CprE 308 Project 2: Multithreaded Server

Department of Electrical and Computer Engineering
Iowa State University

1 Submission

Submit the following items on Canvas:

- A zip or tar of your commented source code and Makefile. This should be the entire project directory - including Bank.h and Bank.c.
- A PDF of your completed report, using the template from Canvas.

Grading:

- 10 pts - program compiles with submitted Makefile
- 50 pts - fine-grained program produces correct output, without crashing, using provided test script
- 10 pts - coarse-grained program also produces correct output, without crashing, using provided test script
- 30 pts - project report based on template

2 Description

In this project, you will write a multithreaded program that simulates management of a set of bank accounts. This program is referred to as the *server*. When the server is started, a fixed number of bank accounts are created, and users can perform transactions on these accounts, as well as query an account's current balance.

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 must use multiple threads to service the requests simultaneously.

User requests are made via the command line. When the user types in a request, the server program records the request and presents the user with a transaction ID, and then immediately accepts more requests. The requests are processed in the background; when finished, the results of the requests are printed to a file in order to avoid interfering with user input.

The program should contain two types of threads: a "main" thread, and one or more "worker" threads. The main thread will initialize the server and create the "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 server exits. The number of worker threads is specified as an argument on the command line (see Section 3.1).

2.1 Account Database Interface

You are given Bank.h and Bank.c, available on Canvas. You are required to use these files, unmodified, for the project. These files provide an interface to the account database. This database stores all the accounts and manages access to them through the following functions:

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

This function should be called once, near the start of your program. The parameter is the number of bank accounts to initialize, which should be passed in to the server program as a command line argument (see Section 3.1). 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 the specified account.

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

This function writes the specified value to the specified account.

```
void free_accounts();
```

This function frees the bank accounts and should be called right before your program exits.

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, these functions also include artificial delays in order to simulate disk access.

2.2 Rules and Restrictions

The server should be written in C on a Linux machine, using the pthreads package. You may use the standard C libraries and other C libraries typically found on a Linux machine. When in doubt, ask a TA if a library is allowed. You must use the provided Bank.h and Bank.c files, and you may not modify these files in any way (except possibly for your own testing/debugging purposes).

3 Server Syntax

3.1 Running the Server

The syntax used to launch the server 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
```

Tip: you can open a second terminal and use `tail -f responses` to follow the request results output.

3.2 Requests

A request is a single line of input. The first word of the line identifies 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 first request ID should be 1, and the ID should be incremented each time a new request is issued.

3.2.1 Balance Check

A request for a balance check takes the format:

```
CHECK <accountid>
```

An account ID is a positive (nonzero) integer less than or equal to the maximum account ID. The output of balance check requests are written to the results file in the following form:

```
<requestID> BAL <balance> TIME <starttime> <endtime>
```

The values starttime and endtime are the system timestamps at the start of the request (recorded in the main thread as soon as the request is received), and at the end of the request (immediately before printing the results to the file). The timestamps should have a microsecond-level precision. Below is an example using the function `gettimeofday()` to get the system time and print it in the required format.

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

3.2.2 Transaction

A request for a transaction takes the format:

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

There may be from 1 to 10 accounts in any single transaction, with accounts occurring no more than once in any given transaction. The amounts should 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 balance of that account.

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

```
<requestID> OK TIME <starttime> <endtime>
```

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> TIME <starttime> <endtime>
```

where acctid is the account that had insufficient funds (there may be several; you only need to identify one). See the description of balance checks for information on starttime and endtime.

3.2.3 Program Exit

A request to halt the server program takes the format:

```
END
```

After this request, no further user requests are allowed. All currently queued requests 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.

3.2.4 Invalid Input

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

4 Hints

4.1 Synchronizing Account Access

Since the accounts may be needed by multiple threads at once, you should ensure that concurrent access to the same account is prevented. One way to protect the accounts from simultaneous access is to use a mutex for each account.

For transaction requests with more than one account, the worker thread must hold the mutex for each of these accounts before proceeding with the transaction. This behavior can easily lead to deadlocks unless the program is written carefully. One possible solution is to lock the mutexes in a sorted manner, i.e., always lock the account with the smallest ID first, then the next smallest, etc. If desired, you may use qsort() from the C standard library to sort the accounts.

4.2 Buffering User Requests

To ensure the main thread does not block when requests are coming in faster than they are being processed, your program should maintain a queue-like structure to hold requests that have been received but are not yet being processed. You can implement the queue as a linked list, with the main thread adding new requests to the tail of the list, and worker threads taking requests from the head of the list. Since the queue will be accessed by multiple threads, it should be protected with a mutex.

Below is some sample skeleton code that could be used as the basis of a request queue linked list. You are free to implement the data structures in other ways, if you would prefer.

```
struct transfer {    // structure for a transfer pair within a transaction
    int acc_id;      // account ID
    int amount;      // amount to be added, could be positive or negative
};

struct request {
    struct request * next;    // pointer to the next request in the list
    int request_id;           // request ID assigned by the main thread
    int check_acc_id;         // account ID for a CHECK request
    struct transfer * transfers; // array of transfers if a TRANS request
    int num_transfers;           // number of transfers if a TRANS request
    struct timeval starttime, endtime; // starttime and endtime for TIME
};
```

```
struct queue {
    struct request * head, * tail; // head and tail of the list
    int num_jobs;                  // number of jobs currently in queue
};
```

When there are no requests in the linked list, a worker thread must wait until there is one. You should use `pthread_cond_wait()` and `pthread_cond_broadcast()` to implement this waiting.

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. If you have issues with some of the output not being written to the file (e.g. the last few lines), try adding a sleep(1) just before closing the file and exiting the program.

4.4 Test Script

A test program Project2Test.c is included for testing. Compile it and run

```
./Project2Test
```

to see its usage.

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

```
./Project2Test ./appserver 10 1000
```

5 Input/Output Example

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
```

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 in the same order 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 for an account should yield the final balance in that account.

6 Part II - Locking Granularity Exploration

In this section you will explore a tradeoff 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, an implementation of coarse-grained locking.

6.1 Coarse-Grained Locking

Create a copy of your project. Modify the new version to lock the entire bank, with a single mutex, 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 run-time of both of your solutions (fine-grained and coarse-grained). For consistency, use the provided test script with the below parameters.

```
./Project2Test ./appserver 10 1000
./Project2Test ./appserver-coarse 10 1000
```

Record the average wait time per request for TRANS and CHECK requests, from the script output, in the project report.

6.3 Summary

Answer the following questions in the report template.

- 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?
- Discuss the probable "optimal" locking granularity (fine, coarse, or medium)?