CSC 510 Project 4

Points:	25
Due:	11:59 PM on April 25, 2012. No late submissions accepted
Purpose:	Implement a Distributed Point-to-Point Talk System; use both kinds of sockets; semaphores, shared memory, signal processing

Description: This project will build upon the last project and develop a distributed point-to-point talk server. This project will be closer to what you might expect in a real-world multi-party talk server. While implementing this project, you will work with UDP (Datagram) & TCP (Stream) sockets, shared memory and semaphores, and signals.

In this project, there will be two programs to write. There will be a lookup server for the system and local clients on each machine. Following are descriptions of these programs:

A. LOOKUP_SERVER

The lookup server must be "well known" to all the clients. This server, called the LOOKUP_SERVER, runs on acad. Once the LOOKUP_SERVER is launched it starts waiting for connections from any client. The clients may be running on any machine on the Internet. Clients requesting service are connected to the LOOKUP_SERVER using a stream socket that uses your assigned port number.

The LOOKUP_SERVER's purpose is to provide directory lookup service to clients wishing to send messages to other clients in the system. In order to assemble the directory, clients register themselves with this LOOKUP_SERVER as their first task. If the name of the client is already in use by another client on any machine in the system, the client is to be informed and that client will exit the system. During the registration process, the client sends the lookup server its logical name and a sockaddr_in structure that can be used by other clients to contact it. The child LOOKUP_SERVER corresponding to this client will ascertain that the total number of clients in the system, including this new one, does not exceed the limit. If the system is at capacity, the LOOKUP_SERVER will send back an error message to the client. If the id is unique and the number of clients is below the limit then the login is granted and the client's info is put into the shared memory at the LOOKUP_SERVER site.

The LOOKUP_SERVER should first setup a shared memory segment that will maintain information concerning a maximum of 10 clients. Each time the LOOKUP_SERVER receives a connection request from a client, it should service it using the usual TCP/IP method, forking off a child LOOKUP_SERVER which, from then on, handles all requests made by the client. Thus, each client will have one dedicated child LOOKUP_SERVER. The LOOKUP_SERVER should also keep a count of the number of child LOOKUP_SERVERs in the shared memory area. This count is incremented each time a new LOOKUP_SERVER is forked off and decremented each time a child LOOKUP_SERVER exits the system. The increment can be done by only the parent LOOKUP_SERVER but the decrement operation will be done by the exiting child LOOKUP_SERVER.

When a child LOOKUP_SERVER exits, the parent LOOKUP_SERVER should react to the system generated signal and wait() for its exiting child.

When a client wishes to message another client, it will first have to contact its dedicated child LOOKUP_SERVER and retrieve the sockaddr_in structure of the destination client. This retrieval is accomplished by the child LOOKUP_SERVER searching the information in the shared memory database. The sockaddr_in structure returned is used in sendto() (UDP send) to route the message to the correct client.

The structure of the shared memory to be maintained at the LOOKUP_SERVER site is shown in the box. Note that you may want to convert the C-style structs in this description to C++ style (including the all cap names). The entire segment is held in one DIR struct.

```
#define MAX_CLIENTS             10

typedef struct tagCLIENT   // Note the C-style dec.
{
      char Name[20] ;
      struct sockaddr_in serverAddr ;
      struct timeval startTime ;
      // Time of most recent lookup
      struct timeval lastLookupTime ;
} CLIENT_INFO ;

// this structure is to be kept in the shared memory
typedef struct tagDIR  {
      CLIENT_INFO clientInfo[MAX_CLIENTS] ;
      int numClients ; // Total Clients in System
} DIR ;
```

Since DIR is a shared memory segment to which multiple child LOOKUP_SERVERs and the parent LOOKUP_SERVER have access, it represents a critical section, and you must synchronize access to it by using semaphores. More on this later.

B. Client

The client programs are launched from the command line in the following manner:

`Unix prompt>` client <name>

Client info is maintained within a shared memory segment on the local machine. Shared memory for clients will be maintained by the clients themselves on every machine on which there is a running client. This segment will be used to keep track of information about each client running on the local machine. The layout of the shared memory for local clients is in the box.

```
typedef struct tagLOCAL_INFO
{
      char name[20] ;
      struct timeval startTime;
      struct timeval lastMsgTime;
      int numMsg;
      pid_t pid ;
} LOCAL_INFO ;

typedef struct tagLOCAL_DIR
{
      LOCAL_INFO  localInfo[MAX_CLIENTS];
      int         numClients;
      int         totalMsgs;
} LOCAL_DIR ;
```

The first thing a client does on startup is to try to register with the LOOKUP_SERVER, in order to be allocated space in the LOOKUP_SERVER's lookup table. If the LOOKUP_SERVER is not found, or has no more space for new clients, the client should terminate gracefully. A client that successfully registers with the lookup server but finds no local IPC (IPC for the LOOKUP_SERVER doesn't count), will allocate the shared memory and semaphore sets for local clients. It also initializes the numClients and totalMsgs fields once it sets up the shared memory. The semaphores to guard shared memory accesses must not allow other clients to proceed until this client is finished setting everything up. If a client finds the shared memory infrastructure in place, it does not allocate any IPC.

Each client will initialize the next free space in the LOCAL_DIR.LOCAL_INFO field with its Name, startTime, and pid. The numMsg and lastMsgTime fields are initialized to 0. Whenever a client sends or receives a message, it will update its numMsg and lastMsgTime fields. The client also needs to bind a port to a datagram socket (port values will be determined in class). You need to use the select() command to wait for either user input or messages sent from other clients.

The client now updates the local shared memory with details about itself. The client will get access to and increment the local client counter.

A message entered from the keyboard should be of the form :

<dest user> <message>

E.g.

Eloise Love that CS prof!

where Eloise is the name of the user to whom "Love that CS prof!" is sent. Once this message is typed, the client should have the LOOKUP_SERVER do a lookup on <user>. The LOOKUP_SERVER will return with either an error value or the sockaddr_in structure with which the destination <user> can be contacted (an error can also be a sockaddr_in structure with a bad port number; e.g. 0). If the user is not found, the client should print a message along the lines of "Eloise not logged on". Otherwise, the client uses the sendto() function to send the message packet to the the destination client. The message packet should include the client's name.

The command line has three reserved words that are commands (and aren't to be used as usernames, i.e. if a client is started with these commands as the username (command line argument) it should terminate with an appropriate message IMMEDIATELY). The commands are LIST, ALL, and EXIT. LIST causes the data on all local clients to be output, ALL lists all clients in the system, and EXIT causes the client to leave the system.

When a client sends a message, no acknowledgement of its receipt is required. When a client receives a message, it should display it with the format

Message from <user>:
<Message Text>
The client will then update the totalMsgs field of the shared memory.

B. Local Shared Memory

As in the previous shared memory case,
Be aware that the number of semaphores available may be limited by the system. If access to semaphores becomes a problem, we will have to limit the number of semaphores in a set. Thus, you should be prepared to limit the number of semaphores used (you can detail your super-duper solution in your README file). You will be expected to give a description of each semaphore you use in your README file (which is required). Use getuid() as the key for allocating your semaphore set.

Semaphores

The readers-writers algorithm should be implemented for all critical sections that are accessed read only at some times and read/write at others. At no time should it be POSSIBLE for data to be corrupted by any normal and reasonably expected event.

You will again need to use one or more semaphores to guard access to various parts of shared memory. You may use multiple semaphores to increase parallelism of access, but be sure not to violate mutual exclusion principles. You should not allocate more than one semaphore set on any machine except the LOOKUP_SERVER machine, which could have two semaphore sets if there is also at least one local client present.

Signal Handling and Shutdown

You will need to use signals in the project. In the LOOKUP_SERVER program, the child LOOKUP_SERVERs should send a signal to the parent LOOKUP_SERVER when it exits. The child LOOKUP_SERVER processes should call the exit() function when they are done, automatically generating a SIGCHILD signal to the parent process. The parent process will catch this signal and issue a wait() to (optionally) collect this child's exit status. When a child LOOKUP_SERVER exits, the parent LOOKUP_SERVER will consult the # of clients in shmem and if it is 0, the parent will send a SIGINT signal to itself (before re-entering the accept() routine). Upon receipt of SIGINT, (this can happen from the keyboard, too) the LOOKUP_SERVER parent will enter a signal handler that asks the user whether to shut down or wait for more clients. Action will be taken according to the user's reply.

Thus, the server must be brought to the foreground (if not already there) when the number of clients becomes 0.

Client programs need to check if they are the last client on a machine when they exit the system. They first decrement the NumClients field of the local shared memory segment. If this value becomes zero, the client deallocates the IPC that the first client created.

Assume that any LOOKUP_SERVER that catches a SIGINT while it has clients associated with it will ignore that signal.

Assume that no other signals are used in this project.

You will need to write signal handlers for each of the signals you wish to catch. Make sure that you call the signal() function to load your signal handler. Use the kill() function to generate the signals. Example: sigdemo.c in the public/demo directory.

E. Notes

- Clients can’t keep sockaddr_in structs locally.
- The name of a client may not contain whitespace.
- A client is rejected if it has the same name (case insensitive) as a client already in the system or a name matching one of the three reserved words.
- Upkeep of the arrays in shared memory segments is up to you. Make sure that all unused spaces are available. One way to keep track is to add a bool to each slot that indicates whether it is in use (make sure you initialize them).
- Accesses that are not specifically mandated (server or client is to access) are your design decisions. An example of a mandated access protocol is that child lookup servers decrement the numClients field of the shared memory and the parent increments this value.
- The project will be tested with clients on the three machines acad, yin, and yang (i.e. three local servers).
- Anytime a process starts out of order (client before lookup server), the system should handle the situation gracefully. For example, suppose a client starts up and wants to register with the LOOKUP_SERVER. When the connection is refused (for lack of anything listening at the other end) the client should exit, cleaning up (if necessary).
- A client can send a message to itself.
- If a signal interrupts a non-reentrant function (accept() is not reentrant; find out about select) you must restart the function (use a loop).
- The name of a destination or command on the command line should be case insensitive.

Turnin:
Turnin all files, and a makefile. The executables should be named *server* and *client.* They should be able to be made individually, but entering *make* with no argument should make them both.