CS 350 Operating Systems

Study Question Solutions

1 Interprocess Communication

1.
   - socket() creates a communications end point
   - bind() assigns an address to an end point
   - listen() informs the kernel that the end point will be used to accept new connections, and causes space to be allocated (in the kernel) for a queue of pending connections
   - accept() is used to accept a new connection
   - connect() is used to request a new connection

2. The active process initiates the connection by naming a target address in a connect() call. Only the socket() and connect() calls are required in the active process. The passive process assigns an address to its socket and then waits for connection requests. It uses socket(), bind(), listen(), and accept().

3. Datagram sockets are not connection-oriented, transmit messages of a fixed maximum length, are support only unreliable, unordered delivery. Stream sockets are connection-oriented and do not provide message boundaries. They support reliable, ordered transmission.

4. Use two pipes:

```
main() {
int pipea[2],pipeb[2],pid;
pipe(pipea);
pipe(pipeb);
pid = fork()
if (pid == 0) {
    /* child process */
    /* close one end of each pipe */
    close(pipea[0]);
    close(pipeb[1]);
    /* remainder of child code here */
}
else {
    /* parent process */
    close(pipea[1]);
    close(pipeb[0]);
    /* remainder of parent code here */
}}
```

5. To connect the two sockets, accept() and connect() must be called. Since both are blocking calls, one process will not be able to call them both.

6.
```
main {
int d, ns[2], r, choose;
char c;
    d = socket(STREAM);
    bind(d,"mediator");
    listen(d);
    /* accept connections from both processes */
```

```
    ns[0] = accept(d);
    ns[1] = accept(d);
    /* "choose" is used to make the mediator fair */
    choose = ns[0];
    DO FOREVER BEGIN
        r = select(ns[0],ns[1]);
        IF (r = BOTH) THEN
            r = choose;
            IF (choose = ns[0]) THEN choose = ns[1];
            ELSE choose = ns[0];
        read(r,&c,1);
        write(r,'x',1);
        read(r,&c,1);
    END
}
```

7. relative to pipes:

   - +: no need to have pipe created by common ancestor process
   - -: access to data limited only by file system permissions (less control)
   - -: need to worry about names: duplication, cleaning up old ones, etc.

   relative to sockets:

   - +: simpler to establish connection
   - -: limited to a single machine
   - -: only simplex (one-way) communication is supported

8. If pipes are simplex, use two pipes per pair of processes, one for communication in each direction. Since there are three distinct pairs of processes, six pipes will be needed. They must be created by the parent.

9. ```
   int amt;
   int ps,cs,s,ns; //socket desriptors
   char c,buf[N]; //N is a positive constant
   s = socket(STREAM);
   bind(s,address-of-R);
   listen(s,2);
   // accept a connection
   ns = accept(s);
   // read one character from the connection
   read(ns,&c,1);
   // determine whether our connection is to P or C
   if (c == 'p') then ps=ns else cs=ns;
   ns = accept(s);
   read(ns,&c,1);
   if (c == 'p') then ps=ns else cs=ns;
   do forever {
       //read at most N bytes from the producer
       amt = read(ps,buf,N);
       //write what we read to the consumer
       write(cs,buf,amt);
   }
   ```

10. The server needs to have one socket for each client, i.e., N sockets. In addition, it needs a socket for accepting new connections. Each client needs a single socket to communicate with the server.
   ```

11. In this case the server needs just one socket - messages from all clients arrive at the one socket. (The server can distinguish messages from different clients using the message return addresses.) As before, each client needs only a single socket.

12. Examples include: shared virtual memory, pipes, message queues, shared files.

13.

Client	Server
	s = socket(STREAM);
	bind(s,SERVERADDRESS);
s = socket(STREAM);	listen(s,queueLen);
connect(s,SERVERADDRESS);	**while** true **do** {
send(s,message,messageSize);	s2 = accept(s,clientAddressBuffer);
recv(s,responseBuffer,bufferSize);	recv(s2,requestBuffer,bufferSize);
close(s);	send(s2,response,responseSize);
	close(s2);
	}