TF-X Mission and Weapon System Software Team

Programming Assignment

Duration: **1 Day**

Date Last Edited: February 16, 2022

In this assignment you will implement a shared memory based multicasting queue. In a regular queue, each message is read by a single process and once it is read, another thread cannot read it. In multicasting operation, a message is delivered to all processes that are interested in.

In the assignment, a server process creates the shared memory and constructs the queue data structure. Afterwards, it starts accepting UNIX domain stream connections. A client process connecting the socket gets the information about the shared memory of the multicast queue. Also clients send their messages to the server using the socket connection. Each message sent by a client is inserted at the multicast queue and readable by all connected processes (including the original sender). Server forks a new process per connection and this child process serves the commands from the client.

The socket connection maintains the session and lets clients push messages where the multicast queue communications is established over the shared memory. See Figure 1.

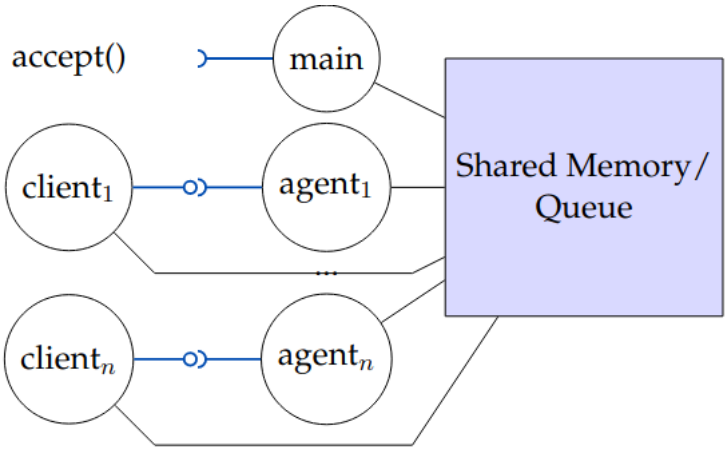

Figure 1: *Architecture of the system*

When the system working there is one server instance, n client instances and n agents which are children of the server. Note that the server and agent code is the same. The server pseudo-code is as follows:

```
create and attach shared memory
create queue data structure
create synchronization variables
bind and listen unix stream socket
while ns = accept()
    if (fork())
                close(ns)
    else
                agentcode()
```

The agent pseudo-code is as follows:

```
send shared memory informatin over socket
while command=readline
        if command == SEND messagebody
                insert messagebody in the queue for N readers
        else if command == QUIT or EOF
                decrement N
                exit
```

The agents primary responsibility is to provide the multicast semantics, update shared memory data structures so that n readers can read the same message. n is the current number of connections.

The client pseudo-code is as follows:

```
connect server
read shared memory information over socket
attach to shared memory
set mode to NOAUTO
while command = readline (stdin)
        if command == SEND messagebody
                send command over socket
    else if command == FETCH
                receive message  from shared memory queue (blocking)
                write message on stdout
        else if command == FETCHIF
                receive message from shared memory queue if exists
                if available, write message on stdout
        else if command == AUTO
                set AUTO mode
                start a thread that:
                        while True:
                                fetch message from shared memory queue
                                write message on stdout
        else if command == NOAUTO
                set NOAUTO mode
                cancel AUTO thread if exists
        else if command == QUIT or EOF
                close connection
                clear all remaining messages from queue
```

The client only reads from the UNIX domain socket at the beginning of the connection. If you choose to have SysVR4 IPC, it will be the key of the shared memory, if you choose to have memory mapped I/O, it will be the path of the file to map on memory. For the remainder of the operation, it only reads from stdin and writes on socket or the stdout based on the user request.

When the client gets a SEND command from user, it writes the command to push the message to multicast queue. The other commands do not use the socket. The client also interacts with the shared queue in two basic modes: AUTO and NOAUTO. In the NOAUTO mode, the queue read is on demand basis, user enters a blocking FETCH command to read next message on the queue. Also there is a FETCHIF command that fetches a message if available, otherwise reports 'No new message' on stdout. In the AUTO mode, client automatically fetches messages as soon as they arrive with help of the shared memory synchronization between the processes.

When client terminates, there might be messages that it did not fetch yet. Client has to clean them up so no messages will be left garbage.

Implementation

You are free to choose between SystemVR4 shared memory and memory mapped I/O for establishing the shared memory. You will need to implement all data structures for the queue in this area. POSIX thread synchronization tools, mutexes and condition variables work for processes as well if they are stored in shared memory and special attributes are set. See *pthread_mutexattr_setpshared* and *pthread_condattr_setpshared*.

There is an important restriction of the implementation, the messages should not be duplicated in the shared memory. The message has single copy and the queue has n references to the message. A typical data structure that you can use is the circular buffer. You can choose to have n circular buffers or implement n connections in a single circular buffer. However multiple buffers should contain the references to messages,

not the messages themselves. Otherwise single copy restriction is violated.

The data structures like queues are dynamic where the size of the shared memory is fixed when it is created. In order to solve this problem, the following assumptions are made:

- Number of maximum alive (not completely read) messages is bounded by 1024.
- Number of concurrent connections is bounded by 100.
- Maximum length of a message is 512 bytes.

You can use these numbers to calculate the maximum size for your shared memory and allocate maximum number of messages assumming the maximum message size. However you need to allocate/deallocate messages out of this fixed pool. That means you also need the allocation information in the shared memory. You are free to chose any structure as long as you follow the single copy rule.

No need to mention, your code should be free of deadlocks, busy waits and sleeping delays. The clients should unblock as soon as a message is available.

Execution

Your code will compile into two binaries *srv* and *cli*. Client and server get a single command line argument, the path of the UNIX stream socket.

Questions

Provide a .tar.gz file (no zip, rar or fancy tools!). The Makefile should compile your C or C++ code into *srv* and *cli*. All sources, makefile and compiled binaries should be in the same directory (directory hiearchy makes the evaluation harder).

This assignment is adapted from material of CENG536 Advanced Unix / Dept. of Computer Engineering / Middle East Technical University.