CSCI 3120 Assignment 2

Due date: 11:59pm, Sunday, June 12, 2022, submitted via Git

Objectives

This assignment has several objectives: (i) to reinforce your understanding of priority-based scheduling; (ii) to reinforce your understanding of how aging prevents starvation; and (iii) to provide additional practice in C programming and understanding specifications.

Preparation:

1. Complete Assignment 0 or ensure that the tools you would need to complete it are installed.
2. Review the Assignment 1 specification. Assignment 2 builds on Assignment 1
3. Complete Assignment 1 and/or review the provided solution to Assignment 1
4. Clone your assignment repository:
   `https://git.cs.dal.ca/courses/2022-summer/csci-3120/assignment-2/????.git`
   where ???? is your CSID.
5. Copy either your own solution or the provided solution as a starting point for Assignment 2 because Assignment 2 builds on Assignment 1. A solution will be accessible on May 27 from Brightspace.

The repository has the same structure and caveats as Assignment 1.

Background[1]

The goal of BitCoin[2] miners is to maximize their profit by earning as much BitCoin as possible. BitCoin is earned in two ways: (i) when a miner generates (mines) a new block for the block chain, and (ii) the collection of all the transaction fees from the transactions in the mined block. Hence, the miners will prioritize transactions that carry higher fees. Furthermore, since blocks are a fixed size, transactions that take less space are preferred as well because more transactions can be fit into a single block. For example, a single large transaction with a transaction fee of 150 is less profitable than two small transactions that are half the size with a fee of 100 each. Hence, just like an OS schedules processes to optimize various scheduling criteria, we would like our miner to schedule transactions to optimize its profit.

Problem Statement

Extend your program from Assignment 1 (or the provided solution) to simulate a miner that prioritizes high value transactions, packs blocks for maximal profit but, ensures that all transactions eventually get processed. Your program must be compiled to an executable called **`miner`**. The program will read in five events. The first four are the same events as in Assignment 1 and the fifth event, called **epoch** triggers an aging operation in the miner's transaction queue.

Please see the Problem Statement in Assignment 1 for a description of the first four events.

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

1. Update the priorities of transactions in the *mempool* as described in the Processing Section below.

Note: the changes to the simulator in Assignment 2 deal with how the transactions are selected for processing. All other functionality remains the same.

---

[1] This background description assumes that you have read and understood the background for Assignment 1.

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

Input

Your program will read its input from **stdin**. The input will consist of events in the format described below and in Assignment 1.

Mine, End, Transaction, and Block Events

Please see the Input section in Assignment 1 for the format description.

Epoch Event

The format of the **epoch** event is simply the word `EPOCH`, e.g.,

```
EPOCH
```

Processing

The **end** and **block** events are processed in the same way as in Assignment 1.

The **transaction** event is processed in the same way as in Assignment 1, except that the order of transactions in the *mempool* is priority-based as described in the Transaction Ordering section below.

The **mine** event is processed in the same way as in Assignment 1, except that the transactions are selected from the *mempool* on a priority basis as described in the Transaction Ordering section below. Transactions are selected until no more transactions can fit into the current block.

When an **epoch** event occurs an aging action should be performed on the *mempool.*

Transaction Ordering

In this simulation the *mempool* is implemented as a priority-based queue instead of an ordered-list. The priority-based queue should have 10 priority levels, numbered 0 through 9, where 0 is the lowest priority and 9 is the highest. When a transaction is inserted into the *mempool*, its priority is computed and it is inserted into the queue associated with that priority. The priority of a transaction is defined as:

$$Priority = \min\left(\left\lfloor \frac{Fee}{Encoding\ Size} \right\rfloor, 9\right)$$

where the *Fee* is the last integer in a transaction and the *Encoding Size* is the size of the transaction in bytes. That is, transactions that have a high fee to encoding size ratio are more profitable than transactions with a lower fee to encoding size ratio. Note: the encoding size ranges between 16 and 78 bytes. Priorities greater than 9 are considered priority 9.

Transaction selection from the *mempool* is based on two criteria: the priority of the transactions in the *mempool,* and the amount of space available in the block being mined. When a transaction is selected from the *mempool*, the selection function is passed the current available space in the block. The selection function should select the transaction from the highest priority queue in the *mempool*, that fits within the specified space. That is, you will need to search the queues from highest to lowest priority and select the first transaction that fits. If no transactions in the *mempool* are small enough to fit into the specified space, the block can then be completed.

When an **epoch** event occurs an aging operation should occur in the *mempool*, the first transaction in the 9 lower priority queues should be moved to the tail of the next priority queue. E.g., if there is a transaction at the front of priority queue 7, it should be moved to priority queue 8. In this way every transaction will eventually get processed.

Output

The output for this assignment is the same as that of Assignment 1, except when an **epoch** event occurs, the simulator should output each transaction that gets promoted in the *mempool*. The format is:

`Aging transaction (P) : transaction_encoding`

where the ***transaction_encoding*** is the same format as specified in Assignment 1 and P is the priority to which the transaction was promoted. The aging messages should be in order from highest to lowest priority. The encoding should be on a single line and terminated by new line.

Example

Input	Output

`TRX 11235 Alice Bob 150000 20`
`TRX 12358 Carol Dave 350000 40`
`EPOCH`
`MINE`
`END`
	`Adding transaction: 11235 Alice Bob 150000 20`
	`Adding transaction: 12358 Carol Dave 350000 40`
	`Aging transaction (2):12358 Carol Dave 350000 40`
	`Aging transaction (1):11235 Alice Bob 150000 20`
	`Block mined: 1 0 0x00000000 2`
	`12358 Carol Dave 350000 40`
	`11235 Alice Bob 150000 20`
	`0x000000da 0x00b75fc5`

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 1.

Hints and Suggestions

- Start on this assignment early. The sample solution is under 100 lines of additional code, but it may take you a bit of time to understand the specification.
- Most of the hints and suggestions in Assignment 1 still apply.
- The sample solution adds an additional layer between the main loop (`main.c`) and the transaction list (`transactions.c`). This layer (`mempool.c`) implements priority-based scheduling on top of the transaction list.
- Use an array of transaction lists to implement your priority-based queue.

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)	Proportionate to the fraction of tests passed.					
Solution Quality (20 marks)	Solution uses and implements correct algorithm in an efficient manner. Appropriate data structures are used.	Solution uses and implements correct algorithm. Appropriate data structures are used.	Solution uses and implements correct algorithm.	Solution implements a priority algorithm but not correctly.	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	