CSCI 3120 Assignment 1

Due date: 11:59pm, Sunday, May 23, 2022, submitted via Git

Objectives

This assignment has several objectives: (i) to refresh your coding abilities in C, and your programming skills in general; (ii) to review how to use and implement linked lists and queues, which are at the heart of every operating system; (iii) to serve as an example of the difficulty, style, and format of assignments that you can expect in this course; and (iv) to practice reading and implementing nontrivial specifications. This assignment is as much about reading and understanding this document, as it is about programming.

Preparation:

1. Complete Assignment 0 or ensure that the tools you would need to complete it are installed.
2. Clone your assignment repository:

   ```
   https://git.cs.dal.ca/courses/2022-summer/csci-3120/assignment-1/????.git
   ```

   where ???? is your CSID. Please see instructions in Assignment 0 and the tutorials on Brightspace if you are not sure how.

Inside the repository there is a **`miner`** directory, in which the code is to be written. There is also a **`tests`** directory that contains tests that will be executed each time you submit your code. Please do not modify the **`tests`** directory or the **`.gitlab-ci.yml`** file that is found in the root directory. Modifying these files may break the tests. These files will be replaced with originals when the assignments are graded. You are provided with a sample **`Makefile`** that can be used to build your program. If you are using CLion, a **`Makefile`** will be generated from the **`CMakeLists.txt`** file generated by CLion.

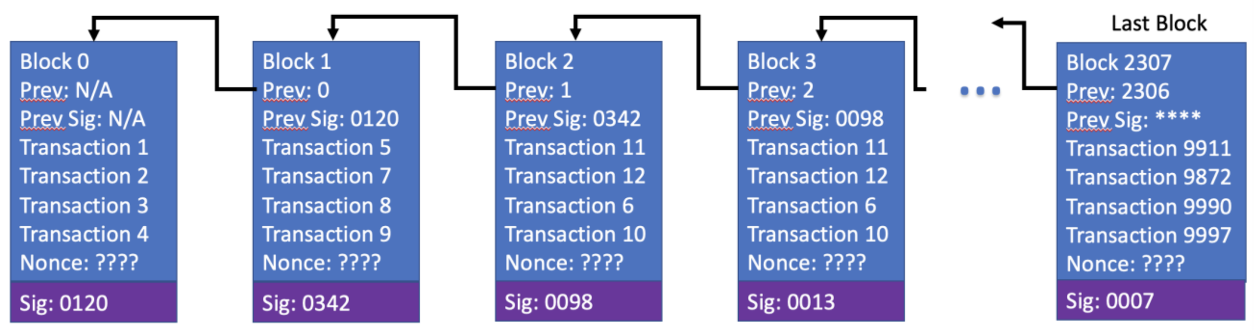
Background

BitCoin[1] is all the rage these days. BitCoin is a virtual currency that is managed by peer-to-peer network of computers instead of banks. To make a bitcoin transaction a digitally signed message is broadcast in the peer-to-peer network. Each transaction includes the

- **ID:** a unique identifier
- **payer**: the source of the funds (the transaction is digitally signed by the payer)
- **payee**: the receiver of the funds
- **amount:** the amount of the funds
- **fee:** the amount of bitcoin that will be paid for processing the transaction (by the payer)
- **signature**: of the payer to authenticate the transaction
- **Other information**

Transactions are processed by adding them to a digital ledger called a blockchain. A blockchain is simply a sequence of blocks where each block stores a series of transactions and has a reference and the signature of the previous block in the chain. Consequently, it makes it computationally infeasible to modify (forge) a block because all blocks succeeding it would also need to be modified. Thus, if Transaction 7 in Block 1 was to be modified, Block 1's signature would change, requiring Block 2 to be updated, requiring Block 3 to be updated, and so on. This is computationally infeasible because to generate a block (mining) requires a lot of computation.

[1] https://en.wikipedia.org/wiki/Bitcoin and https://bitcoin.org/en/how-it-works



Mining

One way to become rich is to *mine* blocks. The payment for mining a block is one bitcoin. As transactions are broadcast in the peer-to-peer network, they are aggregated by hosts in the network called *miners*. An aggregate of transactions that have not yet been added to a block is called a *mempool* (short for memory pool). To mine a block, a miner selects a set of transactions from the mempool and computes a new block. To compute a block the miner assembles the transactions in memory, along with the block's number, the previous block's number, the previous block's signature, and a *nonce*. The nonce is a 32-bit value that is determined by the miner such that the resulting signature of the block (when computed) starts with a certain number of 0s. The more 0s, the longer the search for the right nonce. The function used to compute the signature of a block (called a hash function) is expensive to compute. To find the right nonce the miner must try many different nonces, which takes a lot of computation. When a miner discovers the right nonce, it has "mined" a block. The miner broadcasts the block to the rest of the peer-to-peer network, and if the block is verified and accepted by the majority of the peers, the miner collects a bitcoin. The transactions in the new block are removed from the mempool and the miner proceeds to the next block. Note: the miner that mines the block first gets the reward. All others restart on the next block.

The goal is to mine blocks as efficiently and as quickly as possible. Mining takes computation, which requires computing hardware, electricity, cooling, and other supports that cost money. Miners must be fast because only the first miner who successfully mines the block gets the reward. Managing a computer in an efficient and fast manner is what good operating systems do. That is, designing a fast and efficient miner has similar challenges to designing a fast and efficient operating system.

Problem Statement

Write a program that simulates a bitcoin miner. Your program must be compiled to an executable called `miner`. The program will read in a sequence of four kinds of events (i) mine a block (**mine**), (ii) receive a transaction and add it to the mempool (**transaction**), (iii) receive a block that was mined by another miner (**block**), and (iv) end the simulation (**end**). For each of these events your program will need to perform a specific action as described below. These events correspond to actual events experienced by a real miner. For example, the **transaction** and **block** events correspond to the miner receiving transactions and blocks from the peer-to-peer network, and the **mine** event corresponds to the miner deciding to begin the creation of a new block.

On a **mine** event, your program must

1. select transactions from its *mempool* in the order that the transactions were added
2. create a new block from the selected transactions
3. remove the selected transactions from the *mempool*
4. print out the created block

On a **transaction** event, your program must

1. decode the transaction in the event
2. add the transaction to the *mempool*
3. print out the transaction

On a **block** event, your program must

1. decode the block. In the event
2. remove any transactions in the *mempool* that were listed in the block
3. print out the removed transactions in the order they were listed in the block

On an **end** event, your program must

1. exit

Note: the simulator will focus on the high-level operations. It will not need to perform any cryptographic functions or verify the correctness of a block or transaction.

Input

Your program will read its input from **stdin**. The input will consist of events in the format described below. You should just be able to use the `scanf()` function to read in all the input.

Mine Event

The format of the **mine** event is simply the word `MINE`, e.g.,

```
MINE
```

End Event

The format of the **mine** event is simply the word `END`, e.g.,

```
END
```

Transaction Event

The format of the **transaction** event word TRX followed by an encoding of a transaction, e.g.,

`TRX` ***transaction_encoding***

A ***transaction_encoding*** has the following format:

TID Payer Payee Amount Fee Source

where

TID: an unsigned integer denoting the block ID, e.g., `1123`
Payer: a single word (at most 31 characters), denoting the name of the payer, e.g., `Alice`
Payee: a single word (at most 31 characters), denoting the name of the payer, e.g., `Bob`
Amount: an unsigned integer denoting the transfer sum (in 1/1000000 of a bitcoin), e.g., the integer `1000000` is equal to 1 bitcoin.
Fee: an unsigned integer denoting the transfer fee (in 1/1000000 of a bitcoin), e.g., `100`

An example of a ***transaction_encoding*** would be:

1123	**Alice**	**Bob**	**150000**	**15**
TID	Payer	Payee	Amount	Fee

An example of a **transaction** event is:

```
TRX 1123 Alice Bob 150000 15
```

Block Event

The format of the **block** event word BLK followed by an encoding of a block, e.g.,

```
BLK block_encoding
```

A ***block_encoding*** has the following format:

```
ID PreviousID PreviousSig Transactions Signature
```

where

ID: an integer denoting the block ID, e.g., `42`

PreviousID: an integer denoting the block ID of the previous block, e.g., `41`

PreviousSig: an integer (in hexadecimal) denoting the signature of the previous block, e.g., `0x00debeef`

Transactions: An integer ***T*** denoting the number of transactions followed by ***T*** transactions each encoded as a ***transaction_encoding***, e.g.,

```
3
1123 Alice Bob 150000 15
1235 Dave Eve 100000 10
2358 Carol Gina 1000000 20
```

Nonce: the smallest unsigned integer (in hexadecimal) such that the computed signature has a 0 in the most significant byte.

Signature: an unsigned integer (in hexadecimal) denoting the signature of this block, e.g., `0x0cafe11a`

An example of a **block** event could look like

```
BLK 42 41 0x00debeef 3
1123 Alice Bob 150000 15
1235 Dave Eve 100000 10
2358 Carol Gina 1000000 20
0x00001011 0x00cafe1a
```

Note: the line breaks are optional. I.e., the entire block could have been encoded on one line. Again, using `scanf()` would be the easy way to decode this. Remember, `scanf()` will properly interpret integers written in hexadecimal if you use the "`%i`" format string for each hexadecimal being read.

Processing

When an **end** event occurs, the simulation should terminate. I.e., the program should free all its memory, close all files that it may have open, and exit.

When a **transaction** event occurs, add the transaction to the *mempool*. The *mempool* should be an ordered list of transactions in the order that they occur. You may assume that all transactions are correct and do not require verification. Using a linked list of structs to store the transactions would be an easy and efficient approach to managing transactions. **Note:** there is no limit to the number of transactions.

When a **block** event occurs, decode the block and remove any transactions in the block from the *mempool*. Transactions match if they have the same transaction ID (**TID**). Note, there may be transactions in the block that are not in the *mempool* and there will be transactions in the *mempool* that are not in the block. Only transactions that are in **both** should be removed from the *mempool*. There is no limit to the number of blocks.

When a **mine** event occurs, create a new block by determining the contents of the block (you do not actually need to assemble everything in one location):

- The **ID** of the block should be one more than the block **ID** of the most recent **block** or **mine** event.
- The **PreviousID** and **PreviousSig** are those of the block from the most recent **block** or **mine** event.
- If no **block** or **mine** event has occurred before this **mine** event, the **ID** is 1 and the **PreviousID** and **PreviousSig** are 0.
- The **Transactions** added to the block should be selected from the *mempool* in the same order that they were added to the *mempool*. I.e., the transaction at the head of the list is selected first.
- The number of transactions in a block is limited by its size, which is **256 bytes**:
  - Each integer is assumed to be 32 bits (4 bytes)
  - Each string takes space equal to the length of the string (at most 31 bytes) plus 1.

  For example, the block on page 4 is 90 bytes long: 15 integers of size 4 bytes plus six strings taking 6, 4, 5, 4, 6, and 5 bytes is equal to (15 x 4) + 6 + 4 + 5 + 4 + 6 + 5 = 90.
- The number of transactions selected for the block should be as many as possible without exceeding the block size limit.
- The **Nonce** for the block should be the smallest integer such that the most significant byte of the **Signature** is 0.
- The **Signature** should be computed using the provided signature generator (`siggen.h` and `siggen.c` in the starter code). Please see `siggen.h` on how to use.
- The **Signature** is computed on all parts of the block except the **Signature** itself. I.e., **ID, PreviousID, PreviousSig, Transactions,** and **Nonce.**
- Note: You will need to search for the right **Nonce** by selecting different values and computing the resulting **Signature** of the block.

After (or during) each of the events the program should perform output.

Output

All output should be performed to **stdout**. The output must be exactly as specified.

When a **transaction** event occurs, output the transaction after it is added to the *mempool*. The format is:

`Adding transaction:` ***transaction_encoding***

where the ***transaction_encoding*** is the same format as specified in the **Input** section. The encoding should be on a single line and terminated by new line.

When a **block** event occurs, output each transaction that is removed from the *mempool* in the order that the transactions are in the block. The format is:

`Removing transaction:` ***transaction_encoding***

where the ***transaction_encoding*** is the same format as specified in the **Input** section. The encoding should be on a single line and terminated by new line.

When a **mine** event occurs, output the newly created block. The format is:

`Block mined:` ***block_encoding***

where the ***block_encoding*** is the same format as specified in the **Input** section. The encoding should be in three parts. The first part should contain the **ID**, **PreviousID, PreviousSig,** and number of transactions on a single line. The second part should list the transactions in the block, one per line. The last part should contain the **Nonce** and the **Signature** on a single line. (See example in the **Input** section.) **Note:** use the **0x%8.8x** format string in printf()to display **PreviousSig**, **Nonce**, and **Signature** in hexadecimal.

Example

Input	Output
`TRX 11235 Alice Bob 150000 20`	`Adding transaction: 11235 Alice Bob 150000 20`
`TRX 12358 Carol Dave 350000 40`	`Adding transaction: 12358 Carol Dave 350000 40`
`BLK 1 0 0x00000000 1`	`Removing transaction: 11235 Alice Bob 150000 20`
`11235 Alice Bob 150000 20`	`Block mined: 2 1 0x0055a4cc 1`
`0x0000019a 0x0055a4cc`	`12358 Carol Dave 350000 40`
`MINE`	`0x000002ec 0x00b520fc`
`END`	

Grading

The assignment will be graded based on three criteria:
Functionality: “Does it work according to specifications?”. This is determined in an automated fashion by running your program on a number of inputs and ensuring that the outputs match the expected outputs. The score is determined based on the number of tests that your program passes. So, if your program passes t/T tests, you will receive that proportion of the marks.

Quality of Solution: “Is it a good solution?” This considers whether the approach and algorithm in your solution is correct. This is determined by visual inspection of the code. It is possible to get a good grade on this part even if you have bugs that cause your code to fail some of the tests.

Code Clarity: “Is it well written?” This considers whether the solution is properly formatted, well documented, and follows coding style guidelines. A single overall mark will be assigned for clarity. Please see the Style Guide in the Assignment section of the course in Brightspace.

If your program does not compile, it is considered non-functional and of extremely poor quality, meaning you will receive 0 for the solution.

The following grading scheme will be used:

Task	100%	80%	60%	40%	20%	0%
Functionality (20 marks)	Equal to the number of tests passed.					
Solution Quality (20 marks)	Solution is decomposed into appropriate files. Appropriate data structures (e.g. linked list) and language features (e.g., structs) are used.	Appropriate data structure (e.g., linked lists) and language features (e.g., structs) are used.	Appropriate language features (e.g., structs) are used.	Solution is workable but does not use appropriate data structures or language features	An attempt has been made.	No code submitted or code does not compile
Code Clarity (10 marks) Indentation, formatting, naming, comments	Code looks professional and follows all style guidelines	Code looks good and mostly follows style guidelines.	Code is mostly readable and mostly follows some of the style guidelines	Code is hard to read and follows few of the style guidelines	Code is illegible	

Assignment Submission

Submission and testing are done using Git, Gitlab, and Gitlab CI/CD. You can submit as many times as you wish, up to the deadline. Every time a submission occurs, functional tests are executed and you can view the results of the tests. To submit use the same procedure as Assignment 0.

Hints and Suggestions

- Start on this assignment early. The sample solution is under 200 lines of code, but it may take you a bit of time to understand the specification. It took me about 2-3 hours to write the code, so expect it to take at least 6 – 8 hours of work, depending on your proficiency in C.
- You will need to understand how to use `struct`s in C and how to implement a linked list. You will need to implement operations such as *append*, *find*, *remove*, and *remove_first*.
- The sample solution divides the code into three parts: (i) the signature generator (`siggen.c`), which is provided; a linked list implementation of *mempool* to store the transactions; and the main program (`main.c`).
- Please see `siggen.h` for a description of how to generate signatures.
- Use `scanf()` or `fscanf()` to read in the input. For example, to read in **unsigned_int string hex_int** from the input you can use the format string "`%u %s %i`". I.e., `scanf("%u %s %i", …);` where … are the locations into which the values should be read.
- Your program does **not** need to store blocks when they are received or created. I.e., use the information in a block as it is being read in and print out the block as it is being created. Hence, it does not need to be stored anywhere after it is created.
- Also, when computing the nonce, you do not need to recompute the signature for the entire block. Simply keep the signature for the block up to the nonce in variable, so you do not have to recompute it each time, because only the nonce changes during the search.