**XV6 file system checker:**

<http://pages.cs.wisc.edu/~remzi/Classes/537/Spring2016/Projects/p5a.html>

For this project, we will use the xv6 file system image as the basic image that we will be reading and checking. The file **include/fs.h** includes the basic structures you need to understand, including the superblock, on disk inode format (struct dinode), and directory entry format (struct dirent). The tool **tools/mkfs.c** will also be useful to look at, in order to see how an empty file-system image is created.

Make sure to look at **fs.img** , which is a file system image created when you make xv6 by the tool **mkfs** (found in the tools/ directory of xv6). The output of this tool is the file **fs.img** and it is a consistent file-system image

Steps:

1. $ git clone <https://github.com/gauthamsunjay/filesystemchecker.git>
2. $ cd filesystemchecker
3. $ ls

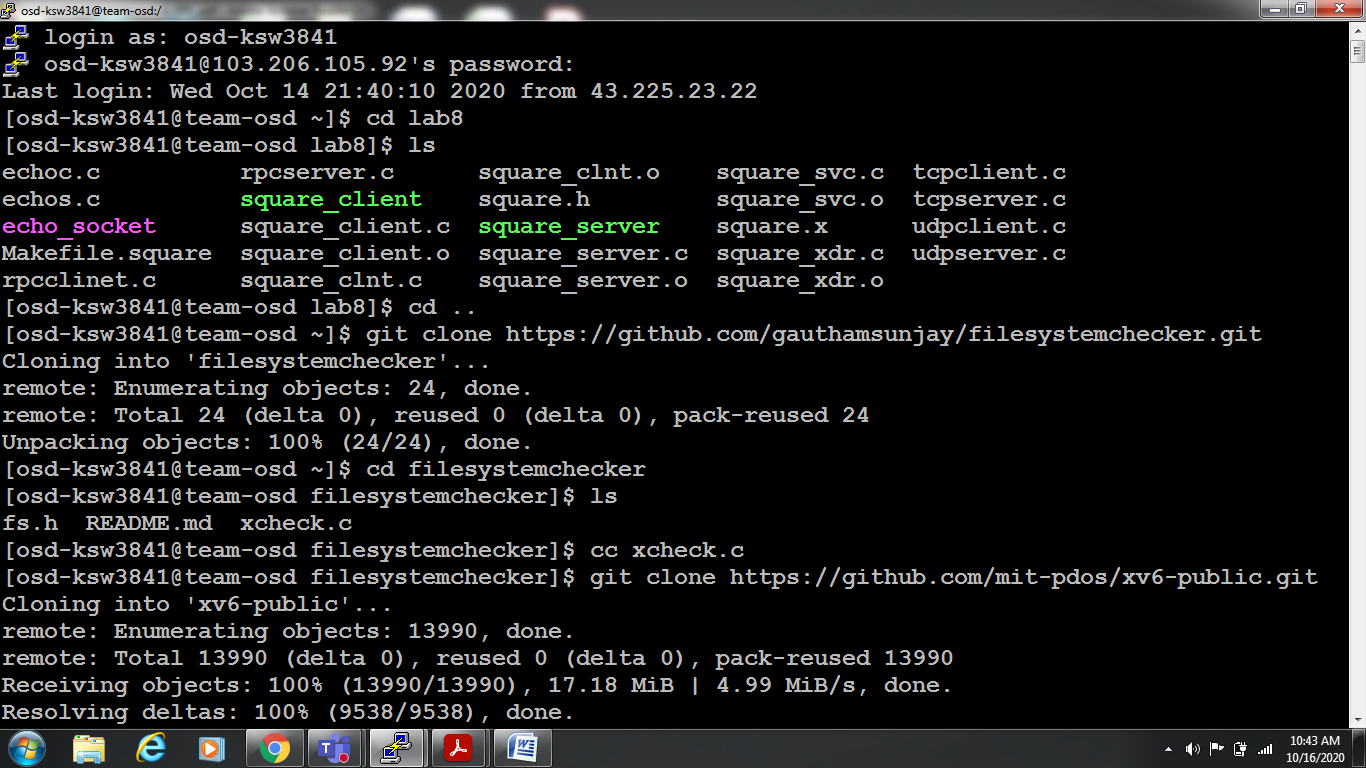
fs.h README.md xcheck.c

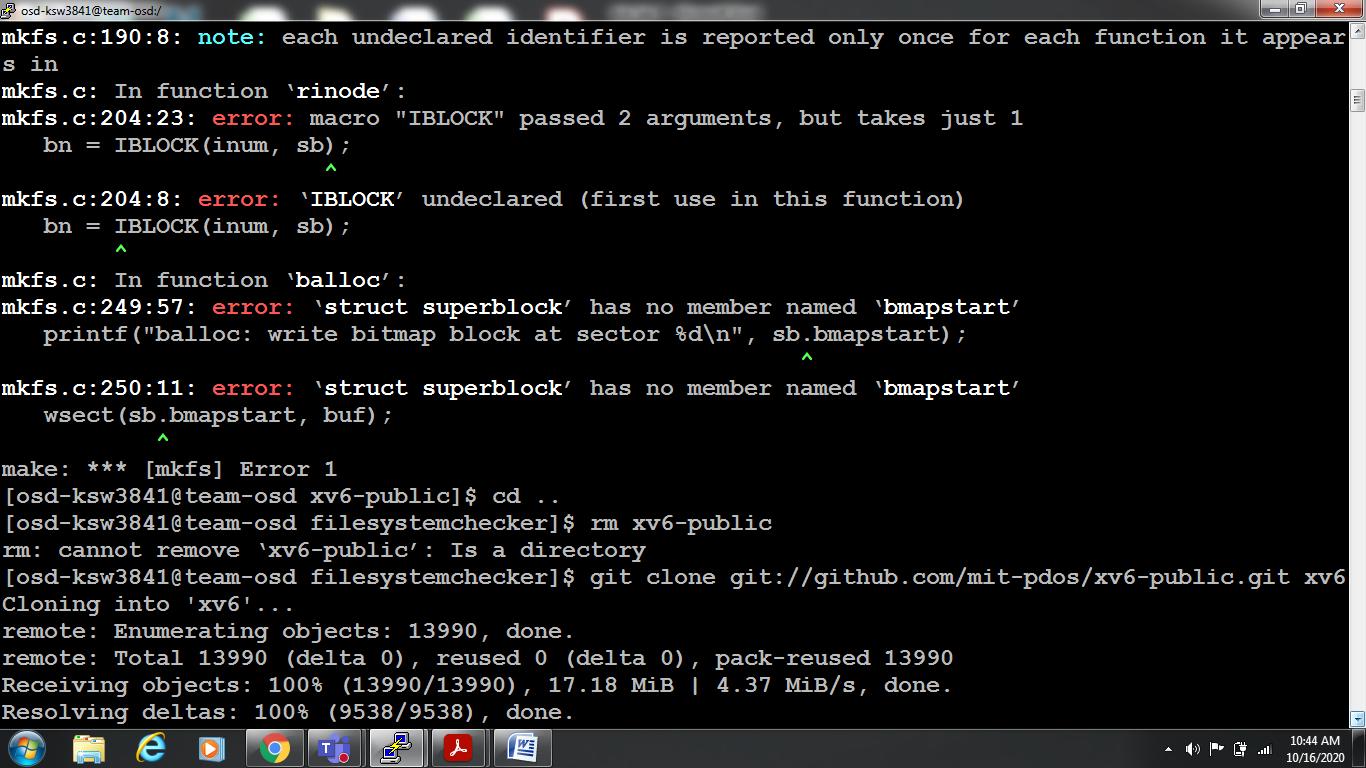
1. $ cc xcheck.c
2. $ git clone git://github.com/mit-pdos/xv6-public.git xv6
3. $ cd xv6
4. $ make qemu-nox
5. type ctrl a, x
6. $ cd ..
7. $ cp fs.h ./xv6
8. $ cp xcheck.c ./xv6
9. $ cd xv6
10. $ cc xcheck.c
11. $ ls

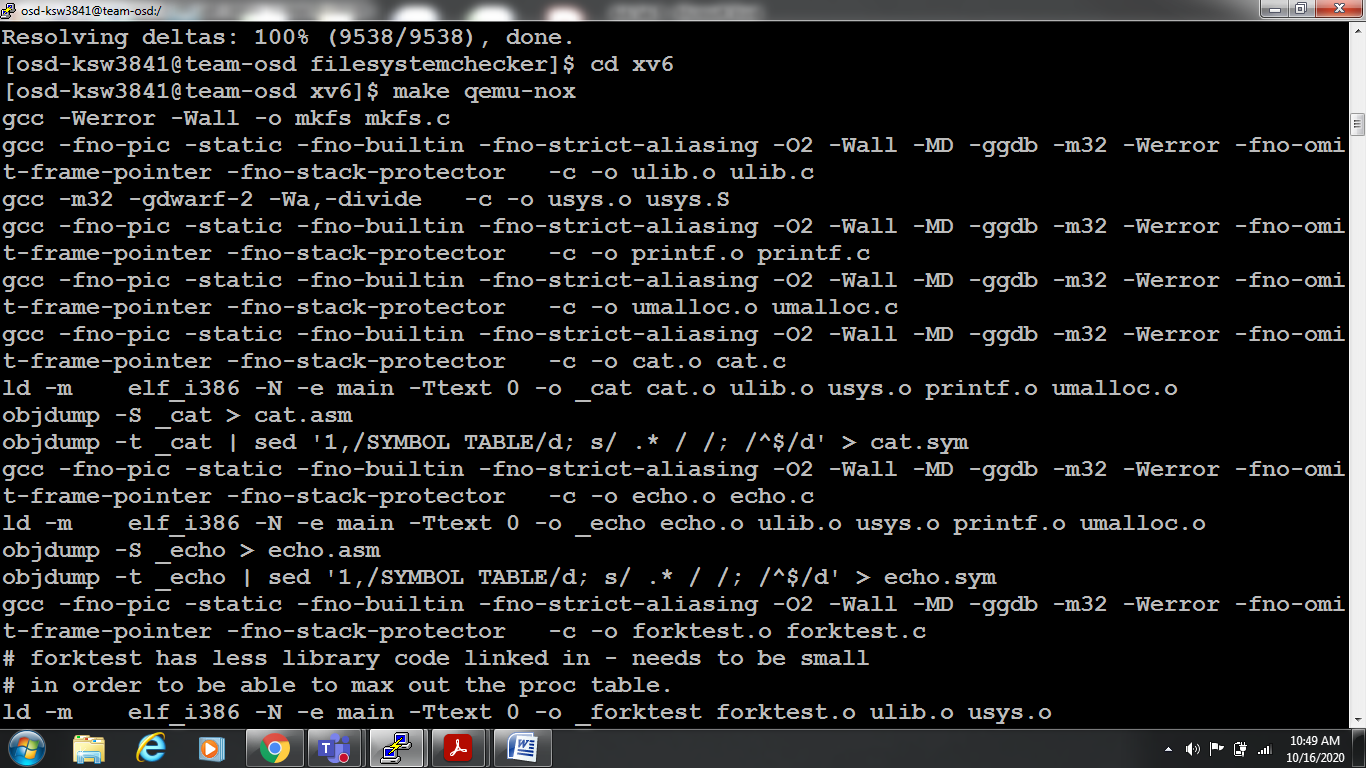
When we type ls we can see fs.img file which has been created

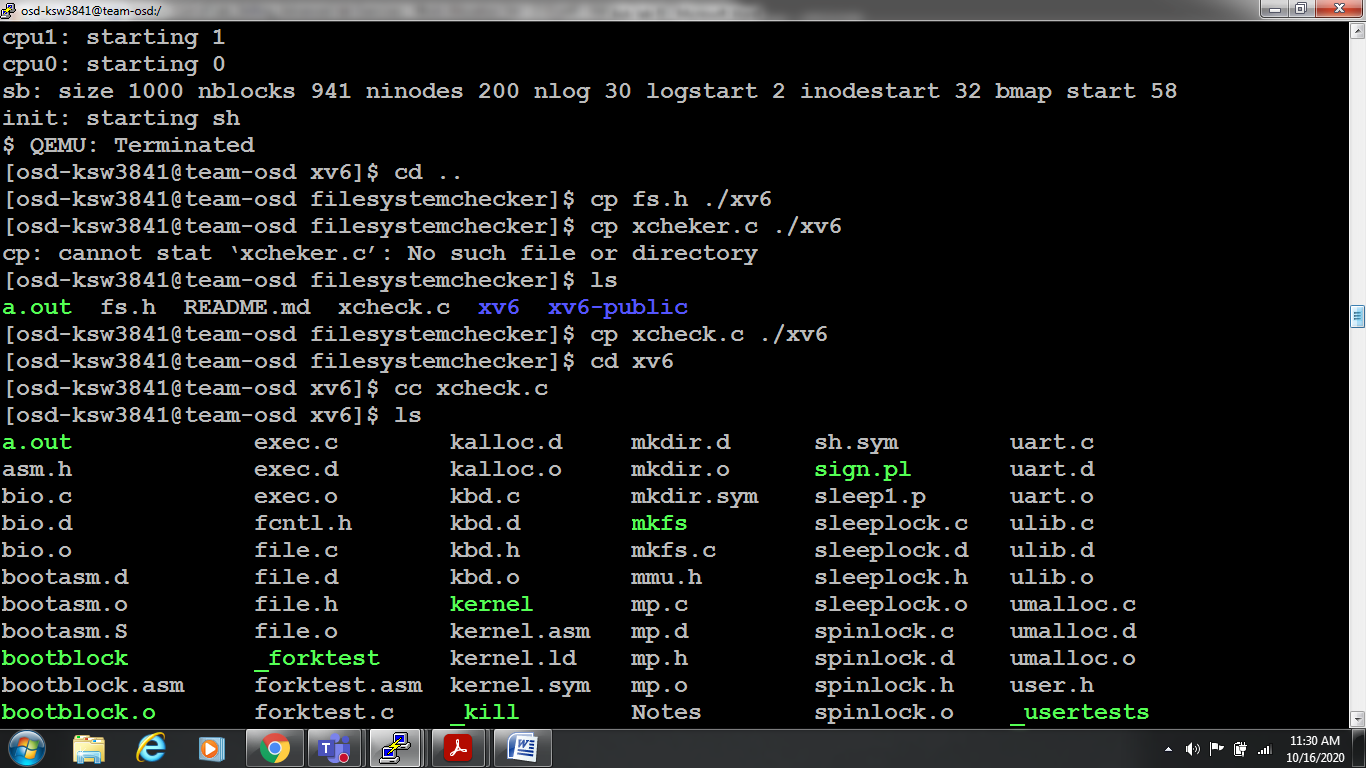
1. $ ./a.out fs.img

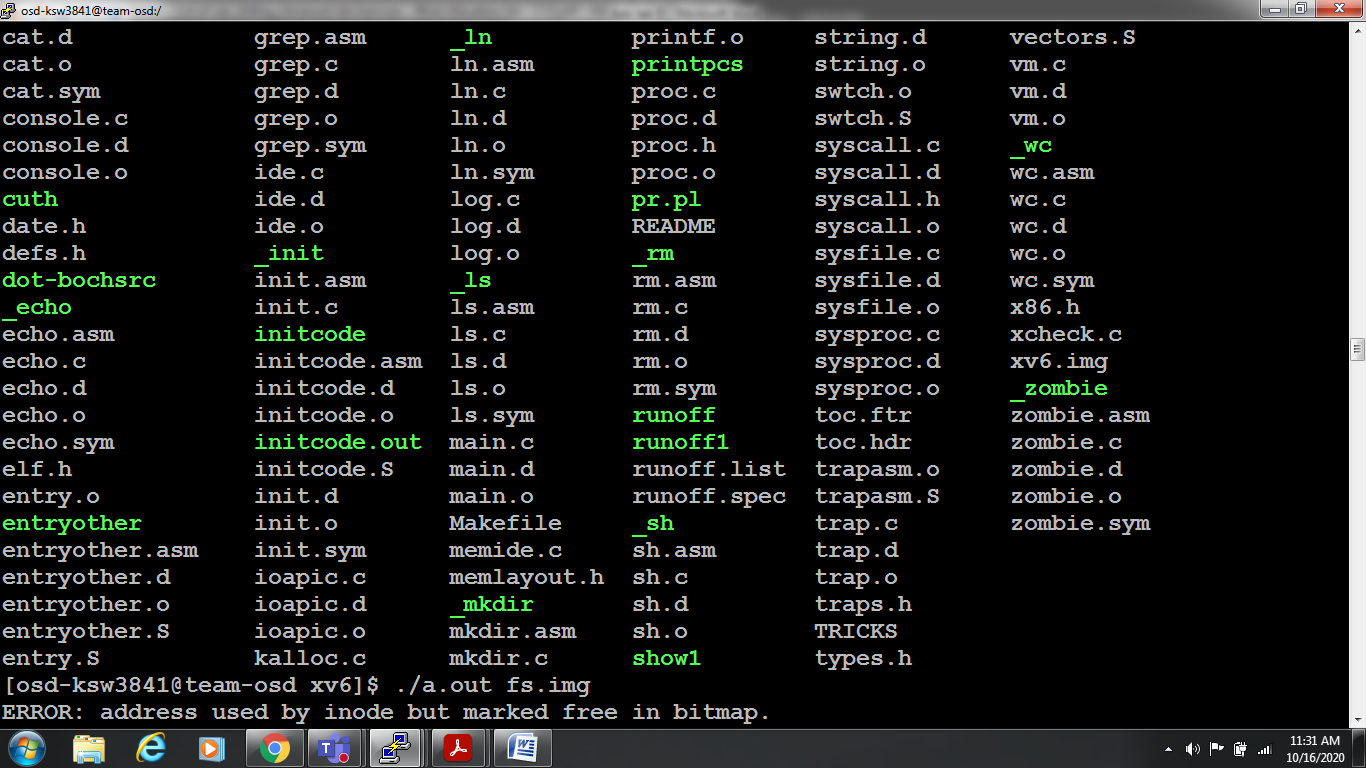
It prints “ERROR:address used by inode but marked free in bitmap” as one inode is not enabled.

****

****

****

****

****

**dup2:**

Here we are using dup2\_test.c file which includes two arguments as first one is oldfile and second one is newfile. So oldfile has file descriptor-3 and newfile has file descriptor-2. So the variables are assigned as origfd=3 , newfd=2.

By using write system call, we are assigning data into newfd i.e second file in arguments as ignored

write(newfd, "ignored\n", 8);

whereas by using the write system call we are assigning data into oldfd i.e first file in arguments as foo

write(origfd, "foo", 3);

by using dup2 we are assigning both origfd and newfd as 3

|  |
| --- |
|  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  | if (dup2(origfd, newfd) < 0) { |
|  | printf(2, "dup2 error\n"); |
|  | } |
|  |  |

then we are closing newfd

using another write system call we are appeding content to oldfd so the oldfd will be “foobar” as bar is appended to previos data foo.

So finally the contents in the two files will be as

Oldfd is foobar and newfd is ignored.

Commands to execute dup2:

1. $ git clone <https://github.com/williamsandrew/xv6.git> xv6dup2
2. $ cd xv6dup2
3. $ make qemu-nox
4. $ ls where dup2\_test will appear
5. $ dup2\_test f1 f2

Here f1 is old file and f2 is new file

1. $ ls

Displays both files f1 and f2

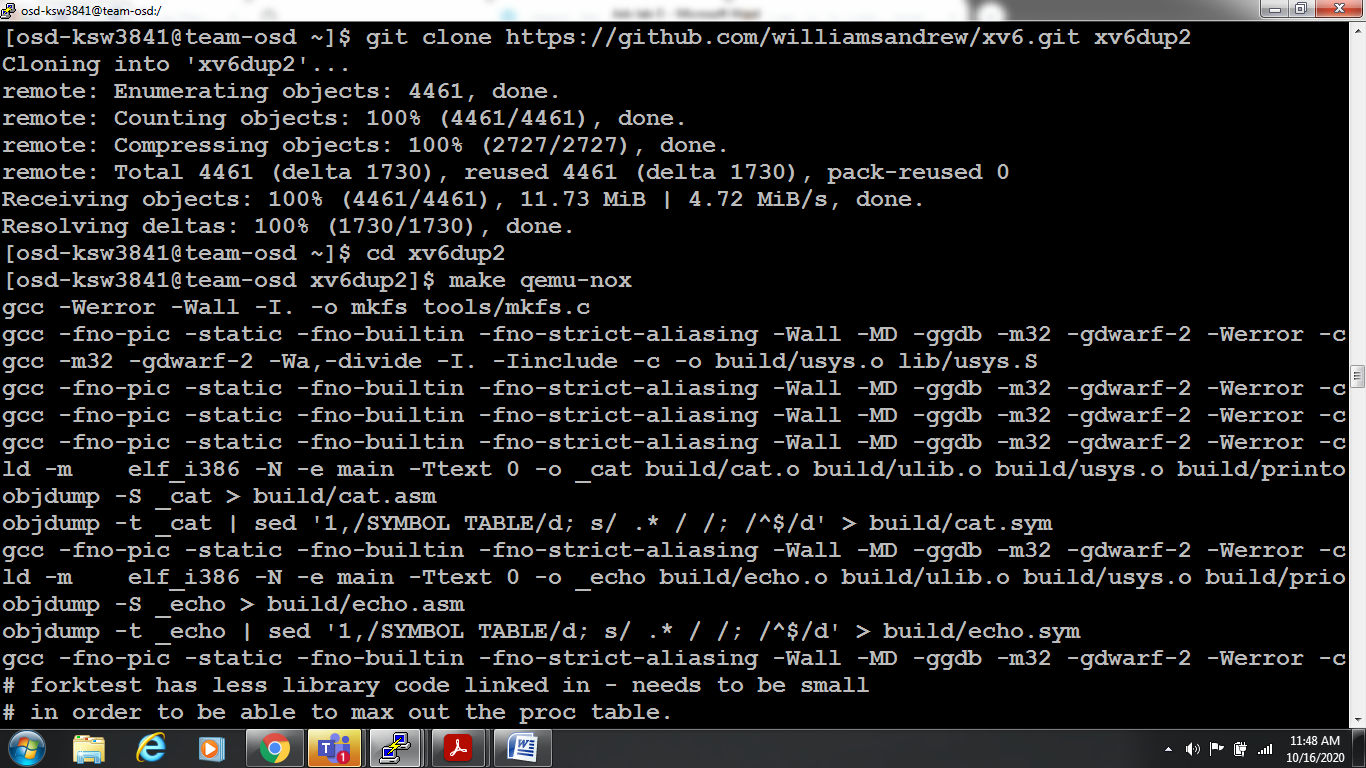
1. $ cat f1

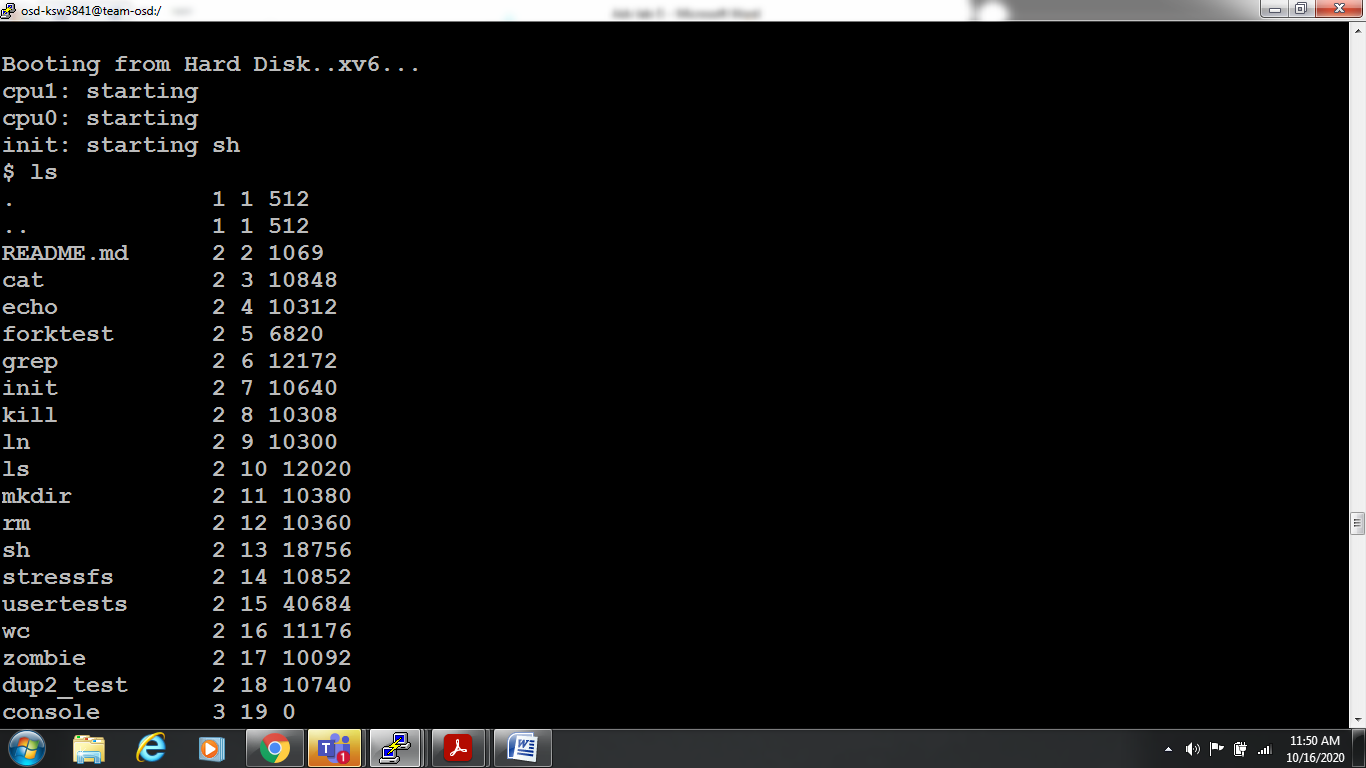
Prints foobar

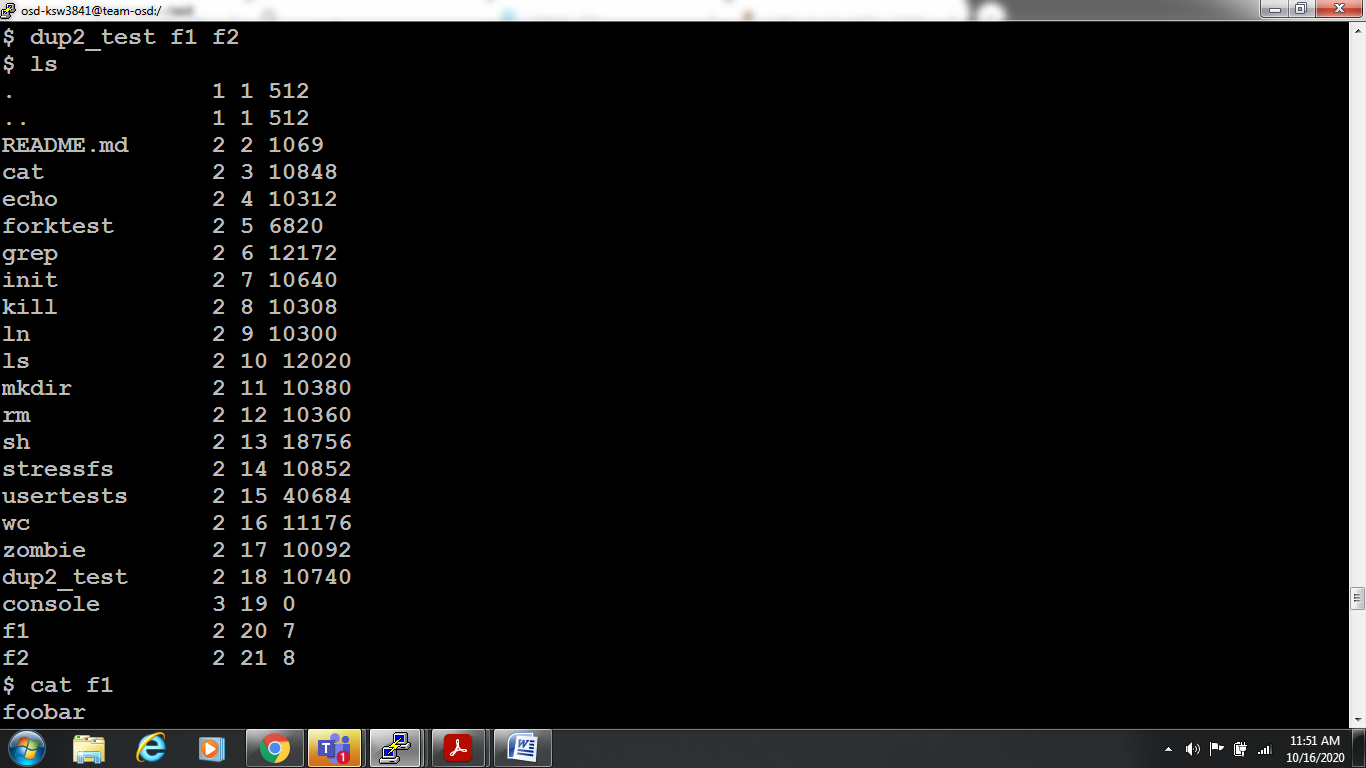
1. $ cat f2

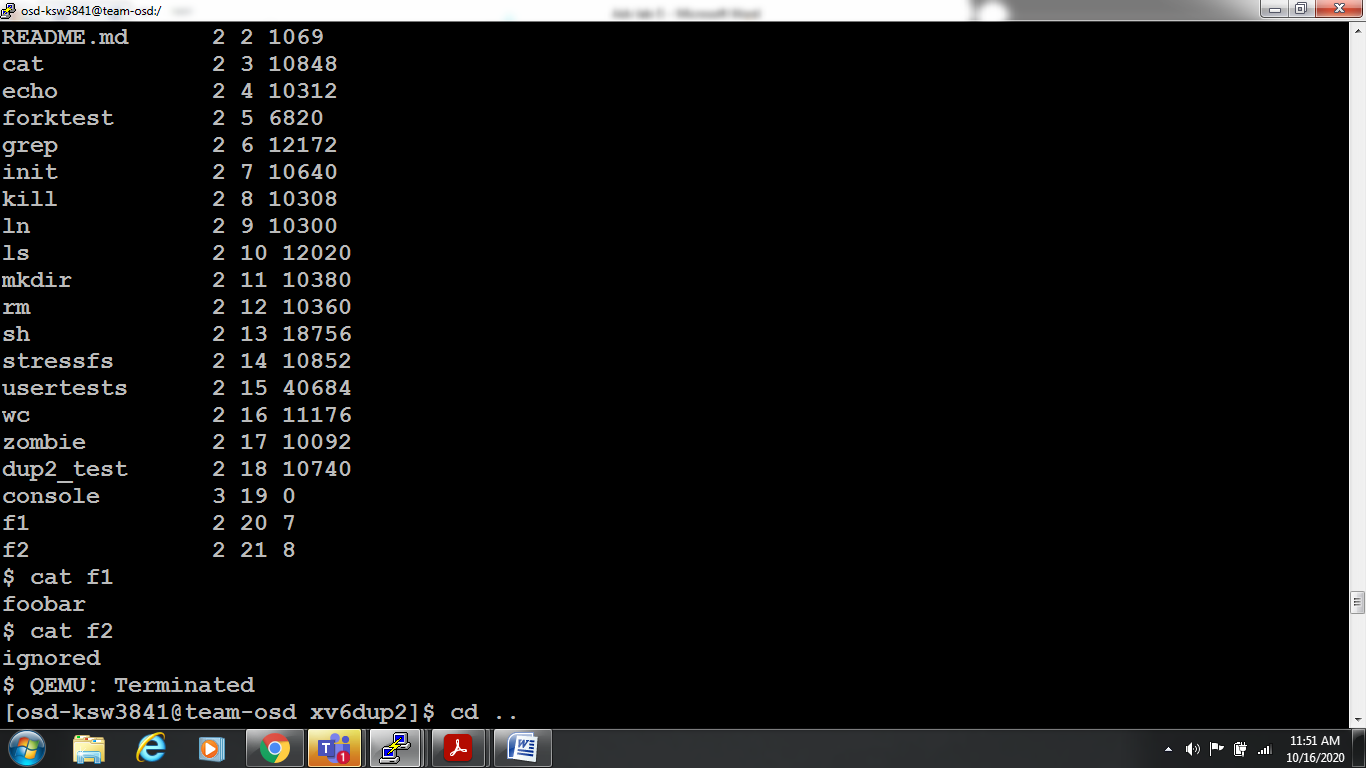
Prints ignored

1. Type ctrl a,x









**Unix Sockets:**

Open two terminals in order to execute client in one terminal and server in another terminal.

**Echo Server**

/\* echos.c -- the echo server for echoc.c; demonstrates unix sockets \*/

#include <stdio.h>

#include <stdlib.h>

#include <errno.h>

#include <string.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <sys/un.h>

#define SOCK\_PATH "echo\_socket"

int main(void)

{ int s, s2, t, len;

struct sockaddr\_un local, remote;

char str[100];

if ((s = socket(AF\_UNIX, SOCK\_STREAM, 0)) == -1) {

perror("socket");

exit(1);

}

local.sun\_family = AF\_UNIX;

strcpy(local.sun\_path, SOCK\_PATH);

unlink(local.sun\_path);

len = strlen(local.sun\_path) + sizeof(local.sun\_family);

if (bind(s, (struct sockaddr \*)&local, len) == -1) {

perror("bind");

exit(1);

}

if (listen(s, 5) == -1) {

perror("listen");

exit(1);

}

for(;;) {

int done, n;

printf("Waiting for a connection...\n");

t = sizeof(remote);

if ((s2 = accept(s, (struct sockaddr \*)&remote, &t)) == -1) {

perror("accept");

exit(1);

}

printf("Connected.\n");

done = 0;

do {

n = recv(s2, str, 100, 0);

if (n <= 0) {

if (n < 0) perror("recv");

done = 1;

}

if (!done)

if (send(s2, str, n, 0) < 0) {

perror("send");

done = 1;

}

} while (!done);

close(s2);

}

return 0;

}

Output: $ gcc echos.c

$ ./a.out

Waiting for a connection...

Connected

^C

**Echo client**

/\* echoc.c -- the echo client for echos.c; demonstrates unix sockets \*/

#include <stdio.h>

#include <stdlib.h>

#include <errno.h>

#include <string.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <sys/un.h>

#define SOCK\_PATH "echo\_socket"

int main(void)

{ int s, t, len;

struct sockaddr\_un remote;

char str[100];

if ((s = socket(AF\_UNIX, SOCK\_STREAM, 0)) == -1) {

perror("socket");

exit(1);

}

printf("Trying to connect...\n");

remote.sun\_family = AF\_UNIX;

strcpy(remote.sun\_path, SOCK\_PATH);

len = strlen(remote.sun\_path) + sizeof(remote.sun\_family);

if (connect(s, (struct sockaddr \*)&remote, len) == -1) {

perror("connect");

exit(1);

}

printf("Connected.\n");

while(printf("> "), fgets(str, 100, stdin), !feof(stdin)) {

if (send(s, str, strlen(str), 0) == -1) {

perror("send");

exit(1);

}

if ((t=recv(s, str, 100, 0)) > 0) {

str[t] = '\0';

printf("echo> %s", str);

} else {

if (t < 0) perror("recv");

else printf("Server closed connection\n");

exit(1);

}

}

close(s);

return 0;

}

Output: $ gcc echoc.c

$ ./a.out

Trying to connect...

Connected.

> hello

echo> hello

> how are you

echo> how are you

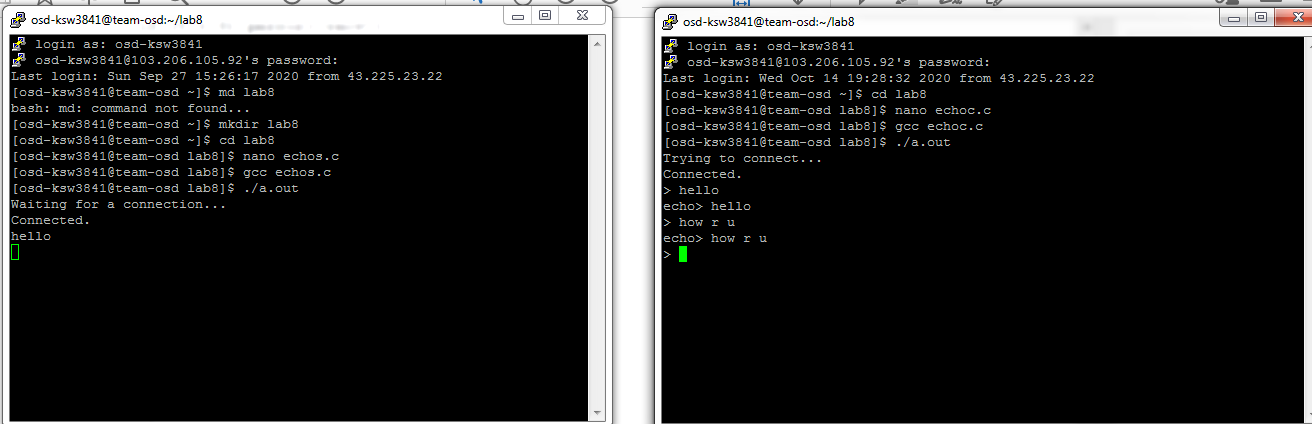
> quit

echo> quit

> q

echo> q

> ^C



**UDP sockets:**

UDP sockets or Datagram sockets are different from the TCP sockets in a number of ways. The most important difference is that UDP sockets are not connection oriented. More technically speaking, a UDP server does not **accept** connections and a udp client does not **connect** to server. The server will bind and then directly receive data and the client shall directly send the data.

**/\* Simple udp server \*/**

#include<stdio.h> //printf

#include<string.h> //memset

#include<stdlib.h> //exit(0);

#include<arpa/inet.h>

#include<sys/socket.h>

#define BUFLEN 512 //Max length of buffer

#define PORT 8888 //The port on which to listen for incoming data

void die(char \*s)

{ perror(s);

exit(1);

}

int main(void)

{ struct sockaddr\_in si\_me, si\_other;

int s, i, slen = sizeof(si\_other) , recv\_len;

char buf[BUFLEN];

//create a UDP socket

if ((s=socket(AF\_INET, SOCK\_DGRAM, IPPROTO\_UDP)) == -1)

{ die("socket"); }

// zero out the structure

memset((char \*) &si\_me, 0, sizeof(si\_me));

si\_me.sin\_family = AF\_INET;

si\_me.sin\_port = htons(PORT);

si\_me.sin\_addr.s\_addr = htonl(INADDR\_ANY);

//bind socket to port

if( bind(s , (struct sockaddr\*)&si\_me, sizeof(si\_me) ) == -1)

{ die("bind"); }

//keep listening for data

while(1)

{

printf("Waiting for data...");

fflush(stdout);

//try to receive some data, this is a blocking call

if ((recv\_len = recvfrom(s, buf, BUFLEN, 0, (struct sockaddr \*) &si\_other, &slen)) == -1)

{ die("recvfrom()"); }

//print details of the client/peer and the data received

printf("Received packet from %s:%d\n", inet\_ntoa(si\_other.sin\_addr), ntohs(si\_other.sin\_port));

printf("Data: %s\n" , buf);

//now reply the client with the same data

if (sendto(s, buf, recv\_len, 0, (struct sockaddr\*) &si\_other, slen) == -1)

{ die("sendto()"); }

}

close(s);

return 0;

}

Output: $ gcc udpserver.c

$ ./a.out

Waiting for data...Received packet from 127.0.0.1:23806

Data: hello

Waiting for data...Received packet from 127.0.0.1:23806

Data: how are you

^c

**/\* Simple udp client \*/**

#include<stdio.h> //printf

#include<string.h> //memset

#include<stdlib.h> //exit(0);

#include<arpa/inet.h>

#include<sys/socket.h>

#define SERVER "127.0.0.1"

#define BUFLEN 512 //Max length of buffer

#define PORT 8888 //The port on which to send data

void die(char \*s)

{ perror(s);

exit(1);

}

int main(void)

{ struct sockaddr\_in si\_other;

int s, i, slen=sizeof(si\_other);

char buf[BUFLEN];

char message[BUFLEN];

if ( (s=socket(AF\_INET, SOCK\_DGRAM, IPPROTO\_UDP)) == -1)

{ die("socket"); }

memset((char \*) &si\_other, 0, sizeof(si\_other));

si\_other.sin\_family = AF\_INET;

si\_other.sin\_port = htons(PORT);

if (inet\_aton(SERVER , &si\_other.sin\_addr) == 0)

{ fprintf(stderr, "inet\_aton() failed\n");

exit(1);

}

while(1)

{ printf("Enter message : ");

gets(message);

//send the message

if (sendto(s, message, strlen(message) , 0 , (struct sockaddr \*) &si\_other, slen)==-1)

{ die("sendto()"); }

//receive a reply and print it

//clear the buffer by filling null, it might have previously received data

memset(buf,'\0', BUFLEN);

//try to receive some data, this is a blocking call

if (recvfrom(s, buf, BUFLEN, 0, (struct sockaddr \*) &si\_other, &slen) == -1)

{ die("recvfrom()"); }

puts(buf);

}

close(s);

return 0;

}

Output: $ gcc udpclient.c

$ ./a.out

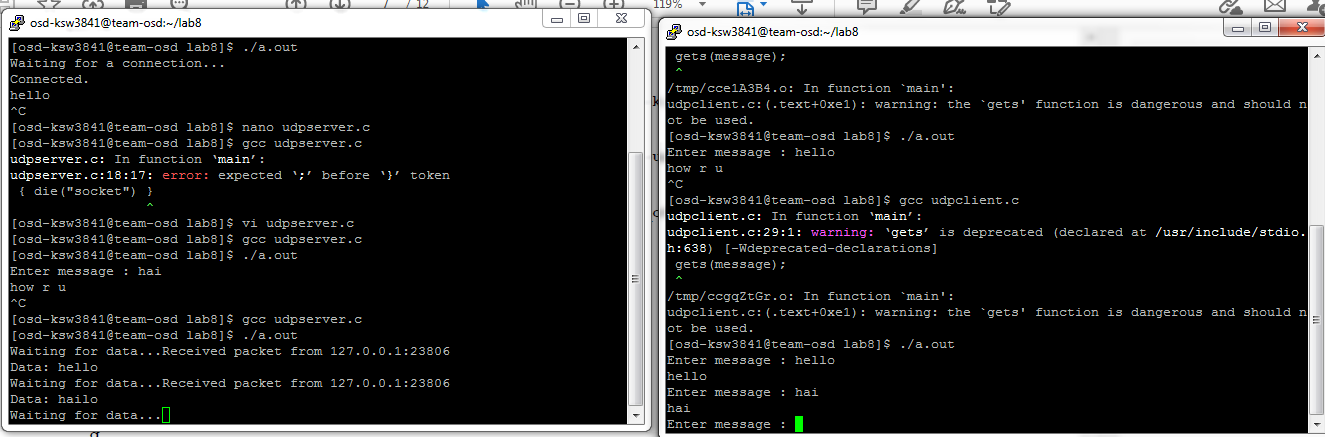
Enter message : hello

hello

Enter message : how are you

how are you

^c

****

**TCP/IP Sockets – Echo Client Server**

**/\* TCP/IP Echo Server \*/**

#include <sys/types.h>

#include <sys/socket.h>

#include <netdb.h>

#include <stdio.h>

#include<string.h>

int main()

{ char str[100];

int listen\_fd, comm\_fd;

struct sockaddr\_in servaddr;

listen\_fd = socket(AF\_INET, SOCK\_STREAM, 0);

bzero( &servaddr, sizeof(servaddr));

servaddr.sin\_family = AF\_INET;

servaddr.sin\_addr.s\_addr = htons(INADDR\_ANY);

servaddr.sin\_port = htons(22000);

bind(listen\_fd, (struct sockaddr \*) &servaddr, sizeof(servaddr));

listen(listen\_fd, 10);

comm\_fd = accept(listen\_fd, (struct sockaddr\*) NULL, NULL);

while(1)

{

bzero( str, 100);

read(comm\_fd,str,100);

printf("Echoing back - %s",str);

write(comm\_fd, str, strlen(str)+1);

}

}

Output: $ gcc tcpserver.c

$ ./a.out

Echoing back - hello

Echoing back - how are you

**/\* TCP/IP Echo Client \*/**

#include <sys/types.h>

#include <sys/socket.h>

#include <netdb.h>

#include <stdio.h>

#include<string.h>

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

{ int sockfd,n;

char sendline[100];

char recvline[100];

struct sockaddr\_in servaddr;

sockfd=socket(AF\_INET,SOCK\_STREAM,0);

bzero(&servaddr,sizeof servaddr);

servaddr.sin\_family=AF\_INET;

servaddr.sin\_port=htons(22000);

inet\_pton(AF\_INET,"127.0.0.1",&(servaddr.sin\_addr));

connect(sockfd,(struct sockaddr \*)&servaddr,sizeof(servaddr));

while(1)

{ bzero( sendline, 100);

bzero( recvline, 100);

fgets(sendline,100,stdin); /\*stdin = 0 , for standard input \*/

write(sockfd,sendline,strlen(sendline)+1);

read(sockfd,recvline,100);

printf("%s",recvline);

}

}

Output: $ gcc tcpclient.c

$ ./a.out

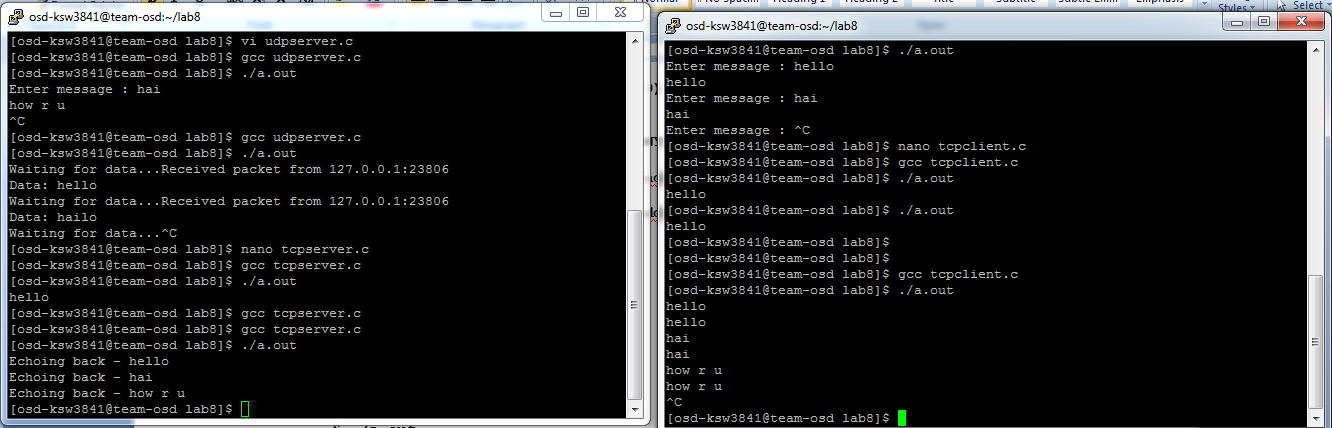
hello

hello

how are you

how are you

^c

****

**RPC:**

Demonstrate How RPC Works: invoking square remote procedure with arguments. Invoking date remote procedure without argument

**Square :**

Remote procedure calls, or RPC, provide such a tool. We code our application using the familiar procedure call, but the calling process (the client) and the process containing the procedure being called (the server) can be executing on different hosts.

The fact that the client and server are running on different hosts, and that network I/O is involved in the procedure call, is for the most part transparent. Indeed, one metric by which to measure any RPC package is how transparent it makes the underlying networking.

The client calls the server's procedure with a long integer argument, and the return value is the square of that value.

**/\* square.x \*/**

**struct square\_in { /\* input (argument) \*/**

**long arg1;**

**};**

**struct square\_out { /\* output (result) \*/**

**long res1;**

**};**

**program SQUARE\_PROG {**

**version SQUARE\_VERS {**

**square\_out SQUAREPROC(square\_in) = 1; /\* procedure number = 1 \*/**

**} = 1; /\* version number \*/**

**} = 0x31230000; /\* program number \*/**

These files whose name end in . x are called RPC specification files, and they define the server procedures along with their arguments and results.

The next program we write is the client main function that calls our remote procedure.

**/\* client.c \*/**

**#include "square.h" /\* generated by rpcgen \*/**

**#include<stdio.h>**

**#include<rpc/rpc.h>**

**#include<stdlib.h>**

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

**{**

**CLIENT \*cl;**

**square\_in in;**

**square\_out \*outp;**

**if (argc != 3){**

**printf("usage: client <hostname> <integer-value>\n");**

**exit(2); }**

**cl = clnt\_create(argv[1], SQUARE\_PROG, SQUARE\_VERS, "tcp");**

**in.arg1 = atol(argv[2]);**

**if ( (outp = squareproc\_1(&in, cl)) == NULL)**

**printf("%s", clnt\_sperror(cl, argv[1]));**

**printf("result: %ld\n", outp->res1);**

**exit(0);**

**}**

**/\* server.c \*/**

**#include<stdio.h>**

**#include<stdlib.h>**

**#include<rpc/rpc.h>**

**#include "square.h" /\* generated by rpcgen \*/**

**square\_out \***

**squareproc\_1\_svc(square\_in \*inp, struct svc\_req \*rqstp)**

**{**

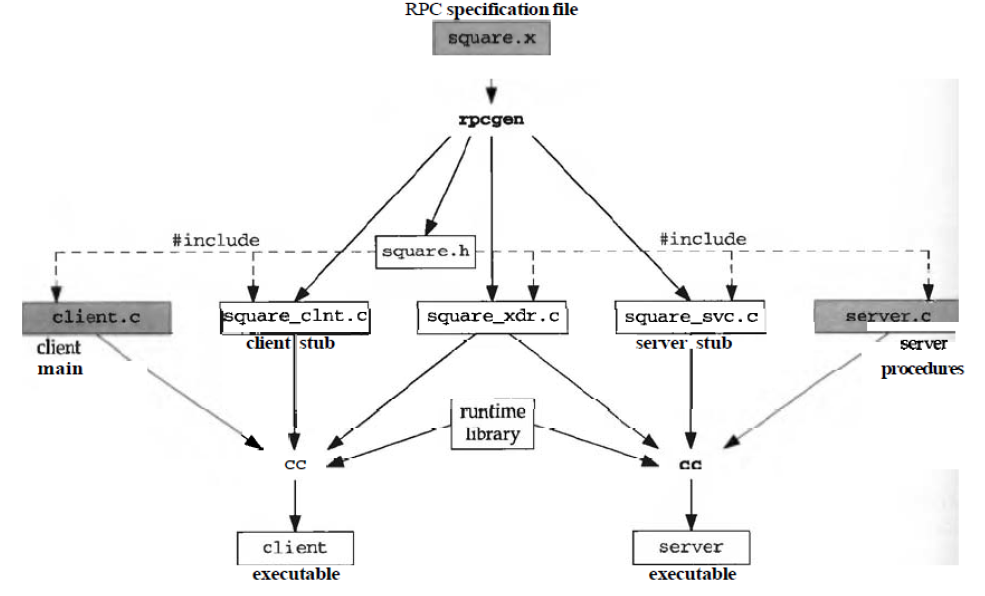
**static square\_out out;**

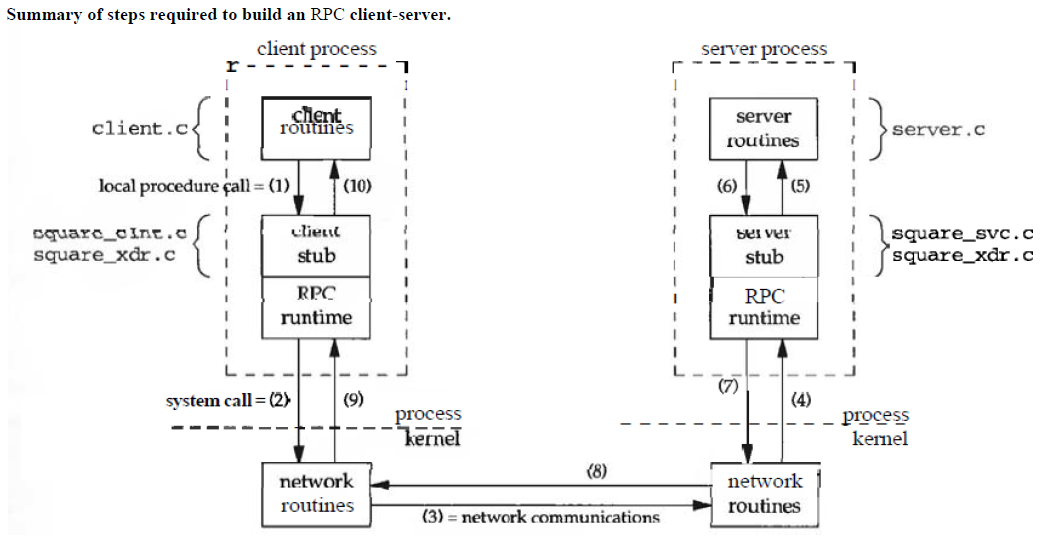
**out.res1 = inp->arg1 \* inp->arg1;**

**return(&out);**

**}**

On the server side, all we write is our server procedure, which we show in Figure. The rpcgen program automatically generates the server main function.





Commands:

1. First copy all programs in all three files square.x, client.c , server.c
2. $ ls

Displays all three programs

1. $ rpcgen square.x
2. $ ls

Client.c server.c square.x square\_clnt.c square.h square\_svc.c square\_xdr.c

1. $ gcc -c client.c -o client.o
2. $ gcc -c square\_clnt.c -o square\_clnt.o
3. $gcc -c server.c -o server.o
4. $ gcc -c square\_svc.c -o square\_svc.o
5. $ gcc -c square\_xdr.c -o square\_xdr.o
6. $ gcc -o server square\_svc.o server.o square\_xdr.o
7. $ gcc -o client square\_clnt.o client.o square\_xdr.o
8. Next open two terminals where we already executed terminal as client and new terminal as server
9. In new terminal type

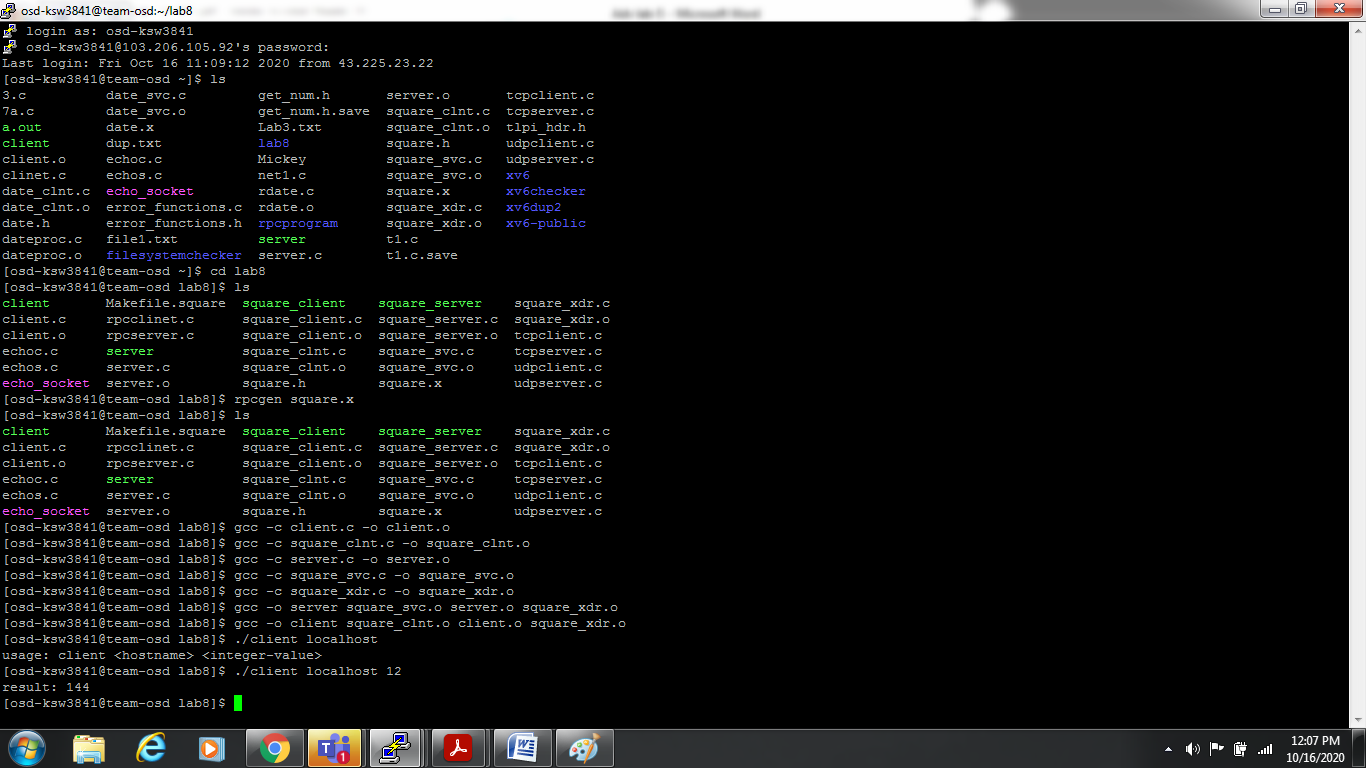
$ ./server

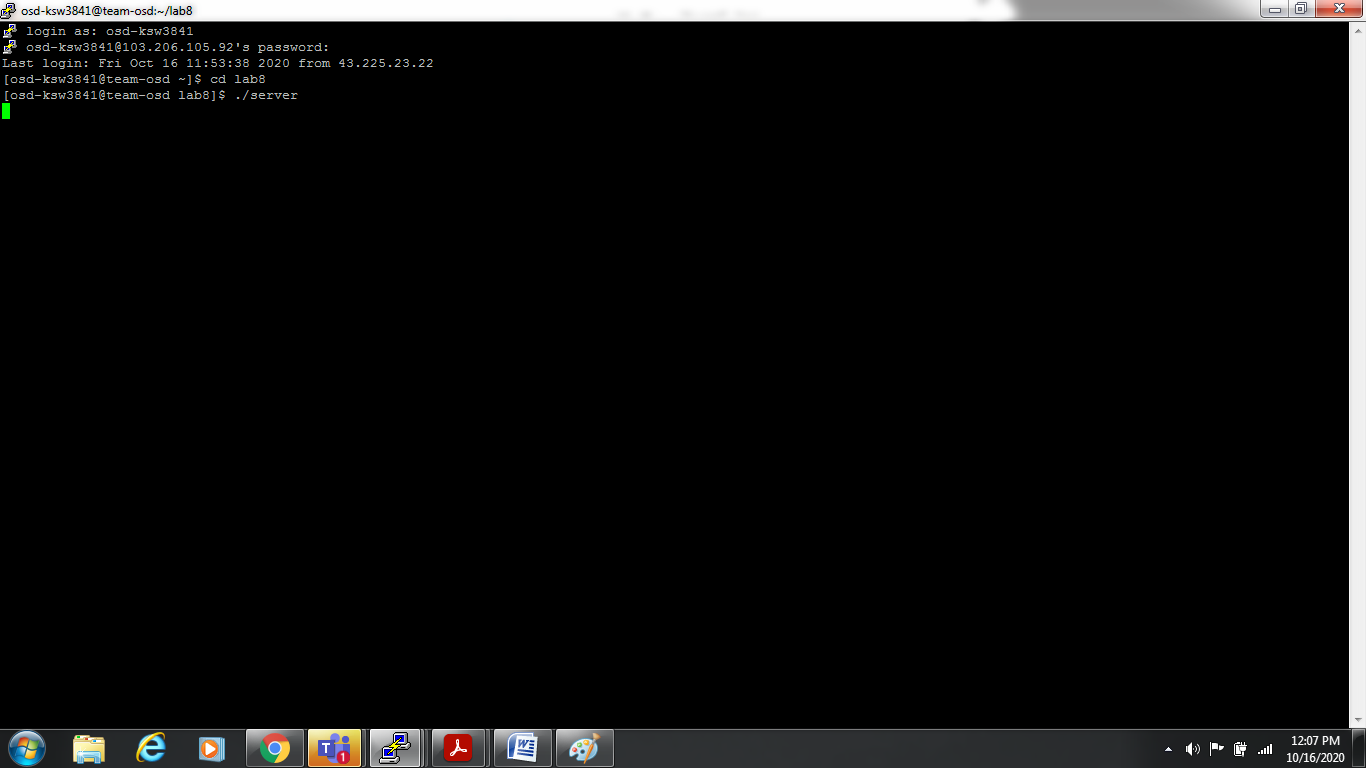
1. In old terminal type $ ./client localhost

usage: client <hostname> <integer-value>

1. $ ./client localhost 12

result: 144





Note: We will ask students to do Invoking date remote procedure without argument