CSC 552 Project 3

Points: 45+5 (presentation)
Due: See Deliverables section. Submit late at your own risk.
Purpose: Implement a Distributed information System; use sockets, semaphores, shared memory, signal processing

Description: This project will develop a distributed system that services inquiries to your data set as was done in Project 2. While implementing this project, you will work with TCP (Stream) sockets, shared memory and semaphores, and signals.

There will be two programs to write, a data server for the system and local clients that can be run on each machine. Following are descriptions of these programs:

DATA_SERVER

The DATA_SERVER must be "well known" to all the clients and will have a stream socket listening on your assigned port on mcgonagall. Once the DATA_SERVER is initialized it starts waiting for connection requests from any client that can reach mcgonagall, on your assigned port. Due to security concerns, clients accessing this server may be running on mcgonagall, cats, or calm.

DATA_SERVER's purpose is to provide a means by which accesses of the data (your chosen set) can be facilitated as well as maintaining a log of all operations carried out on the data. When it starts it must carry out several tasks before beginning its loop handling requests for service:

- Open the binary encoded data file for reading and writing
- Open the server log file for reading and writing

The process file table is duplicated on a fork, enabling all forked server children to directly access these files without need to open them. The parent server will close them when it shuts down

Each time the DATA_SERVER receives a connection request from a client, it should service it using the usual TCP/IP method, forking off a child DATA_SERVER which, from then on, handles all requests made by that client. Thus, each client will have its own dedicated child DATA_SERVER. The DATA_SERVER should also keep a count of the number of child DATA_SERVERs. This count is incremented each time a new DATA_SERVER is forked off and decremented each time a child DATA_SERVER exits the system (the parent DATA_SERVER will be informed of this event).

The accept() fired by the client's connect() provides a sockaddr_in structure that the child server process dedicated to it will use to properly log messages. The child server should log the client's arrival with its IP and port, obtained from the sockaddr_in.

When a user selects an item off the application's menu (similar to last time), the client process will contact its dedicated child DATA_SERVER, which will carry out the command and respond with the required information. It will then get mutually exclusive access (mutex) to the server log file and update the log with an entry containing the requestor's IP address, port, and the system command received.

Since both the binary data file and the log file are accessible for both reading and writing by multiple child DATA_SERVERs, access to each one represents a critical section, and you must synchronize access to it by using semaphores. More on this later.

Client

The first thing a client does on startup is to try to connect with the DATA_SERVER. A client that successfully connects with the data server but finds no local IPC will assume it is first on the platform and allocate semaphore sets and shared memory for clients on that machine. It also initializes the #Clients and its own entry 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 semaphore infrastructure (check first, before shmem) in place, it will not allocate any IPC, and use the existing structures. If the DATA_SERVER is not found the client should terminate gracefully.

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. Its layout in the system drawing. Each client will initialize the next free space in the cli_dat segment with its pid and save the current time as the start time. The #Commands and lastMsgTime fields are initialized to 0. Whenever a client sends or receives a message, it will update its #Commands and lastMsgTime fields.

A client must monitor input from the user as well as the socket. It thus must use the select() command to wait for either user input or messages sent from other clients. The demo *termdemo.c* demonstrates use of select(), the tc commands for manipulating the terminal, and the FD commands for setting up the select() and determining which input source has data.

After each user input is handled the client will get mutex on the local shared memory and update the #Commands and lastMsgTime for itself.

The command menu will be updated as follows:
- A new command to list the pid of all clients on the local machine.
- Logging is now split into the server log and the client log. Each prints the entries in the log in a well-labeled manner. Tabular formatting is (minimally) suggested.

These commands are not logged.

Local Shared Memory & Semaphores

Use getuid() as the key for allocating semaphore sets to guard client logs and shared memory. To avoid any conflicts, use your port number as the key for the semaphores on mcgonagall to guard the binary data file and server log.

<u>Semaphores</u>

The readers-writers algorithm should be implemented for all critical sections, as all 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. For example, a display command from the user requires only read access to the binary data. Similarly, a request to display the log file is also a read only and does not require mutex.

Some user commands, e.g. the change command, require multiple access of the binary data, some that can be concurrent and others that require mutex in the data. Your program is required to have a change command, as it will be a good test that your design is effective.

You will need to use two semaphores to guard access to each file or shared memory segment. On the DATA_SERVER, allocate a set of four to guard the server log and binary data files. If a client comes on mcgonagall, it will allocate a separate set of four, two each to guard the clients' shared memory and log file.

Signal Handling and Shutdown

You will need to use signals in the project. In the DATA_SERVER program, the child DATA_SERVERs send the SIGCHLD signal to the parent DATA_SERVER when it exits. The child DATA_SERVER processes should call the exit() function when they are done, automatically generating the 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 DATA_SERVER exits, the parent DATA_SERVER (in the signal handler?) will decrement 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 DATA_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: If the user chooses to shut down, IPC **must** be properly removed and files closed. If you choose to send a SIGINT (^C) to the server, if you are running it in the background be sure to bring it to the foreground (if not already there) to be able to send the signal from the keyboard and handle the choice. You will need to bring it forward when the number of clients becomes 0 as well.

Client programs need to check if they are the last client on a machine when they exit the system. At that time they 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 and closes the log file.

Assume that any DATA_SERVER that catches a SIGINT while it has clients associated with it will ignore that signal. Further, signals will be blocked as follows:

- The parent DATA_SERVER can only catch a signal while waiting to do an accept().
- A local server can only catch a signal while waiting for select()
- Clients will ignore SIGINT.

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. There are several examples in the public/demo directory.

E. Notes

- Log files are shared and require implementation of the readers-writers algorithm for most efficient access.
  - Server log is named log.ser
  - Client log on each machine is named log.cli.<machine>
- The names of the data and log files are to be hard-coded into your programs.

- Data from the data file or logs is to be handled in single datum, not en masse.
- Upkeep of the 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).
- Processes that set up IPC should leave semaphores at the blocked setting after creation until everything else is ready. Then semaphores can be unblocked and access can proceed.
    - Semaphore sets must be set up before shared memory, as a first client must be able to set things up before others can access.
- It is a good idea to encapsulate semaphore calls. In the testSemUNDO example, functions P() and V() are implemented. You should expand upon this idea by writing functions getConcurrentAccess(), freeConcurrentAccess(), and maybe also getMutex(), and freeMutex() to make your code cleaner and more efficient.
- Slow the server so you can test the select(). Sleep for one second after receiving commands through the socket.
- Accesses that are not specifically mandated (server or client is to access) are your design decisions.
- The project will be tested with clients on mcgonagall, cats (`000721cats`) and calm (`k020022calm`). The latter two are virtual machines.
- Anytime a process starts out of order (client before data server), the system should handle the situation gracefully. For example, suppose a client starts up and makes a request to connect with the DATA_SERVER. When the connection is refused (for lack of anything listening at the other end) the client should exit, cleaning up (if necessary).
- If a signal interrupts a non-reentrant function (accept() is not reentrant; find out about select) you must restart the function (use a loop as in sersig.c).
- Make sure your programs properly remove all IPC. On mcgonagall you may need to remove two semaphore sets and a shared memory segment, as a result of an exiting DATA_SERVER and a last client leaving.
- This project will be graded using scripts. You will be provided at least one example of such a script; this provides an opportunity to test your system under a more realistic load.
    - Projects that don't run with scripts can't really be tested effectively. You MUST provide input scripts, named as required, and a working shell script, e.g. cli.sh.
- A Doxygen site should be prepared, with the link provided in your readme. Be sure to have a mainpage that describes the project and descriptions for files.
- A zip file that contains at least the binary data file and may also contain log files should be submitted so the instructor can establish ownership of the files to be able to test the program properly.
    - Zip files should not have subdirectories. They should unzip to the current directory when an *Extract Here* is requested.

Deliverables:

D2L:Place a readme that contains the following in the Phase 3 dropbox.

- Doxygen link
- Bug Report. If anything is amiss, tell me here. It will go better than if I figure it out without being alerted.
- Design decisions. How did you handle shared memory? If you add or alter anything from the drawing, explain it.
- Directions. How to build and run it.

Turnin (to acad)**:**

- Code files with makefile. The executables should be named *server* and *client.* They should be able to be built individually, but entering *make* with no argument should build them both.
- Three input scripts for a client, named **inputA**, **inputB**, and **inputC**, along with a script file that runs them in the background concurrently. You may submit multiple shell scripts.
- You may also submit the readme to acad, if it is plain text.