Assignment: Process Management and Distributed Computing

Due Date: **Sunday 27th October 2019, 11:59 PM**
Weighting: **40%**
Group or Individual: **Individual/Group of 2-3**

Assignment Description

You have been commissioned to develop a client/server system for an online multi-client message server. The purpose of this service is to manage a queue of text messages to be delivered to connected/subscribed connections. All connected clients can add a message and each message is associated with a channel ID. Clients are subscribed to multiple channels and when any of the subscribed channels has a new message, the clients can fetch and see it.

The client and server will be implemented in the C programming language using BSD sockets on the Linux operating system. This is the same environment used during the weekly practicals. The programs (clients and server) are to run in a terminal reading input from the keyboard and writing output to the screen.

You will have acquired all the necessary knowledge and skills to complete the assignment by attending all the lectures and practicals, and completing the associated reading from the textbook.

Server

The server will take only one command line parameter that indicates which port the server is to listen on. If no port number is supplied the default port of 12345 is to be used by the server. The following command will run the server program on port 12345.

```
./server 12345
```

The server should not be able to take any input from the terminal once it is running. The only way that the server is to exit is upon receiving a SIGINT from the operating system (an interrupt signal). SIGINT is a type of signal found on POSIX systems and in the Linux environment this means ctrl+c has been pressed at the server terminal. You need to implement a signal handler and ensure the server exits cleanly. This means the server needs to deal elegantly with any threads that have been created as well as any open sockets, dynamically allocated memory and/or open files. When the server receives a SIGINT, it is to immediately commence the shutdown procedure even if there are currently connected clients.

In the 9th edition of the textbook, Section 4.6.2 **Signal Handling** (page 183) has an introduction to signal handlers and Section 18.9.1 **Synchronisation and Signals** (page 818) has a detailed description of signals on Linux. You can also type **`man 2 signal`** and **`man 2 sigaction`** to read about system calls that can be used to set up signal handling on Linux. Be sure to also read **`man 7 signal-safety`** to learn what you can and can't do from inside a signal handler.

Clients communicate with the server using commands, the details of which are described below. You will need to implement these commands and a protocol suitable for communicating them in the server.

Client

The client will take two (2) command line parameters: hostname and port number. The following command will run the client program connecting to the server on port 12345:

```
./client server_IP_address 12345
```

A client will receive a welcome message from the server, which must be displayed to the user. The welcome message will include an ID number automatically assigned to the client. e.g. "Welcome! Your client ID is <client id>."

The client's user interface is all console based. No bonuses will be given for complex user interfaces. The user will enter in commands (followed by the enter key) to subscribe to channels, send and receive messages etc.

Submission and External Libraries

You need to submit your project in the form of C source code and a Makefile. C++ and other languages are not permitted. You are not permitted to use any library other than the standard C library, with the following exceptions:

- librt (necessary for some shared memory routines.) Link this by passing `-lrt` to GCC.
- libpthread (necessary for implementing Task 3). Link this by passing `-pthread` to GCC.

You should set up your Makefile so that both the client and server programs are built after typing 'make'. The programs must produce binaries named 'client' and 'server' respectively, as the marker will type these to test your program.

Task 1: Client-Server Computing

You only need to attempt this task if you are aiming to achieve a mark up to 60% for this assignment. Server/client socket implementation is worth 25% mark.

In this task, you will implement a single user message server (that is, only one client needs to be able to connect at a time.) The user can run the following commands which you need to implement:

SUB *<channelid>*

Subscribe the client to the given channel ID to follow future messages sent to this channel. channelid is an integer from 0-255. If the subscription is successful, the server will send a confirmation message to the client. The message should be in the format "*Subscribed to channel <channelid>.*" If the channel ID is invalid, a message like "*Invalid channel: <channelid>.*" will be displayed. If the given channel ID is valid but the user is already subscribed to this channel, the message shown will instead be "*Already subscribed to channel <channelid>.*"

CHANNELS

Show a list of subscribed channels together with the statistics of how many messages have been sent to that channel, how many messages are marked as read by this client, how many messages are ready but have not yet been read by the client:

- The first value counts messages since the start of the server and includes messages before the client subscribed to this channel.
- The second value counts messages that have been read by the client. This means messages sent after the client subscribed to the channel that have been retrieved with the commands NEXT / LIVEFEED.
- The third value counts messages that have not yet been read by the client. This means messages sent after the client subscribed to the channel but have not yet been retrieved with NEXT / LIVEFEED.

The list is sorted by channel order. Each channel is on a separate line, and values are delimited with a tab character. This command will *not* list channels that the client has not subscribed to. If the client has not subscribed to any channels, this command will display no output at all.

UNSUB *<channelid>*

Unsubscribe the client from the channel with the given channel ID. If successful, the server will send a confirmation message to the client, which the client must display to the user, in the format "*Unsubscribed from channel <channelid>.*" If the channel ID is invalid, a message like "*Invalid channel: <channelid>.*" will be displayed. If the given channel ID is valid but the user is not subscribed to this channel, the message shown will instead be "*Not subscribed to channel <channelid>.*"

NEXT <*channelid*>

Fetch and display the next unread message on the channel with the given ID. If the channel ID is invalid, a message like "*Invalid channel: <channelid>.*" will be displayed. If the given channel ID is valid but the user is not subscribed to this channel, the message shown will instead be "*Not subscribed to channel <channelid>.*" If the channel ID is valid and the user is subscribed but there are no new messages, nothing should be displayed at all.

Note that the only messages the client can receive are ones that are sent to the channel *after* the user subscribed to that channel.

LIVEFEED <*channelid*>

This command acts as a continuous version of the NEXT command, displaying all unread messages on the given channel, then waiting, displaying new messages as they come in. Using Ctrl+C to send a SIGINT signal to the client while in the middle of LIVEFEED will cause the client to stop displaying the live feed and return to normal operation. (If a SIGINT is received by the client at any other time, the client should instead gracefully exit.) The LIVEFEED command displays the same messages as the NEXT command if the channel ID is invalid or if the user is not subscribed to it.

NEXT

This version of the NEXT command (without the channelid argument) will display the next unread message out of all the channels the user is subscribed to. If the user is not subscribed to any channels, it will instead display the message "*Not subscribed to any channels.*" If there are no new messages to display, nothing will be output.

When displayed, the message will be prefixed with "*<channelid>:*" to indicate which channel the message came from. So if, for example, channel 127 has the unread message "Hello world.", the text "127:Hello world." will be displayed to the user if it is retrieved using this version of the NEXT command.

Note that the server will need to keep track of either the time each message was received or the order they were received in across all channels for this command to work.

LIVEFEED

This version of the LIVEFEED command displays messages continuously in the same way the regular LIVEFEED command works, except it displays messages that appear on any channel the user is subscribed to, like the parameterless NEXT command. Like the parameterless NEXT command, each line should also be prefixed with the channel it came from. If the user is not subscribed to any channels, it will instead display the message "*Not subscribed to any channels.*"

SEND *<channelid> <message>*

Send a new message to the channel with the given ID. Everything between the space after the channel ID and before the line ending is the message that should be sent. You can assume a maximum message length of 1024 bytes. Users that want to send longer messages will have to divide then up into 1024 byte segments.

If the channel ID is invalid, a message like "*Invalid channel: <channelid>.*" will be displayed. It is legal to send a message to a channel, whether the user sending the message is subscribed to that channel or not.

BYE

Closes the client's connection to the server gracefully. This also effectively unsubscribes the user from all channels on that server, so if the user reconnects, they will have to resubscribe. The same thing should happen if the client is closed due to a SIGINT command being received outside of LIVEFEED mode.

Note that **it is imperative that the server continue to function** even if the client disconnects ungracefully.

Task 2: Multiprocess Programming & Process Synchronisation

You only need to attempt this task if you are aiming to achieve a mark up to 80% for this assignment.

The server you have implemented in Task 1 can only handle a single request at a time. You are required to extend the server to accept multiple concurrent connections. You are asked to write a multiple process application for such a multicasting service. Each connection is served by a forked new process. In order to maintain a shared message board, you need to use shared memory (see Lecture 3 and Practical 3).

Another important constraint is that you are not allowed to duplicate messages internally – a message received by the server should only be stored once. This is to ensure efficient use of resources in a scenario with a very large number of messages and a large number of users. Hence the server will need to keep track of which message each subscribed user is up to in each channel's message queue. Dynamic memory allocation (malloc() and free()) should be used to allow for an arbitrary number of messages, of an arbitrary length.

You will need to consider the critical-section problem when implementing this task because a channel is a database that is shared among all users who subscribed to it, each of which may read or update the channel. You must ensure that no connection can make modifications to the state of a channel while other connections are reading it. However, it is fine to allow multiple connections to read messages from a channel concurrently.

You may consider this critical section problem to be an example of a Readers-Writers problem (as discussed in Lecture 5), and you may reuse the solution to the Readers-Writers problem in completing this assignment. A connection that is retrieving messages to display is considered to be a Reader in this model, while a connection that is sending messages to the channel is considered a Writer.

You may use semaphores or PThreads mutexes to synchronise your application. Note that, while all shared variables should be protected against concurrent access, the performance of the concurrent system should be considered. Global locks should only be used when strictly necessary as these can greatly reduce the performance of a concurrent system.

Task 3: Threading

You only need to attempt this task if you are aiming to achieve a mark up to 100% for this assignment.

In Task 1, there are some usability concerns for the client with the **NEXT** and **LIVEFEED** commands. Those commands don't return immediately (NEXT will normally return very quickly, but in the case of network congestion, the user may end up waiting for a short amount of time), and the client can't issue other commands in the meantime. In this task, we will create a thread for these commands, so that they return immediately. Update your **NEXT** and **LIVEFEED** commands with new constraints, and implement a new command called **STOP**:

NEXT <*channelid*>

NEXT

After issuing the NEXT command, the user will immediately be able to type in new commands. Once the message has been retrieved from the server, it will be displayed.

LIVEFEED <*channelid*>

LIVEFEED

After issuing the LIVEFEED command, all waiting messages will be displayed and new messages will be displayed as they come in, just as in Task 1. However, in Task 3 the user will be able to type in new commands and the LIVEFEED command will continue running in the background.

STOP

This command will immediately terminate all outstanding NEXT and LIVEFEED commands that have been issued.

Unlike with the server in Task 2, which must be multithreaded using fork() and exec(), the client in Task 3 must be multithreaded using either PThreads or OpenMP. Note that, if you are using OpenMP, you should manually control the number of threads created to ensure that parallelism is available even on single core systems.

Submission

Your assignment solution will be submitted electronically via Blackboard before 11:59pm on 27th October 2019. If you work in a group, only one member of your group needs to submit the assignment. Your submission should be a zip file and the file name should contain the student number(s) of your group members and include the following items:

• All source code of your solution.

• The ***Makefile*** used to generate the executables.

• A report in PDF which includes:

a) A statement of completeness indicating the tasks you attempted, any deviations from the assignment specification, and problems and deficiencies in your solution.
b) Information about your team, including student names and student numbers, if applicable.
c) Statement of each student's contributions to the assignment.
d) Description of the implemented data structure(s).
e) Description of how the forking is created and managed in Task 2, if applicable.
f) Description of how the critical-section problem is handled in Task 2, if applicable.
g) Description of how the threading is created and managed in Task 3, if applicable.
h) Instructions on how to compile and run your program. Include this even if no special instructions are needed.

Note: If the program does not compile on the command line in the Ubuntu Linux virtual machine available in the labs and for download from Blackboard, your submission will be heavily penalised. Implementations written in Visual Studio or other IDEs that do not compile on the Linux command line will receive a mark of zero (0).

Information for Groups

The choice to work on this assignment individually or in a group is up to you. The tasks you are required to complete are exactly the same in either case. However, managing your group will be up to you. Special consideration will **not** be granted in the case of, e.g. a group member not contributing the work they were assigned. As such, the advantages you gain by working in a group (the distribution of assignment workload) are considered to be offset by the risks inherent to working in a group. Neither the unit coordinator nor the tutors will mediate or otherwise interfere in your group arrangement. If you are concerned about this, the safest choice is to work individually.

All members of a group will normally receive the same marks. The only exception will be in the case where there is a serious disparity in the contributions between group members. In that case, group members may receive different marks.

You are **not** required to make use of source control (e.g. Git) when working on this assignment as a group, but it is probably a good idea. If you do make use of Git, **do not make your project public.** Both GitHub and Bitbucket allow the use of private repositories (you may have to sign up to GitHub using your QUT email address to gain access to this) and you should use this so that others cannot steal your code.