CprE 308 Project 2: Multithreaded Server

Department of Electrical and Computer Engineering
Iowa State University

This is a three-week programming project assignment: starting in the week of October 21-25, and due in the week of November 11-15.

1 Submission

Submit the following items on Canvas:

- Your well-commented source code and a Makefile (or other compilation scripts with instructions) in a zip file.
- A report summarizing what you learned and your findings from Part II (Section 6). Use the report template `308-proj2-report-template.doc`

Grading criteria:

- 10 pts - program correctly compiles with the given makefile (or other scripts)
- 50 pts - program produces correct output without crashing using the given test script
- 30 pts - project report (a brief summary and part II questions)
- 10 pts - attendance in the first week of lab session (Oct 21-25)

Due Date 3 weeks after your first lab session, in the week of November 11-15.

2 Project Requirements

In this project, you will be creating a multithreaded server that manages access to a set of bank accounts. The program should maintain a certain number of bank accounts and provide two types of user requests: (1) query the remaining balance of an account, (2) post transactions on one or more accounts.

Your program should initialize a set of bank accounts and a set of worker threads on startup, and run an infinite loop accepting user requests until the user decides to exit. When the user types in a request, the program should delegate the request to a background worker thread and be ready to take new requests immediately. The background worker threads will process the requests and write the output to a file instead of printing to the screen.

You are given two files `Bank.c` and `Bank.h` which contain functions that simulate the access of bank accounts. You should include these files in your program and use the defined function to maintain the bank accounts. Note: **Do not modify the code in these two files**.

2.1 Program Input Parameters

The program should accept 3 additional parameters when it is started. For example:

```
./bank_server <num_worker> <num_account> <output_file>
```

Assuming your program name is "bank_server", the last 3 parameters:

- **num_worker** specifies exactly how many worker threads should be created to handle user requests.
- **num_account** specifies exactly how many bank accounts should be generated when the program starts.
- **output_file** specifies the output file path that would be used to store query results. (This is to avoid interference with the user input, since we are servicing requests in multiple threads simultaneously)

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

```
./bank_server 4 1000 responses
```

2.2 User Requests

The main thread of the program should be an infinite loop taking user requests from console input. The first word of each request will identify the type of request. You are required to handle 3 types of requests: balance checks, transactions, and program exit.

2.2.1 Balance Check

The balance check request is in the format of:

```
CHECK <accountID>
```

The parameter `<accountID>` is a positive integer less than the specified maximum account id.

After receiving this request, there should be an immediate response printed to the screen, in the form of:

```
ID <requestID>
```

The parameter `<requestID>` should start at 1 and should be incremented each time a new request (either CHECK or TRANS) is issued. When processing this request, the worker thread should write the following output in the output file:

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

2.2.2 Transaction

The transaction request is in the format of:

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

There may be from 1 to 10 accounts in any single transaction request. 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. **The main thread can choose to reject transaction requests that contain duplicated account IDs.**

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.**

Similarly to the CHECK command, there should be an immediate response printed to the screen, in the form of:

```
ID <requestID>
```

The parameter `<requestID>` should start at 1 and should be incremented each time a new request (either CHECK or TRANS) is issued.

If a transaction was successful, the worker thread 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).

2.2.3 End

The command to end the program is:

```
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.

2.3 Request Timing

In order to evaluate how long requests take to process, you are requested to include two time stamps in the output:

1. the system time when a request is received by the main thread
2. the system time that the request has finished in the output

Include the timestamps in the same line after each request output, preceded by the word "TIME". The timestamps should be in second-level accurary, including seconds and microseconds. For example, a "CHECK" request might have the following output in the output file:

```
1 BAL 100 TIME 1571715269.499487 1571715271.406340
```

You can use function `gettimeofday()` to get the system time and use `"%ld.%06.ld"` to format the time stamp. For example:

```
struct timeval time;
gettimeofday(&time, NULL);
printf("time is: %ld.%06.ld\n", time.tv_sec, time.tv_usec);
```

2.4 Handling Invalid Input

When the user enters an invalid input such as unrecognized commands, non-existent account IDs, or duplicate transactions inform the user of the error and move on. Such invalid input should not result in a successful transaction or a crash. You may also choose how to inform the user of the error.

3 HINTS

3.1 Synchronizing Account Access

Since the accounts themselves maybe accessed by multiple threads, we should ensure that concurrent access to the same account is prevented. To prevent such concurrent access, we can create an array of mutexes - one for each account.

For TRANS requests, the worker 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 all three account locks before processing the transaction. **This behavior can easily lead to deadlocks unless our program is written carefully.**

To avoid deadlocks, one possible solution is to lock mutexes in a sorted manner, e.g. locking the mutexes of smaller account IDs before locking those of higher account IDs. 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.

3.2 Buffering User Requests

To ensure that the main thread does not block when requests are coming in faster than they are being processed, you should maintain a queue-like structure to hold the requests that have been received but not yet processed. You can implement the queue as a Linked-List. New requests can be added to the tail end of the linked-list, while worker threads can poll requests from the head end. Below is an example of the Job and Job Queue structures.

NOTE: This example below is for reference only. You are free to implement the data structures in other ways.

```
struct trans{    // structure for a transaction pair
    int acc_id;  //    account id
    int amount;  //    amount to be added, can be positive or negative
}

struct job{
    int request_ID;  // request ID assigned by the main thread
    int check_acc_ID; // account ID for a CHECK request
    struct trans *transactions; // array of transaction data
    int num_trans; // number of accounts in this transaction
    struct timeval time_arrival, time_end; // arrival time and end time
    struct job *next; // pointer to the next request
};

struct queue {
    struct job *head, *tail;  // two pointers pointing to the head and tail of the queue.
                              // add jobs after the tail, take jobs from the head
    int num_jobs; // number of jobs in current queue
}
```

When there are no requests in the linked list, the worker thread must wait until there is one. You can use `pthread_cond_wait()` and `pthread_cond_broadcast()` calls to achieve such waiting. Most importantly, since the job queue will be accessed by multiple threads, it also needs to be protected by a mutex.

3.3 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 Test Script

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

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

For example, to test your program with 10 worker threads, 1000 accounts, and a seed of 0 (for the RNG), use:

```
./testscript.pl ./bank_server 10 1000 0
```

You might need to run the following command once to give the script execution permission:

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

The document **testscript-description.doc** contains a more detailed description of how this test script works.

5 Input/Output Example

Lines beginning with > are input, lines beginning with < are output.

```
$ ./bank_server 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
```

After the above commands, 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. 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. Include your results in the project report.

6.3 Summary

Based on your finding, answer the following questions?

- 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)?