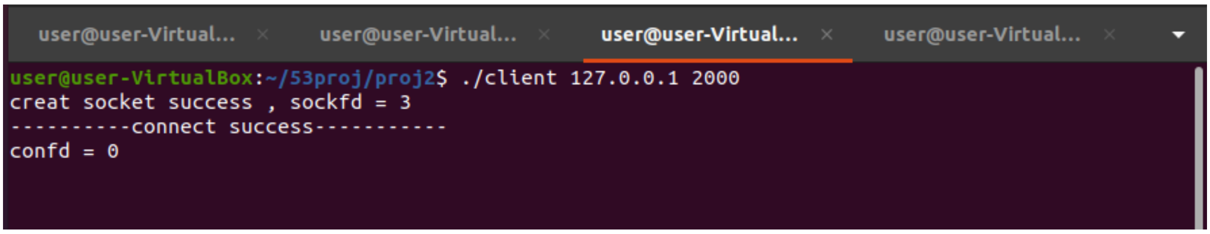
<Figure 5> Server – Terminal 1



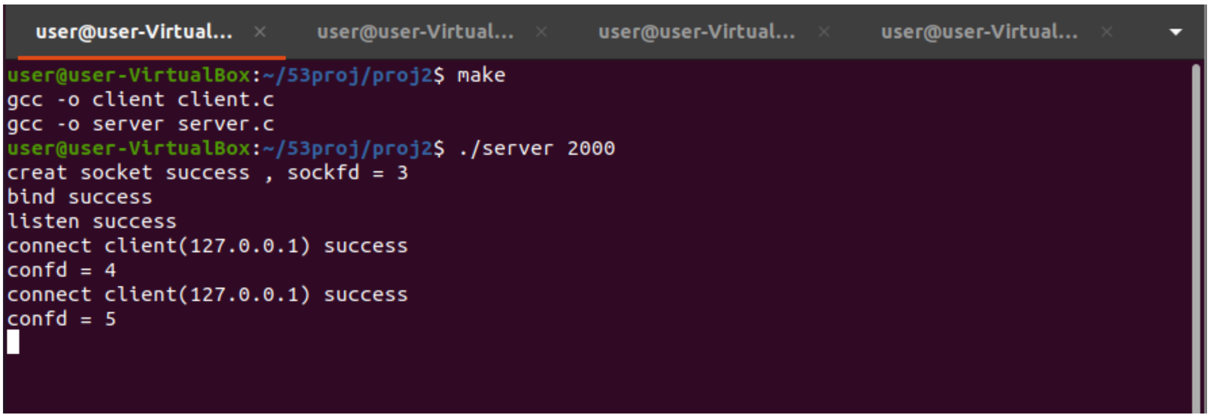
<Figure 6> First client – Terminal 2



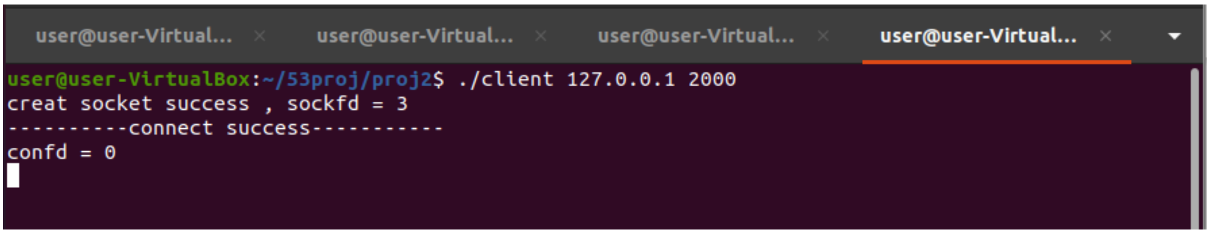
<Figure 7> After connection with client1 – Terminal 1



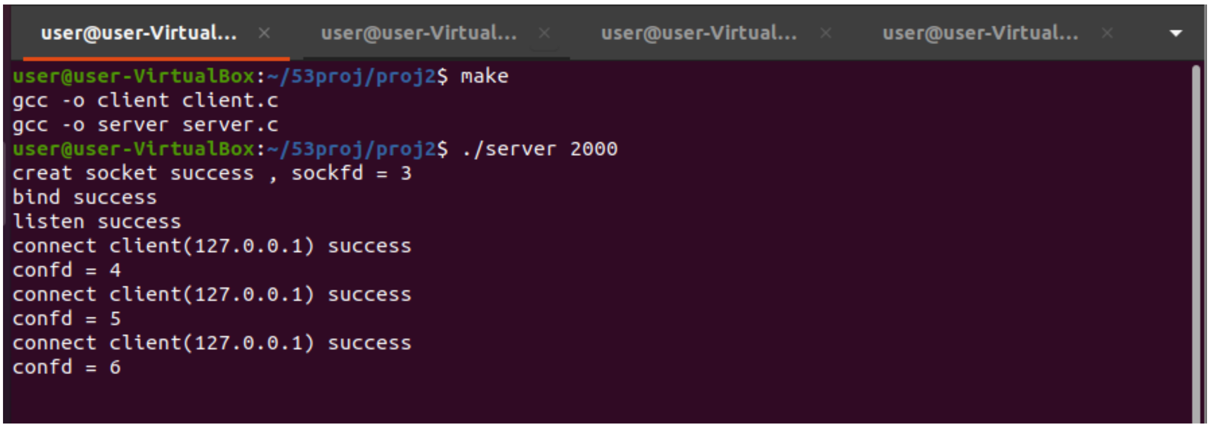
<Figure 8> Second client– Terminal 3



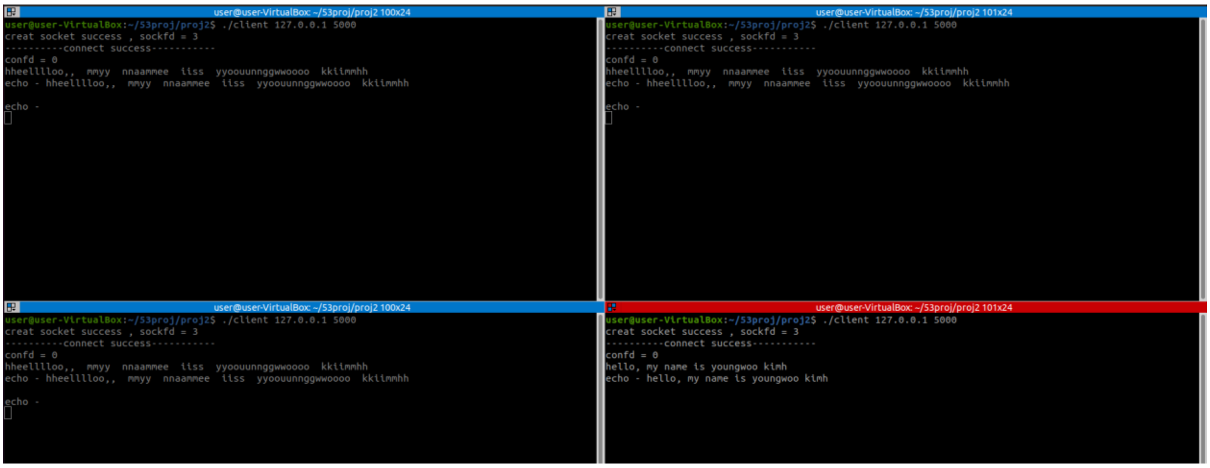
<Figure 9> Server program after connection of second client – Terminal 1



<Figure 10> Third client– Terminal 4



<Figure 9> Server program after connection of third client – Terminal 1



<Figure 10> Sending message simultaneously using terminator

<Figure 11> Server program after sending message – Terminal 1

As you can see, the server successfully received simultaneous message from multiple clients. Also, the server and the clients remain online after sending message.