Operating Systems Structure – 046209

Wet Assignment #2

Building a Parallel Bank Interface

Introduction

Note – Code originality will be checked. Copying homework is a disciplinary offense at the Technion with all its implications.

In this assignment, we will implement a bank that holds accounts and several ATMs through which operations can be performed on accounts (withdrawal, deposit, transfer between accounts, etc.). The ATMs will perform operations in parallel, where the goal of the assignment is to write a parallel program with threads using pthreads. In this assignment, you may not use any other library related to parallel programming and/or synchronization mechanisms (for example – <thread>, <semaphore>, <mutex> in C++ are forbidden).

Some tips:

1. Before asking questions on the forum – make an effort to verify that the answer to your question is not written in the assignment instructions/was already asked on the forum/in the course group. Special emphasis for this assignment – questions like "Is it okay that I locked X while Y..." are part of the required implementation in the assignment.
2. In the next part of the document you will find a general description of the assignment, followed by precise instructions. First read the general description to understand the structure of the program, and then the precise instructions.
3. **Plan first!** Draw a general flowchart of the program. What are the required data structures? How will we implement them? Define declarations of structs/classes, plan functions you want to implement, and only then start writing. Once you commit to a certain implementation that doesn't account for one of the requirements, it's much harder to change.
4. Try to identify each part of the assignment as a problem we discussed in the course with a known solution, and try to think how the problems integrate and affect each other for an overall solution. For example, many situations you'll encounter in the assignment can be modeled as the readers-writers problem – it's worth writing a generic readers-writers mechanism and reusing it.
5. The assignment has no skeleton – plan wisely how you handle basic requirements of the assignment like argument parsing and variable initialization.
6. Learn to use gdb – debugging parallel programs can be very complex, and printing to screen usually doesn't provide an adequate solution. In the MAMT course there's a workshop with everything you need to know about gdb, and with some internet searching you can display exactly which thread is executing which code section, what the state of its variables is, etc.

Good luck!

Program Description, User Interface and Execution Process

Data Structures:

The following data structures will appear in the program:

1. **Bank Account** – Each account will store an account number, password, and current balance.
2. **Bank** – The bank will contain all existing accounts.
3. **ATMs** – The ATMs will operate on the accounts in the bank.

Program Initialization:

The program will receive as input some number of paths to input files and will initialize that number of ATMs, where each ATM will receive its own private input file. Each input file will contain a sequence of commands for the ATM to execute, where the command format will be detailed later. For example, if the input to the program is 3 files, 3 ATMs should be initialized, where ATM 1 reads and executes commands from file 1, ATM 2 reads and executes commands from file 2, ATM 3 reads and executes commands from file 3, and so on. The file number will be the ATM's serial number – file number 1 corresponds to ATM number 1, etc.

Program Termination:

The program will terminate when all ATMs have finished executing all commands, or on an unrecoverable error (details later).

ATM Operations and Activity:

Every operation performed by an ATM must be recorded in a log file detailed later.

Operations:

The following operations must be supported in ATMs: 1. Open account | 2. Close account | 3. Deposit to account | 4. Withdraw from account | 5. Balance inquiry | 6. Transfer money between two accounts | 7. Close ATM | 8. Restore bank to previous status | 9. Currency exchange | 10. Stock market investment | 11. ATM on break

Bank Activity:

The bank charges a periodic commission which is a random percentage from all existing accounts. Every 30 milliseconds, the bank must charge all accounts 1%-5% of the current balance of each account in each currency the account holds. The bank must maintain its own account which is not accessible through the ATMs, where the money collected through commissions will be stored. The bank also prints to the screen the status of all accounts periodically.

Program Call:

The program will be called as follows from the command line:

```
./bank <number of VIP threads> <ATM input file 1> <ATM input file 2> …
```

The variable number of VIP threads will be detailed later in the VIP commands section. You can assume that VIP threads is a non-negative integer and that all file paths are valid; there's no need to perform input validation.

Implementation Details

Command Implementation:

Implement the commands as described below. Each command must produce an appropriate print to the log file according to the command type and instructions below. The command format is the form in which the command appears in the command files of each ATM. Every object that appears in angle brackets like <object> describes some variable that should be replaced with the appropriate value (in prints, commands, etc.). Do not print the angle brackets.

Log File:

You must create the log file yourself and its name must be log.txt, in the directory where the bank's executable file is located. You can use Linux's built-in system calls for file handling (open, close, write, etc.) or C++ built-in classes that facilitate file handling like ofstream. The writing to the file must be valid – that is, you must implement a writers mechanism on the file to ensure that each thread writes its entire line completely and only then another thread enters.

Historical Note:

The days are mid-May 2002, and therefore the exchange rate is 1 dollar equals 5 shekels (the actual figure was 4.9, rounded for convenience), surprising we know.

Command Descriptions:

1. Open Account: O <account> <password> <initial amount ILS> <initial amount USD>

a. If an account with the same number exists, write an error message to the log:

```
Error <ATM ID>: Your transaction failed – account with the same id exists
```

b. If the account was opened successfully, write to the log:

```
<ATM ID>: New account id is <id> with password <password> and initial balance <balance
ILS> ILS and <balance USD> USD
```

2. Deposit: D <account> <password> <amount> <currency>

a. If the password is incorrect, write an error message to the log:

```
Error <ATM ID>: Your transaction failed – password for account id <id> is incorrect
```

b. If the deposit was successful, write to the log:

```
<ATM ID>: Account <id> new balance is <balance ILS> ILS and <balance USD> USD after
<amount> <currency> was deposited
```

3. Withdrawal: W <account> <password> <amount> <currency>

a. If the password is incorrect, write an error message to the log:

```
Error <ATM ID>: Your transaction failed – password for account id <id> is incorrect
```

b. If the withdrawal amount is greater than the balance, write an error message to the log:

```
Error <ATM ID>: Your transaction failed – account id <id> balance is <balance ILS> ILS and
<balance USD> USD is lower than <amount> <currency>
```

c. If the withdrawal was successful, write to the log:

```
<ATM ID>: Account <id> new balance is <balance ILS> ILS and <balance USD> USD after
<amount> <currency> was withdrawn
```

4. Balance Inquiry: B <account> <password>

a. If the password is incorrect, write an error message to the log:

```
Error <ATM ID>: Your transaction failed – password for account id <id> is incorrect
```

b. If the balance inquiry was successful, write to the log:

```
<ATM ID>: Account <id> balance is <balance ILS> ILS and <balance USD> USD
```

5. Close Account: Q <account> <password>

a. If the password is incorrect, write an error message to the log:

```
Error <ATM ID>: Your transaction failed – password for account id <id> is incorrect
```

b. If the password is correct, the account will be deleted from the bank and write to the log:

```
<ATM ID>: Account <id> is now closed. Balance was <balance ILS> ILS and <balance USD> USD
```

6. Transfer Between Accounts: T <source account> <password> <target account> <amount> <currency>

a. After performing the transfer, the balance in the source account will decrease by amount, and accordingly the balance in the target account will increase.

b. You can assume that the source account and target account will not be identical.

c. If the password is incorrect, write an error message to the log:

```
Error <ATM ID>: Your transaction failed – password for account id <id> is incorrect
```

d. If the withdrawal amount is greater than the balance, write an error message to the log:

```
Error <ATM ID>: Your transaction failed – balance of account id <id> is lower than
<amount> <currency>
```

e. If the transfer was successful, write to the log:

```
<ATM ID>: Transfer <amount> <currency> from account <source account> to account <target
account> new account balance is <source balance ILS> ILS and <source balance USD> USD new
target account balance is <target balance ILS> ILS and <target balance USD> USD
```

7. Close ATM: C <target ATM ID>

a. Any ATM can instruct any other ATM to terminate its execution.

b. The ATM will request from the bank to close the ATM identified with ATM ID. In every print of account status (detailed later), the bank will check if there are requests to close ATMs, and if so will execute them.

c. An ATM that the bank instructed to terminate will finish the command it is currently executing and then release all its resources and will not execute additional commands.

d. If ATM ID does not identify an ATM (i.e., if ATM ID > argc), write an error message to the log:

```
Error <source ATM ID>: Your transaction failed – ATM ID <ATM ID> does not exist
```

e. If the closure was successful, the bank should write to the log file:

```
Bank: ATM <source ATM ID> closed <target ATM ID> successfully
```

f. If the closure failed because the ATM is already closed, write an error message to the log:

```
Error <source ATM ID>: Your close operation failed – ATM ID <ATM ID> is already in a
closed state
```

8. Restore: R <iterations>

a. To handle server crash cases, the bank will support an option to perform a rollback to a previous status. As you will see later, the bank must print its status every 10 milliseconds – the bank will allow, via an ATM command, to return the bank to the exact status it was in at any of these prints.

b. Returning the bank to a previous status will restore the balance of all accounts to their value, close accounts that weren't opened yet, open accounts that were closed, etc. This can be implemented in any way that preserves correctness and maintains maximum parallelism.

c. The bank must save its previous state for one second back – since a status print is performed every 10 milliseconds, about 100 statuses of the bank must be "remembered" backwards.

d. You can assume that a restore request for more than 100 statuses will not be received (i.e., the R argument is less than 100). You can also assume that R will be a positive integer, i.e., $100 \geq R > 0$.

e. The iterations parameter will count backwards to the desired status – for example, R 1 will return the bank one status back, and R 32 will return the bank 32 statuses back.

f. After the restore is performed, write to the log:

```
<ATM ID>: Rollback to <iterations> bank iterations ago was completed successfully
```

In all operations, if there is an attempt to access an account that doesn't exist (or was closed), print an error message to the log:

```
Error <ATM ID>: Your transaction failed – account id <id> does not exist
```

In the case of a transfer command from account to account, first check that the source account exists (and if it doesn't exist, stop the operation and print an error), and then that the target account exists (and if it doesn't exist, stop the operation and return an error).

9. Currency Exchange: X <account> <password> <source currency> to <target currency> <amount in source>

a. If the password is incorrect, write an error message to the log:

```
Error <ATM ID>: Your transaction failed – password for account id <id> is incorrect
```

b. If the exchange amount is greater than the balance in the currency, write an error message to the log:

```
Error <ATM ID>: Your transaction failed – account id <id> balance is <balance ILS> ILS and
<balance USD> USD is lower than <amount in source> <currency>
```

c. If the exchange was successful, write to the log:

```
<ATM ID>: Account <id> new balance is <balance ILS> ILS and <balance USD> USD after
<amount> <currency> was exchanged
```

10. Stock Market Investment: I <account> <password> <amount> <currency> <time in msec>

This operation was added to the system since the bank manager read the book "Investments for OS people". We believe it's important that account holders invest their money and not leave it all in checking accounts, so they can better care for their future.

The investment will be for a known period in advance with an interest rate of 3 percent per 10 milliseconds (we wish). The time parameter must be a multiple of 10. The calculation is performed according to compound interest: final amount = amount × 1.03^time

In this assignment, we don't implement the stock market system, so the invested money is not visible to us during various account operations. There's no need to update that the money was re-deposited to the account in real-time. At the end of the investment period, when the updated value returns to the system, we expect to see the updated amount both in the bank status print and in the various operations performed in the system.

11. ATM on Break: S <time_in_msec>

This operation was added to the system since the ATMs demand social rights, meaning they demand rest from time to time. The ATM that calls this command goes on a break (sleep) for a number of milliseconds. The value must be positive. This operation is a social right and therefore can never fail!

a. When performing the operation, print:

```
<ATM ID>: Currently on a scheduled break. Service will resume within <time_in_msec> ms.
```

VIP Commands:

1. To handle special customers who pay for a VIP account, the string "VIP=X" can be added to any command, where X is a number between 1 and 100. The string will appear at the end only with one space.
2. In this case, the ATM thread that receives the command will write it to a special data structure of commands.
3. The bank will create some number of special threads that are responsible for handling only requests from special customers, and they will execute commands from that queue without delay (i.e., without the sleep that ATM threads perform) and according to priority order determined by the constant X from high to low (i.e., the request of a customer with VIP=90 will be executed before a request of a customer with VIP=10, when both are in the queue in parallel).
4. The number of threads executing VIP commands will be given to the program as an argument in the command line as detailed in the program call section.
5. Threads executing VIP commands must perform operations identically to threads that are not VIP threads, except for the sleep that regular threads perform.
6. Threads executing VIP commands must write to the log identically to threads that are not VIP threads. There's no need to implement priority for VIP threads in writing to the file (this is a mechanism called priority lock).

Notes and Additional Instructions Regarding ATM Activity and Commands:

1. Each ATM must be implemented in a separate thread.
2. Account number is a number within int bounds and you can assume it doesn't start with 0.
3. Password is a number between 4 digits.
4. You can assume that when opening an account, a non-negative balance will be passed in both currencies.
5. You can assume that the input in the files is valid (i.e., there are no commands in a format different from what appears above and there's no need to handle errors in the command formats themselves).
6. Commands related to the same account must appear in the log in the order they were executed by the ATMs – i.e., if command A was executed before operation B on account 1, then operation A will appear in the log before operation B. However, there's no significance to the order of operations in the log between different accounts.
7. An operation is performed only in the currency listed in it (unless it involves 2 different currencies by definition). If an operation was performed on a certain currency, only it should be considered in execution.
8. When a bank account invests its money in the stock market, that amount is blocked and cannot be withdrawn or have operations performed on it, even if it means operations will be canceled in the system.

Example Input File:

```
O 12345 1234 100 0
W 12345 1234 50 ILS
D 12345 1234 12 ILS PERSISTENT
O 12346 0234 45 55
W 12345 0224 35 ILS
R 1
C 1
```

Bank Operations:

Every 30 milliseconds, the bank must charge all accounts 1%-5% of the current balance of each account. The specific percentage should be randomized (use standard mechanisms of the chosen language for generating random numbers). For each commission charge, print to the log file:

```
Bank: commissions of <commission percentage> % were charged, bank gained <commission ILS> ILS and <commission USD> USD from account <account id>
```

The bank must print to the screen (not to the log file) the status of all accounts every 10 milliseconds, to the upper left corner of the screen, in parallel with the bank's ongoing work. The order of accounts in the print should be by id, as follows:

```
Current Bank Status
Account <id1>: Balance - <balance1 ILS> ILS <balance1 USD> USD, Account Password - <password1>
Account <id2>: Balance - <balance2 ILS> ILS <balance2 USD> USD, Account Password - <password2>
…
```

And so on for all accounts, where id1 < id2. Additionally, after printing the status and in the following order:

1. The bank will check if requests to close ATMs have been received and if so will execute them.
2. The bank will check if requests to restore previous bank status have been received and if so will execute them. Note – the meaning is that the bank's status must be saved backwards and an option to restore it must be created.
3. Note that the bank only reads and doesn't write to accounts, so a readers-writers mechanism must be implemented on the accounts and they shouldn't be locked unnecessarily.

Special printf commands for your use:

```
printf("\033[2J") – clears the screen.
printf("\033[1;1H") – moves the cursor to the upper left corner of the screen.
```

This printing method creates a kind of animation of the bank status on the screen.

Notes Regarding Bank Activity:

1. The print must represent an accurate picture at that point in time of all accounts. For example, suppose there are 100 accounts in the bank with id between 1 and 100, and we started the print with account number 1 with a balance of $30. When we print account number 100, account number 1 should still have $30 (otherwise, the status picture of the print is not accurate at any point in time).
2. If any account is locked during printing, the print should be delayed until it's possible to read from the account (otherwise a violation of note number 1 is created).
3. The bank operates as long as there are commands in ATMs that haven't been executed.

General Notes:

1. Before implementing the parallel part, it's recommended to create a simple skeleton for the program that works with a single ATM and then expand the implementation to many ATMs.
2. All parallel aspects of the program (threads, synchronization mechanisms, etc.) must be implemented using pthreads. For clarity, it is forbidden to use objects such as std::thread, std::mutex, and submissions that use them will not receive points.
3. It's advisable to test the program on a wide range of number of ATMs, number of VIP threads, and number of operations in each file, from runs of single threads to tens of thousands of threads. It's recommended to write scripts in bash/python to generate input files and possible logs matching them of arbitrary length.
4. The main requirement in the assignment is that the program must be as parallel as possible – i.e., if you perform locks in unnecessary places, points will be deducted. Think for each operation you implement whether it requires parallel synchronization, and if so how to perform it with minimal blocking of other operations. Each entity in the program should be as independent as possible (hint – implemented in a separate thread) and interfere with other entities as little as possible.
5. The bank's operation mode with more than one ATM is inherently non-deterministic because it depends on the thread scheduling order which has no guarantee. Think about how to debug anyway (gdb with threads info), how correctness can be defined, and how to handle all the different requirements.
6. You can use int type variables for money and always round to the nearest int in case of non-integer fractions.
7. There's no situation where an account should go into negative, and if this does happen it indicates a significant bug – any withdrawal of an amount beyond the current balance should fail.
8. Do not use pthread_rwlock_t.
9. Make sure prints match 100% the requirements in the assignment – errors in prints may lead to point deductions.
10. Write in C or C++ only.
11. The forum is at your service for questions on any topic related to the assignment.

Error Handling:

If no parameters were passed to the program or one of the given files doesn't open for reading, print to stderr the following error and terminate the program:

```
Bank error: illegal arguments
```

If a system call fails and the arguments passed to it are valid, print an error message with perror() and terminate the program, in the following format:

```
Bank error: <syscall name> failed
```

Note: Failure of a system call in this wet assignment, unlike wet assignment 1, is a situation that shouldn't occur in a correct run and indicates a significant logical bug in the code – for example, pthread_mutex_lock can fail if the thread trying to lock the mutex already holds it, which indicates a need for code change and not some runtime handling.

Compilation and Linking

Make sure the code compiles using the following commands, for C++ and C respectively:

```
g++ -std=c++11 -g -Wall -Werror -pedantic-errors -DNDEBUG -pthread *.cpp -o bank
gcc -std=c99 -g -Wall -Werror -pedantic-errors -DNDEBUG -pthread *.c -o bank
```

The -Werror flag causes regular warnings in g++/gcc compiler to become errors. Make sure your code compiles on the course machine without warnings/errors.
Code that doesn't compile/compiles with warnings will not receive a grade.

You must provide a Makefile – the rules that must appear in it are:

1. A bank rule that will build the bank program
2. A rule for each separate file that exists in the project
3. A clean rule that deletes all compilation products

Make sure the program builds using the make command. The program must be compiled and linked with the flags shown above. It's recommended to separate the implementation into .cpp/.c and .h files to facilitate building the program – this should be addressed in the Makefile.
In addition to the above files, a readme file must be provided according to the format in the homework procedure.

A script is attached to the assignment that partially checks the submission validity (technically, not in terms of the assignment itself). The script takes two parameters – path to a zip file and the executable file name. For example:

```
./check_submission.py 123456789_987654321.zip bank
```

Good luck!