CS39002: Operating Systems Lab
Spring 2015
Assignment 6 (Shared Memory)
Development of Conferencing System
Due on: April 3, 2015 (EOD)

1. Overview: Conferencing System

In this assignment we shall explore the domain of shared memory by building a conferencing system. In doing so, our goal is to combine this new method of IPC with previously learnt methods such as semaphores and message queues and see how they work together in a multi process environment.

There are two tiers of the system: server and client. The server initiates the conference and proceeds to figure out who all have logged in and subsequently sends them a message saying: "COMMENCING CONFERENCE: RESPOND". The logged in users may or may not wish to participate in the conference. If they are interested (*active users*), they run the client program and receive all meta information required for inter process communication (thus all clients share the same information and may now communicate). When a client wishes to send a message, it writes the same to a shared memory segment. The server reads this message and broadcasts it to all other active users using message queues. Of course, at one time, only one user is allowed to send a message (achieved using semaphores). Finally, each user can log off by saying "*bye*". And when the last user says bye, the server frees up the resources and ends the conference.

Now that we have a conceptual overview, let us go deeper into the specifics of the program.

2. Components of the system

- **Server:** The server is responsible for initialization of IPC resources, connecting with active users and subsequently broadcasting messages in the chat system.

Task 1: Initialization of IPC resources (message queue, shared memory segments and semaphores)

1. Message Queue (*mq*): The message queue is used by the server to broadcast messages to all active users.
2. Shared Memory Segments: We shall need two shared memory segments in this system:
    a. PID array (*shm1*): This is an array of the pids all active clients in the system. The array is updated by the clients as they enter/leave the system. The server uses this array to identify the processes to which it must send messages (using the message queue). The size of the array may be assumed to be 500 bytes.
    b. Message (*shm2*): This segment is used to store the message itself. Whenever a client wishes to broadcast a message, it stores the message on this memory segment. The server then reads this and broadcasts the message to all other clients. The size of the message may be assumed to be 5000 bytes.

    Refer to functions shmget, shmat.
3. Semaphores: Two semaphores are needed for this system.

a. The first semaphore (*sem1*) is needed to provide mutual exclusion to the PID array. This is needed because multiple clients may enter/exit the system at the same time and may try to modify the array at the same time.
b. The second semaphore (*sem2*) is needed to provide mutual exclusion to the message segment. The message segment is either accessed by the clients (while writing the message) or by the server (when reading and broadcasting the message). However, this is slightly different from the standard readers-writers problem. This is because we want the server to read after every new write by a client. We achieve this using the method outlined in **Appendix B**.

Task 2: Initializing the server and identifying active users

- The main function accepts command line arguments which are actually a list of allowed users for the conference (eg. ./server sau sau2 Guest) where *sau, sau2* and *Guest* are various logins on the Unix machine. In a practical scenario, we may want to do this because we may selectively wish to allow only a few logins to participate in the conference.
- It then proceeds to writing its pid in a file called ser.txt. The file is opened under the O_CREAT and O_EXCL permissions. Thus, open will fail if the file already exists. If it does, exit the program. This is done to ensure that only one server exists and that only singular copies of SHMs, semaphores, etc exist.
- Unix-like systems contain a file called UTMP which contains information about the users currently logged into the system. The logged in users are filtered by the list of allowed logins obtained previously. Once the allowed logged in users are identified, the server sends the commencement message ("COMMENCING CONFERENCE: RESPOND") to their corresponding terminals (via *commence.c* code). If the user wishes to participate in the conference, it then runs the client program. By doing so, it becomes an active user. The technical detail of this component is outlined in **Appendix A**.

Now that the basic environment is set, the actual communication can start.

Taks 3: Broadcasting messages: This parts occurs in an infinite loop.

- After initialization, the server waits on *sem2* until a client writes a message.
- Once a message is written, the server reads the message. The format of the message is: <login_name>/<pid_of_sender>:<msg> (elaborated in client).
- It first checks if the msg is the special message ".". "." (dot) is a special message which indicates that the client doesn't have anything to send in this iteration. If the message is a ".", the server simply ignores the message and doesn't broadcast it to any user.
- It then checks if the msg special termination msg '*'. If so, it moves on to Task 4.
- If not, the server sends the message to all active users (i.e., all users present in the PID array) except for the sender using the message queue (the sender's pid is part of the message structure). Follow **Appendix B** to achieve mutual exclusion.

Task 4: Termination

A message "*" is a special message (explained later in client) which is sent when the last user sends *bye*. If the server receives the message "*", all resources are released (message queue, shared memory segments, semaphores, ser.txt file) and the program exits.

- **Client**: The client performs two jobs: sending messages and receiving messages. We'll achieve this by running two asynchronous processes thus ensuring a duplex form of communication.

Initialization

- The client checks if the server exists by checking if the file "ser.txt" exists. If not, it exits.
- IPC resources (message queue, shared memories, semaphores) are attached and initialized.
- The client then forks the program. The parent runs the program to send messages while the child runs the program to receive messages.

Receiver (child)

- Initially prepares itself to receive the messages by adding its parent's pid to the pid array protected by *sem1*.
- Initiate an infinite loop:
    - In each iteration, receive a message from the server via the message queue and print the message.

Sender (parent): Sender executes an infinite loop.

The sender maintains a local variable msg whose default value is set to be "." (dot) which indicates that the client has nothing to send. When a client wishes to send anything, it fires the SIGINT signal (Ctrl+C). When the same is fired, msg is read from the user.
Note that the receiver needs to ignore the SIGINT signal.

In each iteration of the loop, the following steps are followed:

- First the message must be prepared. The following format is followed to construct the message: <login_name>/<pid_of_sender>:<msg>. login_name is extracted using the function getlogin(). The message (msg) is entered by the user.
- Once the msg is entered, the client checks if it is "*bye*". If it isn't, it is simply copied to the shared memory protected by *sem2* (Follow **Appendix B** to achieve mutual exclusion.). The local variable msg is set back to ".".
- The client enters *"bye"* when it wishes to terminate. If the msg entered was in fact "*bye*", we must remove the pid of the corresponding user from the PID array (so that it doesn't receive any future messages). Access to PID array is protected by *sem1*. Further, if there is only one pid remaining in the PID array, we know that this is the last process. Set the message to be "*" so the server knows that the last user has logged out and the conference may now be ended.

Since the sender and receiver processes are asynchronous, it is possible that the sender process sends prepares and sends a message before the receiver registers it on the PID array. This is considered tolerable since not being registered only prevents the client from receiving the messages and has no effect on the sending.

.

Deliverables

Submit three files (commence.c, server.c, client.c).

Appendix

Appendix A: Identifying users of the system and commencing communication.

In this assignment, we shall introduce a more formal interpretation of users of the system which is more in sync with how Unix-like system defines users.

Defining Users

We explain this part with a small exercise. Open a unix terminal and enter the command who. Now open another terminal and type the command again. What do you see?

The who command shows information about users who are currently logged in. Thus different terminals act as various logged in users represented by device ids like ttys***. Note that a single login may have various active devices.

The meta information related to the users is stored in a special UNIX file called utmp (http://en.wikipedia.org/wiki/Utmp, http://man7.org/linux/man-pages/man5/utmp.5.html) that keeps track of all logins to the system. The file is a sequence of utmp structures defined in <utmp.h>. We shall only be using ut_name and ut_line (read their descriptions).

Additionally, the logins are associated with entries in the password database of the system (the passwd file). The entry can be accessed using the command getpwnam (http://linux.die.net/man/3/getpwnam) with ut_name as parameter.

Getting a list of allowed logged in users

- The UTMP file is opened (the location may be specific to your operating system). Code the address of the file based on the m/c suitable for a demo.
- Loop through the UTMP file:
    - Read each entry as follows:
        - struct utmp u;
          fread(&u, sizeof(u), 1, fp); /*fp for UTMP file*/
    - In addition to the list of users, the UTMP file consists of some other details as well. To ensure that the struct really does point to a real user, we call getpwnam on ut_name (if ut_name is not NULL) as described previously. If ut_name points to a real user and not some random information, it would have an entry in the passwd file. Thus the result of getpwnam would not be NULL.
    - Once we know that a user is legitimate, we check if that user exists in the list of allowed users for the conference (present in argv). If so, it is deemed to be an allowed logged in user.

Commencing communication with active users

The server sends a commencement notification to all allowed logged in users. If the user wishes to participate in the conference, it can do so by running the client program on its terminal thus becoming an active user.

We send the commencement notification to the corresponding user using the following code snippet. This runs the program commence and redirects the output to the corresponding user. Thus the terminal representing the user would be cleared and print the commencement message:

sprintf(name, “commence > /dev/%s”, u.ut_line);
system(name);

Here commence is a simple program (commence.c) which simply clears the screen (system(‘clear’)) and prints “COMMENCING CONFERENCE: RESPOND”. It exits after that.

Appendix B: Providing mutual exclusion for the message segment

As described earlier, the message segment is either accessed by the clients (while writing the message) or by the server (when reading and broadcasting the message).

The interesting part of this problem is that we want the server to read after every new write by a client. This can be achieved using a simple trick outlined as follows:

- Say the initial value of the semaphore *sem2* is 0.
- The server waits until the value of *sem2* is 2.
- When a client wishes to send a message, it reads the message from the user and then waits until the value of *sem2 is 0.*
- Since the initial value of *sem2* was 0, one client enters the critical section setting the value of *sem2* as 1 thus preventing other clients from entering their critical sections. Note that the server is still waiting.
- When the client copies its message into the message segment, it exits its critical section setting the value of *sem2* as 2. Other clients still can’t enter their critical sections since they are waiting for the value of *sem2* to be 0.
- Now that the value of *sem2* is 2, the server enters its critical section where it reads the message from the message segment and broadcasts it via the message queue.
- The server exits its critical section setting the value of *sem2* back to 0.
- The next waiting client may now enter its critical section and the process continues.

Note that since the client scans the message before waiting to enter its critical section, it may receive multiple messages before it actually sends out its message. Thus, in a strict sense, the sent messages aren’t synchronous and may not be ordered according to their timestamps. This is an acceptable limitation of the design of this system.

Sample Output

Server	**Device ttys001 (pid: 4997)**
--- Initialization complete --- Sending commence notification to ttys001 --- Sending commence notification to ttys002 --- Sending commence notification to ttys003 --- Received msg “sau/4997:Hello” --- Sending msg to pid 5001 --- Sending msg to pid 5005 --- Received msg “Guest/5001:World” --- Sending msg to pid 4997	<Ctrl+C is pressed> --- Enter your message: Hello --- Received message: “Guest/5001:World” <Ctrl+C is pressed> --- Enter your message: bye

--- Sending msg to pid 5005 --- Received msg “Guest/5005:*” *(The order of receiving messages may be reversed)* --- Terminating conference.	
Device ttys002 (pid: 5001) <Ctrl+C is pressed> --- Enter your message: World --- Received message: “sau/4997: Hello” **OR** --- Received message: “sau/4997: Hello” <Ctrl+C is pressed> --- Enter your message: World <Ctrl+C is pressed> --- Enter your message: bye	**Device ttys003 (pid: 5005)** --- Received message: “sau/4997: Hello” --- Received message: “Guest/5001:World” **OR** --- Received message: “Guest/5001:World” --- Received message: “sau/4997: Hello” <Ctrl+C is pressed> --- Enter your message: bye