CS 350 Operating Systems

Study Questions

1 Interprocess Communication

1. When communicating using UNIX stream sockets, processes use the system calls socket(), accept(), bind(), listen(), and connect(). What is the purpose of each of these calls?

2. When a connection is set up between a pair of stream sockets, one process is considered to be "passive", and the other "active". What is the difference between the passive and active ends of the connection? Of the system calls listed in the previous question, which are used at the passive end, and which are used at the active end?

3. What are the differences between datagram sockets and stream sockets?

4. In UNIX, suppose that a parent process and its child wish to establish two-way communications. Show how this can be accomplished using pipes.

5. Suppose that a single UNIX process wishes to create two stream sockets and then connect them together. Is this possible using the socket system calls we have discussed in class? If so, give the sequence of system calls that will do so, along with a short sentence describing the purpose of each call. If not, briefly (a short sentence or two) state why not.

6. Suppose that two UNIX processes need to share a device. A third process, called the mediator, is used to make sure that only one process uses the device at a time.

   Assume that the two device-using UNIX processes run the program shown below. Each process first establishes a connection with the mediator using a stream socket. To request permission to access the device, it sends a message to the mediator. The reply from the mediator is its signal that permission has been granted. After using the device, the process sends another message to the mediator to indicate that it is done.

```
int d; /* to hold socket descriptor */
char c; /* to hold incoming message from mediator process */
d = socket(STREAM); /* create a stream socket */
connect(d,"mediator"); /* establish connection to the mediator process */
DO FOREVER
    write(d,'r',1); /* request access to the device by sending 'r' on the socket -/
    read(d,&c,1); /* wait for a reply indicating device access has been granted */
    /******************************************/
    /* use the device for a short (finite) time */
    /******************************************/
    write(d,'g',1); /* send message to indicate that this process is finished with the device */
    /*******************/
    /* wait a little while */
    /*******************/
END DO
```

   Write pseudo-code for a mediator process that guarantees that only one of the two processes will use the device at one time. The mediator should not allow either process to use the device until it has established connections with both. Your mediator should guarantee that neither process will wait forever for the device. It should also guarantee that a process requesting the device is granted permission immediately if the other process is not using it, i.e., there should be no unnecessary waiting.

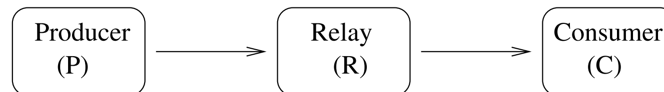
   Your code should use UNIX system calls for communication, though it is not important that you specify their parameters exactly as you would on a UNIX system. You may also use the `select` system call. Select takes two

descriptors as parameters. It causes the process to block until there is data available to read on at least one of the descriptors. Its return value is the descriptor on which data is available for reading. If data is available on both descriptors, a special integer constant "BOTH" is returned. For example, a call to select(3,4) will block until data is available to read using descriptor 3 or descriptor 4. The return value of the call will be either 3 or 4 or "BOTH".

7. Pipes and sockets are two kinds of communications abstractions available in UNIX. An alternative abstraction that might be considered is the "named pipe". To use one, you would assign a pathname to a pipe when it was created. (Think of the named pipe as another kind of file, in addition to regular files, directories, symbolic links, etc.). Then, you could obtain a read or write descriptor for the pipe by simply opening the named pipe. Once two processes had the read and write descriptors, the behaviour would be exactly as for a conventional pipe. Discuss the advantages and disadvantages of a named pipe relative both to a conventional pipe and to the socket mechanism.

8. In UNIX, suppose that a parent process wishes to create 3 child processes, and allow any child to send a data to any other child using a pipe. How many pipes must be created? Which process(es) should create the pipes: the parent process, or the children, or both?

9. Consider the three-process distributed system shown below.



Process P is a data producer, and process C is a data consumer. Data produced by P is sent to C indirectly, through a *relay* process R. The processes communicate using UNIX stream sockets. Pseudo-code for processes P and C is given below:

Process P	Process C

int s,amt;
char buffer[N_p];
/* create a stream socket */
s = socket(STREAM);
/* establish a connection to R */
connect(s,address-of-R);
/* indicate this is the producer */
write(s,"p",1);
do forever {
    /* produce amt $< N_p$ bytes of data and put it in buffer */
    amt = produce-some-data(buffer,N_p);
    /* send the data to R */
    write(s,buffer,amt);
}
	int s,amt;
	char buffer[N_c];
	/* create a stream socket */
	s = socket(STREAM);
	/* establish a connection to R */
	connect(s,address-of-R);
	/* indicate this is the consumer */
	write(s,"c",1);
	do forever {
	    /* read at most N_c bytes of data into buffer. */
	    /* amt is actual number of bytes read */
	    amt = read(s,buffer,N_c);
	    /*********************************
	    do something with the data
	    *********************************
	}

Write pseudo-code for process R. Process R should establish communications with both P and C before relaying any data. In an infinite loop, it should then read data from P, and write that data to C. Each read should read at most N bytes of data, where N is some positive constant.

Your pseudo-code should be similar in style to the pseudo-code shown above. It is important that you show all communication-related system calls in their proper order. However, it is not important that the arguments to the system calls appear exactly as they would in a real program.

10. A server process wishes to communicate with an unknown and time-varying number of unrelated client processes. Clients send requests to the server, and the server sends responses to the clients. At a particular time, there are N clients communicating with the server. Suppose that the clients communicate with the server using

stream (connection-oriented) sockets. How many sockets will be needed by each client process to implement this communication? How many sockets will be in use by the server process?

11. Consider the same scenario described in the previous question, except that communication is accomplished using datagram (connectionless) sockets. How many sockets will be required by each client process? How many sockets will be required by the server process?

12. Name at least three distinct operating system facilities that can be used for interprocess communication. Sockets are an example of a facility (but don't use that example).

13. A server process is communicating with set of client processes. There may be many clients. The server handles client requests one at a time. The pseudo-code below illustrates the behaviour of the clients and the server, assuming that datagram sockets are used for communication.

CLIENTS' CODE

```
s = socket(DATAGRAM); // create a socket
sendto(s,message,messageSize,SERVERADDRESS); // send a request to the server
recvfrom(s,responseBuffer,bufferSize,addressBuffer); // receive a response from the server
close(s); // close the socket
```

SERVER'S CODE

```
s = socket(DATAGRAM); // create a socket
bind(s,SERVERADDRESS); // bind an address to the socket so clients know where to send
while true do { // handle an arbitrary number of requests, one at a time
    recvfrom(s,requestBuffer,bufferSize,clientAddressBuffer); // receive a request
    sendto(s,response,responseSize,clientAddressBuffer); // respond to the client that sent the request
}
```

Rewrite the client and server pseudo-code so that stream sockets, rather than datagram sockets, are used for communication. Each request/response message pair should be carried over a stream established between the server and the requesting client. Once the request/response pair is complete, the stream connection should be broken, and the server should establish a connection with another client.