


Further Clarification re Assessment and Implementation

The only code to be added by students would be additional code to be integrated within the Mine() routine of the fully commented working program provided (for pointers see below).


As the marking scheme below shows, by completing and documenting tasks 1-5, which is simply to run the programme provided and presenting a report on it on the same basis as the Coursework Support Notes, a student can obtain (65%), and by completing just one of the last three tasks (requiring a few lines of code to be added) ($35/3 = 11.66\%$) a student can obtain $65 + 11.66 = 76.66\%$ of the marks; by completing a second task (E.g. Multi-threading using an extra-Nonce for re-hashing to get the block with the right number of zeros responsive to difficulty number) a student can obtain $76.66 + 11.66 = 88.32$ and by implementing an adaptive difficulty algorithm (see hints about using an exponential decay function in  [5- Further Coursework Support](#) a student can obtain up to 100% of the marks; or alternatively a student can suggest their own third task as they wish as long as not trivial.


Part	Evidence of implementation	Report	Total
1 – Project Setup	3	2	5
2 – Blocks and the Blockchain	5	5	10
3 – Transactions and Digital Signatures	5	5	10
4 – Consensus Algorithms (Proof of Work)	10	10	20
5 - Validation	10	10	20
6 – Assignment Tasks 1, 2, 3 Equally divided marks	15	20	35
	48	52	(100)


Coursework Facts

Tasks 1-5 which cover the stages of building the mini-blockchain	65%
---	------------


No programming is necessary, but it is essential to show a comprehensive understanding of the code provided and the mini-blockchain build scenario. Students are to submit a description of each of the steps including screenshots from their own running instance of the blockchain application running on their own computer to demonstrate their understanding. Thus, students will get full marks allocated to these steps (Yellow Parts = 65%) simply by showing their own screenshots arising from the execution of the code provided and outlining the steps for example illustrated by a flow chart.


All the code, including the constructors for each necessary object such as block, wallet and transaction, are integrated within the working code including the GUI using Windows Forms as provided in the practical folder to build a mini-blockchain. The Coursework requires that students refer to the document in the **Blackboard → CS3BC20 Blockchain → Practical folder**  [1- Practical Exercises and Coursework Support](#) which is the cookbook used for step-by-step creation of the mini-blockchain during laboratory sessions. This is to help the students, for the coursework to create their own version of the mini-blockchain on their own computer using the fully commented code that is also in the **Blackboard → CS3BC20 Blockchain → Practical folder**

 [2- BlockchainAssignment -fully commented code](#)

 [3- Practical_Exercises_Implementation_Environment](#)

Some tutorial advice regarding the only additional coding necessary for implementing Part B tasks has been provided in lab/lecture sessions and is also set out in the following two documents available on Blackboard:

 [4- Assignment Starter - notes re the final assignment tasks](#)

 [5- Further Coursework Support](#)

For Tasks 1, 2, 3 of Part 6, students can add some code of their own to the **Mine () routine** within the working code to experiment with alternative scenarios for Proof-of-Work which is the process of mining valid blocks as already implemented within the code.


The points at which this additional code could be introduced is dependent on the Tasks selected for Part 6:


- i) **Task 1:** Multi-threading to carry out, in parallel, the block preparation (Proof-of-Work) process for different degrees of difficulty. Finding out as you increase the degree of difficulty, how high one needs to go, before multi-threading would give significant speed advantage in arriving at a valid block with the right number of leading zeros, noting that multi-threading requires a computational overhead in thread management, and such parallelised computation will lead to an efficiency gain only when the gain from the parallelism is over-compensates for the time taken for thread management.
- ii) **Task 2:** Decision by an algorithm which periodically adjusts the “degree-of-difficulty” consistent with the Block Update Rate set for the next cycle – the time interval between successive block updates that has to allow enough time for the miners to arrive at a block hash with as many leading zeros as the number stated as the “degree of difficulty”.
- iii) **Task 3:** Decision by the miner as to which transactions to select for inclusion in the next block as typically there could be more transactions in the pool than a miner could/would include in each next block to be submitted.

Part 6 - (Tasks 1, 2, 3) Preparatory steps

So far students will each have created their own instance of the mini blockchain by running the code provided and interacting with the window forms produced to be able to show the objects created (Blocks, Transactions, and Wallets) so that they can use this to run 2/3 experiments/Tasks.


Students can now simply create a few new transactions by sending coins from one wallet to another, essentially establishing the mini-blockchain as a ledger – creating a number of accounts (wallets) and say 10-20 transactions within an hour time-period with varying amounts and fees so that the transactions will have sufficiently spread-out timestamps. Students would then be ready to attempt the additional Tasks for the last 35 marks; here are some more hints building on the advice given in the lab-lecture sessions and in the two following documents:


 [4- Assignment Starter - notes re the final assignment tasks](#)


 [5- Further Coursework Support](#)

This is simply a multi-threading exercise. Multi-threading tutorials are available for C# which is the language of the already completed program for the mini-blockchain build but for students who wish to translate the existing program to another language such as Python or Java, multi-threading tutorials have also been included for these languages with lots of code examples by way of additional support. Despite this, the additional code to be written for the additional Tasks is relatively minor, so it would probably be advisable to attempt to continue with C# even if one is more comfortable with some other language.

BB → CS3BC20 → Practicals →

 C# Threads Tutorial

 Java Thread tutorial

 Everything Python Headon -Coursework Support

For this task one can run two or more threads each doing the block validation/mining (Proof-of-Work) process in parallel. First thread would use a standard Nonce and subsequent threads using extra-Nonces, e-Nonce, as a Nonce can only be used with one of the threads, so as to avoid duplication of computation with each new thread instead of advancing the chance of producing the valid block faster. Thread synchronisation is also vital to ensure the first successful Mine() thread is recognised as the final result and all threads are timed to enable benchmarking the efficiency of single versus multi-threaded Mine() which is what the objective of this task.

So, one would need to measure how long it takes with single or more threads and how many threads to satisfy a workable level of Degree-of-Difficulty for a reasonable Block Update Time (time-to -compute-valid-block-Ready).

This just means using the existing code and adding some extensions using Mine() to work out the time each thread takes to do its Mine(). This would mean setting up new variables for thread management and latency registering (could make use of `system.Diagnostics.Stopwatch`).

One should be able to take multiple samples of mining times and compare them for different number of threads and for a few increasing "Degree-of-Difficulty" numbers and seeing for what Degree-of-Difficulty and what number of threads a most efficient [(shortest Mine() time)] is achievable.

So one can keep increasing the difficulty level to examine the point at which overheads of threading are exceeded by the efficiency gain arising from the parallelism provided by the threading. Say starting from four as the "Degree-of-Difficulty" used in the available working program and manually re-setting it for each run with a different number of thread (while increasing or decreasing the number of threads and timing the process to establish at what Degree-of-Difficulty, using how many threads, the Mine() will execute faster to deliver the valid block with requisite number of leading zeros, i.e. the optimal set-up at which the parallel execution of Mine() through multi-threading gets the job done fastest.

Part 6 - Task 2

35/3≈11.67 marks

Clearly there is a trade-off between changing the Degree-of-Difficulty and time-to-submit of valid blocks which is the time it takes to complete the computation required to arrive at a submittable block with the requisite number of leading zeros that thus satisfies the "Degree-of-Difficulty".

The block update time (latency) usually does not change frequently for a specific Blockchains (e.g. In Ethereum the block update time 10-20 seconds, for Bitcoin it is presently 10 minutes and the Degree-of-Difficulty maintained with this is 19, see below).

Example of valid hash of a block as required to be computed under Proof-of-Work

000000000000000000000000578b16c88ffc7a2b59c8489b7cc442c5a8129c7a79c4b

(This string consists of the 19 leading zeros and the hash of data inside the block and the Nonce for this round (Number-used-only-once)

This block hash has 19 leading zeros and is in hexadecimal format, The chance of obtaining the above string for PoW is $1/16^{19}$.

10 thousand trillion hashes need, starting with the initial hash of the block with the Nonce, for the chance to obtain such a string as a block to be submitted for validation.

$$\text{Block time} = \text{timestamp for block } (n) - \text{timestamp for block } (n - 1)$$

For setting the “Degree-of-Difficulty” to vary the computational load of Dynamic Proof-of-Work for miners from time to time, one can assess the trend in the median value of Block Update Time and have an algorithm that dynamically increases the Degree-of-Difficulty responsive to the trend in the median value of the Block Update Time.

$$\text{Block time} = \text{timestamp for block } (n) - \text{timestamp for block } (n - 1)$$

In this way, potentially, Blockchains could use the block update times of the last N blocks to re-adjust the Degree-of-Difficulty responsive to the trend in the value of the Block Update Time.

One can use an exponential function to have the Degree-of-Difficulty be re-set responsive to changes in the Block Update Time (median time it has taken for the miners to submit the last N blocks) or use alternative approaches for such Adaptive Proof-of-Work and decide which would be most effective.

However, the starting point could be to establish a target block time and apply an exponential decay function to converge to a “Degree-of-Difficulty” number incrementally best adjusted for the target block update latency desired (for further information please see [PDF 5- Further Coursework Support](#)).

Using a bit of code to modify the way **Mine()** works, a student can allow the selection of transactions from the transactions pool to take place based on:

- Greedy (choose transactions with the higher fees with a cut-off of say six transactions to chosen)
- Altruistic (check the timestamp of transactions and chose the oldest ones with cut-off on how old)
- Random (run a Rand() function with a small range and choose the transaction which have most matches with say the transaction number or the hash of some parameter of the transaction - with a cut-off on the transactions)
- Address-based (Choosing transactions based on the value of some other attribute of the transaction e.g., the highest wallet_ID addresses with a cut off after the top six chosen)

Start by using the Windows Form to create buttons for each of the above four types of Mining. Declare custom functions to select transactions from the transaction pool, according to each of the above criteria, thereafter, proceeding to run Mine() with this method to select the transactions to be selected for the new block generated – ready to be submitted.

Basic Technical Reporting Hygiene

As you can see 52% of the marks are allocated to the reporting so it is important that the accompanying report submitted is complete, coherent, and well-presented as it with the presentation that one can needlessly lose marks one could easily keep.

Good luck.
