S-38.3600 UNIX Application Programming, Fall 2014

Assignment 2 - Diary

30.11.2014

Jaume Benseny, 463922, jaume.benseny@aalto.fi

INTRODUCTION

The present document describes chronologically how the assignment 2 was programmed. All of the stages describe a part of the program and more specifically include: how I though about it in the beginning, problems and solutions while coded, how it was tested and what it could be improved in the future.

General description and short usage instructions could be found in the main directory as file “short usage instructions.txt”. In addition older versions of the code can be found in the folder /old/.

FIRST STAGE - CLIENTS

The first version of the program implemented 10 threads that compete to get a position in 2 queues corresponding to 2 desks. Here the first mutex (QUEUE MUTEX) was implemented. Desks were initialized with queues equal to 0 and all ended having 5 clients. All clients got a queue number. The third step we created a mutex for each desk DESK MUTEX. Every client will have to access the mutex corresponding to its assigned desk. They will wait for 5 seconds and they will free the desk for some other waiting client to access. In addition, they have to get QUEUE MUTEX one more time to decrement its queue by 1. Code can be found in /old_versions/version3/. I got familiar with the MUTEX basic functionality and semantics.

MESSAGE QUEUE AND DESK MUETX

We created a matrix that will act a message queue as asked. Clients assigned to desk X are delivering commands to row X. Access to the row will be atomic and controlled by DESK MUTEX. Commands are randomly chosen by the clients from a vector of examples instantiated in their thread function. Clients write in the first position of the message queue (row) that is NULL. Version 4 of the old folder. In this stage it was important to decide how DESK MUTEX will be shared between one DESK and all clients assigned to the desk. As described before, QUEUE MUTEX is shared among all clients.

SERVANTS

Servants are the threads that work in each queue processing queue's messages. Each servant works in only one queue. Its access to the message queue is controlled by the DESK MUTEX. Servants read from the message queue one message at a time which means freeing the DESK MUTEX every time. In case there are no messages, they wait. Moreover, the servant writes 1 to the read position in the message queue and decrements its queue Lenght by accessing the QUEUE MUTEX (instead of the clients as done until now).

SYNCHRONIZATION BETWEEN CLIENTS AND SERVANTS – MUTEX CONDITION

To avoid servants accessing their DESK MUTEX to find that there is no message to processes, we have implemented a condition which allows clients to signal servants when they deliver one message. It was troubling to debug and detect errors but we used Valgind and it helped us find code errors. We had to introduce the EXIT FLAG (vector of flags, one per servant) that tells servants to exit the program. The flag is raised by the main thread when there is only one servant sleeping to exit the program. Access to the flag is controlled by DESK MUTEX.

BALANCE – FIRST TRY WITH BCONDITION

The main thread asks for the balance every X time by raising the BALANCE FLAG which is a vector with same number of flags as desks. This FLAG is controlled by the BALANCE MUTEX and all desks try to get it before serving any new client. When they get it, the flag is evaluated. In case balance is 1, they report the balance value and sleep until BALANCE FLAG is 0. Thanks to the condition bcond. Broadcast is not sent until all desks have reported their balance, they can't continue working because they are sleeping in the BALANCE MUTEX condition. (check version 7 and 10 in /old versions/). I realized this design was a nightmare of coordination, because evey time the main thread woke up had to check about the condition of all positions of the BALANCE MUTEX and broadcast the condition woke all desks including the ones that reported already. What made me change my mind was try to coordinate different DESK MUTEX with BALANCE MUTEX. I couldn't find a good solution.

MODULES AND LIBRARIES

At this point the code become intractable and I created different libraries to segment clients and desks functionalities. I created global headers to accommodate the data structure and defined it as extern.

ACCOUNTS - LOCKS

Accounts belong to clients but they are only accessed by servants after reading commands from message queue. Access was controlled by read (for balance) and write (transfer, withdraw, deposit) LOCKS. It's usage is very intuitive and practical.

CLIENT DATA ENTRY AND RE-ENTRY FREE FUNCTIONS

The fact that commands were introduced by the user required some additional effort regarding string parsing and control. In this case, I had to debug with valgrind again to identify the source of the problem. I had to use re-entry safe functions like strtok_r and avoid to pass static variables. strtok_r required careful handling of NULL pointers because at each iteration delimiter char could not be found. I decided to use QUEUE MUTEX to control how clients had access to stdout. To be more precise, after queue is assigned, they have access to the screen. I though it was efficient and was not creating any conflict. In a real scenario petitions would be accepted concurrently and therefore another synchronization system would be required.

BALANCE – SECOND TRY (BARRIERS AND BALANCE FLAG)

I thought that using barriers could be a good idea because SERVANTS can't serve new clients until all DESKS had reported. First, I took only SERVANTs in the barrier. This required to code the coordination of the asking client to collect the results. Finally, I realized that the main thread could also wait as part of the barrier and this way all wake up at the same time. The balance is delivered by SERVANTS to its correspondent positions in the vector BALANCE. Some extra synchronization was required and re-used from the first try. BALANCE[0][desk] is used as FLAG to indicate that balanced is requested. Balance[1][desk] is used to allocate the value itself. Access is controlled by DESK MUTEX. I decided to use as minimum MUTEX as possible. I learned that barriers must be executed out of MUTEX otherwise code gets into DEADLOCKs.

EXIT PROGRAM AND BALANCE

In order to make SERVANTS awake from the condition of DESK MUTEX (waiting for messages to be delivered) because bank is closing or balance is required. FLAGs have been implemented and checked by SERVANTS every time they are awaken.

LOG

Log function writes to FILE controlled by LOCKS. I first tried to re-use my library from the first assignment which implemented FILE LOCKS. After some tries I realized the execution was not thread-safe and I switch to LOCKS. The log system can be improved by the inclusion of all error messages in one same library.

TESTS

During 80% of the development of the code, all test have been performed by automatic introduction of commands predefined from a library. Therefore execution of threads has been randomly driven by the CPU and not based on sequential human data entry. At the beginning of the development I used to test with 10 clients and 2 desks but it has been increased up to 50 clients and 5 desks.

Valgrind has been really useful to identify non re-entry functions (strtok_r), non-initialized mutexes and null pointers. Although I have devoted effort getting familiar with Valgrind, I don't still fully understand it's feedback.

All possible commands have been validated and compared to expected account change. Multiple synchronization problems have been found along the development and have been solved as described in this document.

CONCLUSION

The fact that I used 1 MUTEX for QUEUE CONTROL, then multiple MUTEX (dmutex[]) and multiple conditions (cond[]) to control access to MESSAGE QUEUE and I have finally used a barrier to get BALANCE has given me great hands-on experience on multiple thread synchronization techniques.