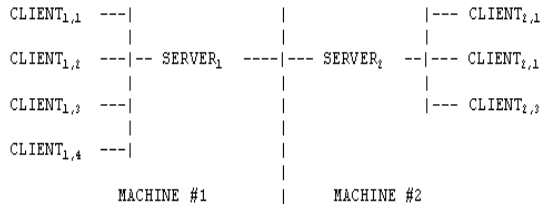
CSC 510 Project 2

Points: 25
Due: 11:59 PM on the designated day. No late submissions accepted
Purpose: Implement Concurrent Processes on multiple machines; use sockets and message queues; catch a signal

Description: For this project you will develop a package that can support a distributed conference being held at two (for now) sites. UNIX IPC in this phase will be implemented using sockets and message queues(msq). Shared memory and semaphores will be added in Project 3.

Your system's servers will support multiple clients at each site and be able to communicate with similar servers at other sites. The design of the entire project (including the clients) is as follows:



$SERVER_1$ and $SERVER_2$ are virtually the same, except there must be one server that is designated as the first server, so other servers can use it as a frame of reference. $SERVER_2$ style servers will confirm that the system is running by communicating with the single $SERVER_1$ process, whose location is *well-known*. Therefore, $SERVER_1$ should be launched before $SERVER_2$ (if not, SERVER_1 will not be found when SERVER_2 tries to connect to it). Once this initial set up is done, both servers will perform the same function.

The $CLIENT_{i,j}$ programs are all the same. Upon initiation, the CLIENT programs first get the id of the message queue that the local server reads from. The call to msgget() to get this id will always use getuid() as the key. If there is no such message queue (return value of msgget() helps determine this), the server is not available (not up yet, not up at all, system error, etc;) and the client should gracefully terminate.

Once the client has hooked itself up to the server's msq, it can send data to the server. Before the first transmission to the server (registration), the client creates its private message queue. It then furnishes this msqid to the server in its registration message.

The server, upon receiving a registration message from a client, adds the client info to its list of clients. Restrict this list to 5 clients per server for now. If a sixth client tries to start up, it should receive a "go away" message from the server (through its private message queue) and terminate gracefully.

Each client is one participant in the conference. At the prompt, a client may type in a message. When the client hits return (to signal end-of-line), a message packet is sent to the local server via the server's message queue, and the server is informed the message is there (see below). This packet will contain the sender's pid (maybe as the mtype field), name, the time of the message and the message text.

By default, each message should be broadcast to every other client (local or remote) in the system. In order to achieve this, the local server sends the message from the local client to the remote server (with which it has a socket connection). The remote server then makes the message available to all the clients on its system via the private message queues which connect the server to each of the clients. The local server must also put the message on the local private queues of each client except the sender, so that participants at the local site can also view the message.

In order to be able to select between processing an outgoing message and displaying an incoming message, each client must fork off a child that will wait for messages to arrive on the client's private msq. When a message arrives, the child client will send it to the parent through a pipe (created, of course, prior to the fork) for display. Meanwhile, the parent process waits for the user to type something. There is to be no conflict in display between what the user is typing and the display of an incoming message (that would be printed on the screen).

When a server receives a message from any client, it will also log the client's name, pid, the total number of messages contributed by that client, total local messages and total messages.

Executing the Programs

The $SERVER_1$ program must run on acad, because it has ports that are open to allow transmission through the KU firewall.

A possible launch sequence is in the box. Names on clients are optional, but recommended.

```
acad> server1          <--- start well-known server on acad
yin> server2           <--- start 2nd server on another machine
acad> client acad1     <--- start a client with a given name.
yin> client yin1       <--- start a client on yin
yin> client yin2       <--- start another client on yin
yin> client yin3       <--- start yet another client on yin
acad> client acad2     <--- start another client on acad
etc;
```

The clients need not be launched in the sequence shown. The only requirement is that the servers be launched before their clients and that server1 on acad be launched before any server2 (if they are to start up correctly). The servers also have to set up the message queue before the clients get access to them. Of course, you will detect incorrect ordering and your programs will report why they can't run (if they can't) before terminating.

There will be one socket connection for each server$_2$ and two types of message queues. There will be one universal message queue on each local machine through which clients can send messages to the server. For server-to-client communication, there should be one private message queue for each client.

The system will shut down when all clients of a server have exited (i.e. the number of clients makes a transition from 1 to 0). When an individual client exits, it first informs its server, then removes its private message queue after the server acknowledges the client's shutdown. If a server has no more clients it informs the other server that it will shut down.

Servers remove their incoming msq, close the sockets (use shutdown() or close()), close file(s), etc;

Pseudo code

The following is partial pseudo code for servers and the clients.

SERVER$_1$

1. Setup socket connection
2. Wait at accept() from any other server on a remote site
3. Setup a message queue to receive messages (and commands) from local clients.
   /*** the above completes the initial setup routine ***/
4. Wait for a message. Messages can come from the message queue (from a local client) or the socket connection (from a client on at a remote location). The server waits on data over the socket from a remote server, and if a local client sends a message, it will also signal the server, causing the socket read to be interrupted (you must restart this). The server will then take the message off its incoming message queue in a signal handler.
5. If a message is received from the socket, then send it to all the clients. This can be achieved by placing the data on each of the private message queues. The data received from the network must be converted to the host format before it is processed.
6. If a message is received on the msq, make it available to other clients via their private message queues. Then convert the message to the network format and send it over the network to the remote server.
7. Go back to polling both the message queue and the socket connection. (GO TO 4)

SERVER$_2$: SERVER$_2$ is analogous to SERVER$_1$ except that it connect()s to SERVER$_1$ instead of 1 and 2 above.

CLIENT:

1. Get access to the local server's incoming message queue.
2. Setup a private message queue for itself.
3. Register itself with the local server. Registration information includes the name of the client, its private message queue ID and process ID. A private message queue is setup using a key of IPC_PRIVATE in msgget(). Note that the name of a client is not going to be very significant in this project, but will be important in Project 3.
4. Set up a pipe for IPC between child and parent process that will be forked off in the next step.
5. Fork off a child process. The child process starts waiting for messages from the local server
6. The parent process starts waiting for user input and input from the pipe. You can use the select() function to poll for I/O.

The parent process must take care of formatting the output on the terminal. Refer to your Unix reference for information and example codes on controlling terminal I/O. Some of the functions that may be of help are :

ctermid(), tcgetattr(), tcsetattr(), setbuf() and fileno()

NOTES

- Each server will maintain one LOG file. The pid of all senders, both local and remote, will be logged.
- You will be assigned a port that has clearance through acad's firewall. Use this port for Server$_1$'s socket. Of course, that process runs on acad.
- Any numbers sent over the internet should be converted using a *hton()* or *ntoh()* function to assure proper byte ordering.
- A message should contain the sender's pid, machine name, possibly the sender's given name, message number (for that client) and the text of the message.
- When displaying a message received from its private msq, a client should display the name of the sending client and the machine on which the client resides.
- A client name may not be duplicated on a single machine, but different machines may have clients with matching names. You may omit the names from this project, but they will be an integral part of the next phase. It is strongly recommended that you implement them in this phase.
- Submit a README file that coherently describes your implementation and includes any design decisions that you feel anyone who would read or use your program should be aware of.
- It is best if your code is fully portable, as Project 3 may be implemented over another machine.

Turnin

All code files plus a makefile. You may name your c or cpp files as you like, but the executables must be named server1, server2, and client.