CprE 308 Project 2: Multithreaded Server

Department of Electrical and Computer Engineering
Iowa State University

Fall 2016

This is a three-week programming project assignment, starting in the week of October 3-7, 2016, and will due in the week of October 24-29, 2016.

1 Submission

Submit to Blackboard by due date (starting time for your lab session), include the following in a zip/tar archive:

- A cohesive summary of what you learned in the project. This should be no more than two paragraphs.
- Your source code; commented and formatted neatly. Use good programming practices (create methods to abstract large bodies of code and code common to multiple methods, only use global variables when you must, and check syscall or system dependant libcall return values for errors – don't assume the OS will always succeed in fulfilling your requests.)
- A Makefile and instructions to compile the source (in a comment at the top of the .c file)
- The output of the provided test script when run on your program with a seed value of 0 for 10 worker threads on 1000 accounts.
- A short report summarizing your findings from Part II (Filled version of the report-template.doc).
- Include your report-template.doc, source code, Makefile and output in a single archive on Blackboard. Name your archive appserver-lastname-section.tar.gz.

Late Policy No late submissions.

2 Project

In this project, you will be creating a multithreaded server that manages access to a set of bank accounts. The program is started with a certain number of accounts in the bank, and user requests can perform transactions on these accounts, as well as query their current status. Note that the number of accounts in the bank remains fixed through the duration of the program.

Each request requires access to multiple records from a database, and also requires some processing time. Since many user requests may arrive simultaneously, servicing the requests sequentially, one after another, would cause a high average latency for the users; servicing the requests in parallel is required to keep response times small. Thus, your program has to use multiple threads to service the requests simultaneously.

User requests are presented via the keyboard. When the user types in a request, the server records the request and immediately presents the user with a transaction ID, and then continues to accept more requests. At a later point of time, when the request has been processed, the results will be written out. To avoid interfering with user input, the delayed responses will be written to a file rather than displayed directly.

Tip: you can open a second terminal and use `tail -f file` to follow the delayed output.

3 Requirements

The server should be written using C on a Linux machine, using the pthreads package. During initialization, the server should create a specified number of worker threads, which service user requests. The number of worker threads will be specified by the user as argument at the command line.

3.1 Threads Within the Program

The program should contain two types of threads, a "main" thread, and one or more "worker" threads. The main thread will initialize any state of the server and create all other "worker" threads. Each worker thread will service one request at a time. After servicing a request, the worker thread will service another request, and so on, until the end of the server. The program can be launched (and usually will be launched) with more than one worker thread. Thus, there are typically many worker threads, simultaneously processing different requests.

The syntax used to launch the program will be

```
$ appserver <# of worker threads> <# of accounts> <output file>;
```

For example, to run a server with 4 worker threads and 1000 accounts that outputs to the file "responses", you would run

```
$ appserver 4 1000 responses
```

Assume that the balance of each account is initialized to zero. Each account is assigned an "account id". If there are n accounts in total, the account ids take consecutive values in the range 1 till n.

3.2 Requests

A request will be a single line of input. The first word of this line will identify the type of request. You are required to handle 3 types of requests: transactions, balance checks, and program exit.

For transactions and balance checks, there should be an immediate response to the user, of the form:

```
ID <requestID>
```

The request ID should start at 1 and should be incremented each time a new request is issued.

3.2.1 Balance Check

The request line will be of the format:

```
CHECK <accountid>
```

An account ID is a positive (nonzero) integer less than the specified maximum account ID. The output of this request written to the result file have the following form:

```
<requestID> BAL <balance>
```

3.2.2 Transaction

```
TRANS <acct1> <amount1> <acct2> <amount2> <acct3> <amount3> ...
```

There may be from 1 to 10 accounts in any single transaction. The amounts will be signed integers representing the transaction amount in cents. For each pair consisting of an account id followed by an amount, the amount should be added to the account balance.

Significantly, all transfers within a transaction should happen as a single unit; This means each transaction is processed in a single thread, and no other thread may access the accounts involved in this transaction while the transaction is being processed. *If any account does not have a sufficient balance to satisfy the transaction, the entire transaction is voided and all accounts should return to their state at the start of the transaction.*

If a transaction was successful, your program should write a result of the following form to the output file:

```
<requestID> OK
```

If the transaction was unsuccessful because some account did not have sufficient funds, the following should be written to the output file:

```
<requestID> ISF <acctid>
```

where acctid is the account that had insufficient funds (there may be several; you only need to identify one).

3.2.3 End

```
END
```

No further commands will be processed from the user. All currently queued commands should be processed, and then the program should exit. You do not need to print a request ID for this command, but can if it is more convenient to do so.

Errors such as invalid input can be handled when a request is entered or in the worker thread when it is processed; they should not result in a successful transaction or a crash. You may also choose how to inform the user of the error.

3.3 Account Database Interface

We will supply you with an interface to an account database. This database stores all the accounts and manages access to them. You can interact with the accounts in the database through three simple functions.

```
void initialize_accounts( int n );
```

This function should be called once at the start of your program. It takes as a parameter an int n - the number of bank accounts. The function initializes the database with n accounts with IDs from 1 to n, and sets the value of each account to 0.

```
int read_account( int id );
```

This function returns the value of account id.

```
void write_account( int id, int value);
```

This function writes value to the supplied account.

You will be given the Bank.h and Bank.c files which contain these functions. You should not modify the contents of these files. Note that these functions provide no safety against concurrent read/writes or protection against writing to invalid accounts; that is the responsibility of your program to manage.

Because the accounts are all in-memory, while they would be stored on disk in a real banking system, these functions also artificially in ate the access delay so that they are similar to what is experienced in a real system.

3.4 Timing

In order to evaluate how long requests take to process, you are requested to include the system time when a request is received by the main thread, in addition to the time that the request has finished. To get more than second-level accuracy, use the `gettimeofday()` function with `NULL` as the second argument; this will provide microsecond timestamps. Include the timestamps in the same line as the request output, preceded by the word "TIME". Use the printf format `"%d.%06d"` to print out seconds and microseconds without needing to use floating-point numbers.

3.5 Command Buffer

You should have a "Linked List" data structure to hold those requests that have been received but not processed yet. When a request is received by the main thread, the thread should add it to the linked list. Each worker thread should pickup requests from the linked list and process them. If there are no requests in the linked list, the worker thread must wait until there is one. Remember to free any memory you allocate.

Because the buffer will be accessed by multiple threads, it needs to be protected by a mutex.

4 Tips

4.1 Account Storage

Since the accounts themselves maybe accessed by multiple threads, we should ensure that concurrent access to the same account is prevented. One way to protect the accounts from simultaneous access is through using a mutex for each account.

For example, an account maybe represented by the following structure:

```
struct account {
    pthread_mutex_t lock;
    int value;
};
```

4.2 Deadlocks

To correctly execute a transaction, a thread must hold multiple locks at the same time. For example, if a transaction has three accounts in it, the worker thread executing that transaction must hold locks for all three accounts at the same time. This behavior can easily lead to deadlocks unless our program is written carefully.

Your program should not lead to a deadlock. To avoid deadlocks due to the use of mutexes, one possible solution is to number the different mutexes in a consistent manner, and for each thread to lock lower numbered mutexes before locking a higher numbered mutex, and never the other way round. This will eliminate the circular waiting that is one of the requirements for a deadlock. Refer to the lecture on pthread mutexes for a discussion on this.

4.3 File Output

You can use the `fprintf()` function to output to a file; its usage is similar to that of `printf`.

The C standard library functions for FILE* objects are threadsafe; the FILE structure contains a mutex that all output functions acquire before using the file. However, if you execute two successive `fprintf()` calls, there is no guarantee that they will output in order. The functions `flockfile(FILE*)` and `funlockfile(FILE*)` are available if you need to ensure correctly ordered output.

4.4 Test Script

A Perl test script testscript.pl is included for testing. To run the script, run

```
./testscript <program> [<nthreads> [<naccounts> [<seed>]]]
```

For example, to test your program with 10 worker threads, 1000 accounts, and a seed of zero,use:

```
./testscript ./appserver 10 1000 0
```

When the test script is downloaded, it will not have execute permissions. To give execute permissions to the file, run the following command

```
chmod u+x ./testscript
```

5 Example

Input/output at the console. Lines beginning with a `>` are input, lines beginning with a `<` are output.

```
$ appserver 4 1000 requests
> CHECK 1
< ID 1
> TRANS 1 100000 2 100000
< ID 2
> TRANS 1 -10000 5 10000
< ID 3
> TRANS 2 -20000 4 10000 7 10000
< ID 4
> CHECK 1
< ID 5
> TRANS 2 -2000 1 1000
< ID 6
> TRANS 1 -10000 4 -10100 5 20100
< ID 7
> CHECK 4
< ID 8
```

The file "requests" might then contain:

```
1 BAL 0 TIME 1224783296.348723 1224783296.913123
2 OK TIME 1224783297.348254 1224783298.034256
3 OK TIME 1224783297.349756 1224783298.068902
4 OK TIME 1224783298.350484 1224783298.647822
5 BAL 90000 TIME 1224783298.348467 1224783298.609814
7 ISF 4 TIME 1224783299.548467 1224783299.741932
6 OK TIME 1224783299.478867 1224783299.834902
8 BAL 10000 TIME 1224783302.899871 1224783303.002389
```

Since the transactions are being processed in multiple threads, the output may not be in the same order as the input; this is the reason request IDs are used.

You do not need to ensure that the results are the same as if the commands were executed in a single thread, but you do need to ensure that the results are correct for *some* ordering of transactions. For example, summing the amounts of all OK transactions should be equal to the balance in a given account.

6 Part II - Locking Granularity Exploration

In this section you will explore a trade-off between coarse and fine grained locking. Until now, you have used fine-grained mutexes to protect the accounts by having a separate mutex for each account. This allows the most control over the individual accounts, but it can come at a performance penalty to process each individual mutex lock and unlock. In this section you will modify the project to instead use a single mutex to protect the entire bank to emulate coarse-grained locking.

6.1 Coarse-Grained Locking

Create a copy of your project. Name the new project appserver-coarse.c Modify the new version to lock the entire bank (every account) for each request. Note that this uses a significantly smaller number of mutex lock and unlock operations, but also leads to a significantly smaller concurrency among the worker threads.

6.2 Performance Measurement

Use the time command to measure the running time of both programs. For consistency, use the provided test script with the same parameters as before. The following commands can be used to determine performance:

```
time ./testscript.pl ./appserver 10 1000 0
time ./testscript.pl ./appserver-coarse 10 1000 0
```

Run the test multiple times on each program to produce an average. Be sure to use the "real" time from the output results.

6.3 Summary

Summarize your findings in a short report. Include your results in a table and be sure to address the following points:

- Which technique was faster - coarse or fine grained locking?
- Why was this technique faster?
- Are there any instances where the other technique would be faster?
- What would happen to the performance if a lock was used for every 10 accounts? Why?
- What is the optimal locking granularity (fine, coarse, or medium)?