Indian Institute of Information Technology, Allahabad.

Course: Computer Networks (ICNW532)

Batch: B.Tech (ECE), (Dual Degree Integrated Program) and MBA (IT)

Instructor: Dr. Vijay Kumar Chaurasiya

Lab Assignment #4

Lab Sessions on 03/11/2017

Deadline: 03/11/2017

1. **Remote Command Execution**. This is the first part of a two part assignment that uses other machines to do work for you. In this part, you are going to use the UDP protocol instead of TCP, and you are going to try out a new system call. When using UDP, you specify that the socket is of type SOCK_DGRAM, and you must use sendto and recvfrom instead of send and receive. Look at the man pages on these calls, and you can look at the programs following.

   The important thing to remember is that UDP communications are completely connectionless, so the only clue that the receiver has as to the identity of the sender is in the address portion of the receive call. If a message is to be returned, the receiver must get and use that address. In this program, instead of choosing your own port, use one of the ports established for this class. Each port has both a tcp and a udp version, so they can be used for either protocol. This is what is meant by the protocol-port addressing pair.

   So what is this program going to do. Again, we will stick with a client server model. The client side will be executed with:

   execute host command

   where host is the name of the host where the server must reside, and command is a command to be executed at the host. The remote client will send the command to the server on host and then print anything that is returned at stdout.

   The server will receive the string containing the command, which could be any legal Unix shell command and use the "system" call to perform the execution, redirecting the command output to go back to the client.

   For example, if execute somehost.thisdomain.edu ls

   is entered, the server would execute the ls command, and send the result back to the client, where it would be output to stdout. There are a number of issues here.

How does the "system" call work? Check the man pages, its easy. If the server were running on the machine as a root process, how would it know what directory to "ls"?

The solution is to make the server handle a login for the user on the other machine. This also provides security so that all sorts of riffraff don't use your machine for executing things. We will ignore this problem, How does the child process redirect the command output to the client?

Well, this does present problems. Basically, you need to change the default assignment for stdout for the command, but this could go badly.

If you attempt to store all of the output in memory, or even a file, you could run into resource availability problems. What is preferable is to direct the output to the network so that it is sent directly to the client process. While this isn't directly possible, you can play some games to make it work.

First, you need to create a pipe with a pipe call.

```
int pd [2];
if (pipe (pd) < 0)
{
fprintf (stderr, "Error opening pipe\n");
exit (-1);
}
```

A pipe is a buffer with a read descriptor (pd[0]) and a write descriptor (pd[1]). After the pipe is opened, you can read and write the pipe, and the processes doing so are synchronized so that if the pipe is full, the reader is blocked on a read and if the pipe is empty, the writer is blocked on a write. The two descriptors cannot be switched.

```
read (pd[0], buf, n); write (pd[1], buf, n);
```

Pipes can be thought of a being like a socket, except that if two processes are using a pipe, they must be related in the parent-child sense.

The dup call can be used to create a duplicate of any file descriptor, and it will get the lowest numbered file number available.

If the process being created has any output to stdout, you can redirect it to the pipe by doing the following.

1. create a pipe (pipe (pd))
2. close file descriptor 1, which is stdout (close (1))
3. dup pd [1], the output side (newfd = dup (pd[1]))

Because stdout was closed and stdin is still open, the newfd will be 1, and it will be assigned to the same i/o structure as pd[1], which is the write side of the pipe. Any attempt to output to stdout, will actually write to the pipe. If the server process reads the pipe, it reads what would normally go to stdout and it can send it back to the receiver.

When the process is started to run with the system call, anything it writes will go to the pipe. The server child can read the pipe and forward the data to the client for output. There is one problem, the pipe has a limited size, so if the system command generates a lot of output, it will block on the pipe, and the server child is blocked waiting for the system call to end. This can all be avoided by using asynchronous I/O which is the Unix version of interrupt driven I/O.

Unix will let you capture signals, which are interrupts that have been handled by Unix, but then passed to a process. The best description is probably an example, which follows.

```
/*
===============================================================
* Example of using a pipe to handle stdout. Under OSF/1, this must
* be compiled in the following way:
*
* cc prog.c -lsys5 -o prog
*
* with the System V library loaded. You need the System V library
* as it defines the semantics for the signal handling. For some reason,
* the BSD library doesn't work right.
*
*
===============================================================
*/
#include <stdio.h>
#include <signal.h>
#include <fcntl.h>
#include <sys/types.h>
#include <unistd.h>

void piperead ();

int dd, pd[2];
void (*oldsigio) ();
void main (int argc, char *argv[])
{
int flags, ct;
char buf [80], ch;
/*
```

```
* Open a pipe, then close stdout and using dup, set up pd[1] to
* be the same file descriptor.
*/
if (pipe (pd) < 0)
{
fprintf (stderr, "Couldn't open pipe");
exit (0);
}
close (1);
dd = dup (pd[1]);
/*
* Make I/O on pd[1] asynchronous, which means this program can get
* the signals from the OS.
*/
flags = fcntl (pd[0], F_GETFL, 0);
fcntl (pd[0], F_SETFL, flags | FASYNC | O_NDELAY);
/*
* Tell the system to catch the SIGIO signal. Technically, we could get
* in trouble, since it will catch all asynchronous I/O, but that shouldn't
* be a problem here.
*/
oldsigio = signal (SIGIO, piperead, -1);
/*
* Do the system command, which if it writes to stdout (file 1), it
* will come back to the process via the signal handler. Note that the
* "system" call will not return until the created child process completes
* and terminates.
*/
system ("ls");
/*
* Reset signal to be safe and clean up.
*/
signal (SIGIO, oldsigio, -1);
close (pd[0]);
close (pd [1]);
close (dd);
}
/*

* Asynchronous signal handler. Reads anything waiting at pd[1] and
* prints it out. It only executes when the system knows that something
* is waiting.
*/

void piperead ()
{
```

```
char buf [80];
/*
* Read the pipe and output to the terminal. Stderr has to be used
* because stdout has been closed. Note, this may output a strange
* line at the end because it doesn't check for a line containing
* a new line alone.
*/
if (read (pd[0], buf, 80) > 3)
fprintf (stderr, "%s\n", buf);
return;
}
```

Asynchronous I/O has two parts. The first part is specifying that a specific I/O descriptor is to be handled asynchronously by using the fcntl call. You are telling the operating system that you want to take more control of the I/O for this descriptor by not waiting for the OS to decide when it is time for you to do something with the data. Instead, you want to be informed when input is ready or output is done, so that you can act accordingly. In this case, that means that you want to know when there is input ready in the pipe, so that it doesn't fill up and block before the "system" call returns.

Next, the system has to be told what to do when I/O is ready, and that is done by indicating that you have a handler for the signal called SIGIO.

Unix does not have a special signal for each I/O device. Instead, all I/O interrupts for your program that are set up to be asynchronous cause the same signal, and so they all have the same signal handler. There are calls to allow your handler to decide what has happened. In this case, there is only one asynchronous file, so that isn't a problem. Other signals are for pressing the control-C key (SIGINT), bus errors (SIGBUS), floating point errors (SIGFPE) and so on (up to 64 signals on modern Unix systems). This is, in large part, the code for the child process, except that you need to get the data coming in from the pipe sent off to the client.

This assignment has quite a few new things in it for most people - pipes, signal handling, datagrams and asynchronous I/O. But the applications are simple. Implement things as shown and read the man pages. In future labs, you will do some things that are more complicated.

```
/* ====================================================>
dg_client.c
* Simple Unix domain datagram type peer process. It comes up and
* sends messages to another process.
*
================================================================
*/
#include <stdio.h>
#include <sys/types.h>
```

```
#include <fcntl.h>
#include <sys/socket.h>
#include <sys/un.h>
void main(argc, argv)
int argc;
char argv[];
{
int flag, sock, addrsize, saddrsize, response;
struct sockaddr_un addr, saddr;
int getmsg(), done, len, rlen;
char socketname[20], buf[80];
sock = socket(AF_UNIX, SOCK_DGRAM,0); /* create a socket for input */
if (sock == -1)
{ perror("opening socket");
exit(-1);
}
addr.sun_family = AF_UNIX;
strcpy(addr.sun_path, "dgclient"); /* create and bind a name */
addrsize = strlen ("dgclient") + 2;
if (bind(sock, &addr, addrsize) == -1)
{ perror("on bind");
exit(-1);
}
/* Send messages to the server to be processed and returned.
* Note that the null message is sent to kill the server.
*/
strcpy(addr.sun_path, "dgserver");
addrsize = 10;
done = 0;
do
{
printf ("Enter a short message to be sent, return to halt\n");
gets (buf);
len = strlen (buf);
if (len < 2)
done = 1;
if(sendto(sock, buf, len, 0, &addr, addrsize) < 0)
perror("on client write");
saddrsize = 32;
rlen = recvfrom (sock, buf, len, 0, &saddr, &saddrsize);
printf ("client: %s <s returned from %s>\n", buf, saddr.sun_path);
} while (! done);
/* Close the socket and unlink the socket name
* which means delete the file representing the socket
*/
shutdown(sock, 2); /* not receiving or sending anymore */
close(sock);
unlink("dgclient");
}
```