Operating Systems 2024/2025

Project

DEIChain: A Concurrency-Focused Blockchain Simulation

NOTE: Before starting to code, read the entire assignment!

1. Introduction

Welcome to **DEIChain**, a hands-on project that explores the world of *Blockchain* technology and its fundamental principles. While real-world blockchains like Bitcoin and Ethereum support cryptocurrencies and decentralized applications, they also present complex challenges related to process management, Inter-Process Communication (IPC), concurrency, and synchronization. In this project, you will create a simplified blockchain simulation, **DEIChain**, to examine these essential operating system concepts. Before proceeding, please take three (3) minutes to watch this video: https://www.youtube.com/watch?v=0B3sccDYwuI to gain a better understanding of the scenario explored in this assignment.

Blockchain technology has garnered considerable attention in recent years, mainly as the underlying infrastructure for various cryptocurrencies. Essentially, a *Blockchain* is a replicated state machine that operates on a distributed ledger. The system records transactions, which are validated and linked in chronological order.

For this simulation, imagine a digital ledger where transactions are recorded in blocks that are linked together in chronological order and secured through cryptographic validation. This is the essence of *Blockchain* technology. Each block contains a set of transactions, a timestamp, and a cryptographic link to the previous block, creating an immutable chain. In the context of real-world cryptocurrencies, miners compete to solve mathematical puzzles, known as **Proof-of-Work (PoW)**, to add new blocks to the chain, thereby ensuring security and consensus.

In **DEIChain**, cryptographic complexity is simplified to focus on the core mechanisms that drive *Blockchains*. You will develop a system that simulates the creation, validation, and linking of blocks, focusing on the importance of concurrency and synchronization. This project will challenge you to design and implement efficient IPC, manage shared resources, and ensure data integrity in a concurrent environment. In a nutshell, the goal behind **DEIChain** is to simulate basic operations in a distributed ledger**,** which are: *Create*, *Validate*, and *Link* blocks.

2. Simulation description

This section describes the technical architecture (see Figure 1) of the system, including the functionalities of the various components of the simulator and how they communicate.

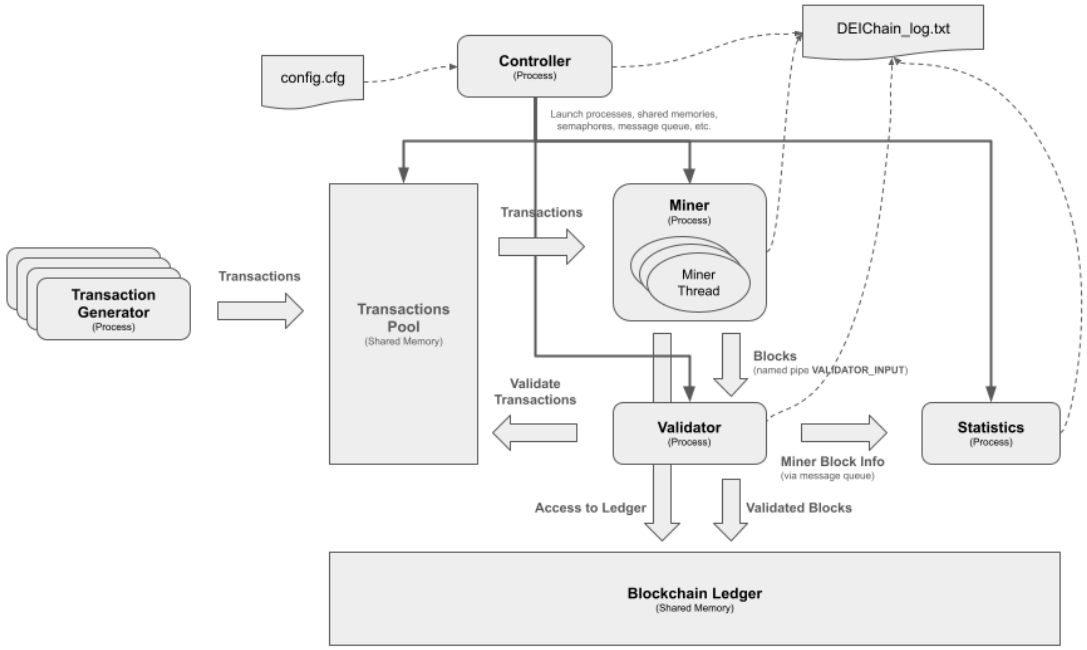

Figure 1. DEIChain modules description and interaction

2.1. Description of components

DEIChain will have the following processes and threads:

- **Controller.** Process responsible for reading the configuration file and initializing most of the components in the simulation environment.
- **Miner.** Process that operates with a pool of NUM_MINERS threads. Each **Miner thread** created will simulate an individual miner; it reads transactions, groups them into new blocks, and performs a simple **PoW** validation step. **Miner threads** can prioritize specific features when grouping transactions, such as the reward for successfully solving the **PoW** for each transaction. Each **Miner thread** has a specific identification number (ID).
- **Validator.** A process that validates blocks and manages the **Transactions Pool**. When a block is sent by a **Miner thread** to the **Validator**, the latter must check the validity of the block and if it can be inserted in the **Blockchain Ledger**. It is also responsible for an aging mechanism that tries to avoid starvation.
- **Statistics.** A process that calculates and prints statistics upon receiving a specific signal and at the end of the simulation.
- **Blockchain Ledger.** A data structure stored in shared memory that maintains a sequential list of validated blocks. The size of the *Blockchain* is specified in the configuration file read by the **Controller**.
- **Transaction Generator (TxGen).** Process executed by the user that produces *transactions* at specified intervals and writes them to a **Transaction Pool** (located in shared memory). Several **Transaction Generator** processes may be active at the same time. The **TxGen** adds a transaction to the **Transaction Pool** by traversing it sequentially, placing it in the first available spot, and making the aging field zero.

Some files will also be used:

- **Log File** ("`DEIChain_log.txt`"). A file that stores all the events produced by the **Controller** and all the processes it creates. All the relevant information about their activity will be logged for later analysis. All the information written in the log should also appear on the screen.
- **Configuration file** ("`config.cfg`"). The configurations for the simulation are read by the **Controller** and are stored in a configuration file that should be read at the beginning. It contains essential information such as the # of **Miner threads**, the size of the **Transaction Pool**, and the **Blockchain Ledger** size (see specific section).

Additional IPCs will be used to enable communication between processes/threads:

- **Transactions Pool**. Shared memory that holds the transactions produced by the **Transaction Generator** processes.
- **Blockchain Ledger**. Shared memory that holds all the blocks that constitute the blockchain.
- **Named Pipe** ("VALIDATOR_INPUT"). Written by the **Miner threads** and read by the **Validator**.
- **Message Queue**. Enables the transfer of information from process **Validator** to **Statistics**.

2.2. Description of operation

The fundamental workflow for the simulation is depicted in Figure 2.

System initialization

During the initialization stage, the **Controller** process reads the configuration file ("`config.cfg`"), creates all the necessary resources, and launches most of the key components. First, it sets up IPC mechanisms, such as shared memory segments, semaphores, message queues, and named pipes. The **Controller** also initiates the **Miner** process that creates NUM_MINERS threads, the **Validator** and **Statistics** processes. By the end of the initialization phase, the basic infrastructure is established, and each process has the resources it needs to communicate with others. All these operations are logged in the file "`DEIChain_log.txt`" and shown on the screen.

Operation

Once the system is running, it is ready to accept new transactions. To create transactions, one or more **TxGen** processes can be launched. They are created by the user, using the following syntax:

```
$> TxGen <reward> <sleep time>
```

reward: 1 to 3
sleep time (ms): 200 to 3000

Example:
```
$> ./TxGen 2 500
```

Each **TxGen** generates a new transaction at each sleep time period and sends it to the **Transactions Pool** (shared memory). Multiple **TxGen** may be active simultaneously. Students must design each generator to be lightweight, focusing on the production and transmission of transaction data.

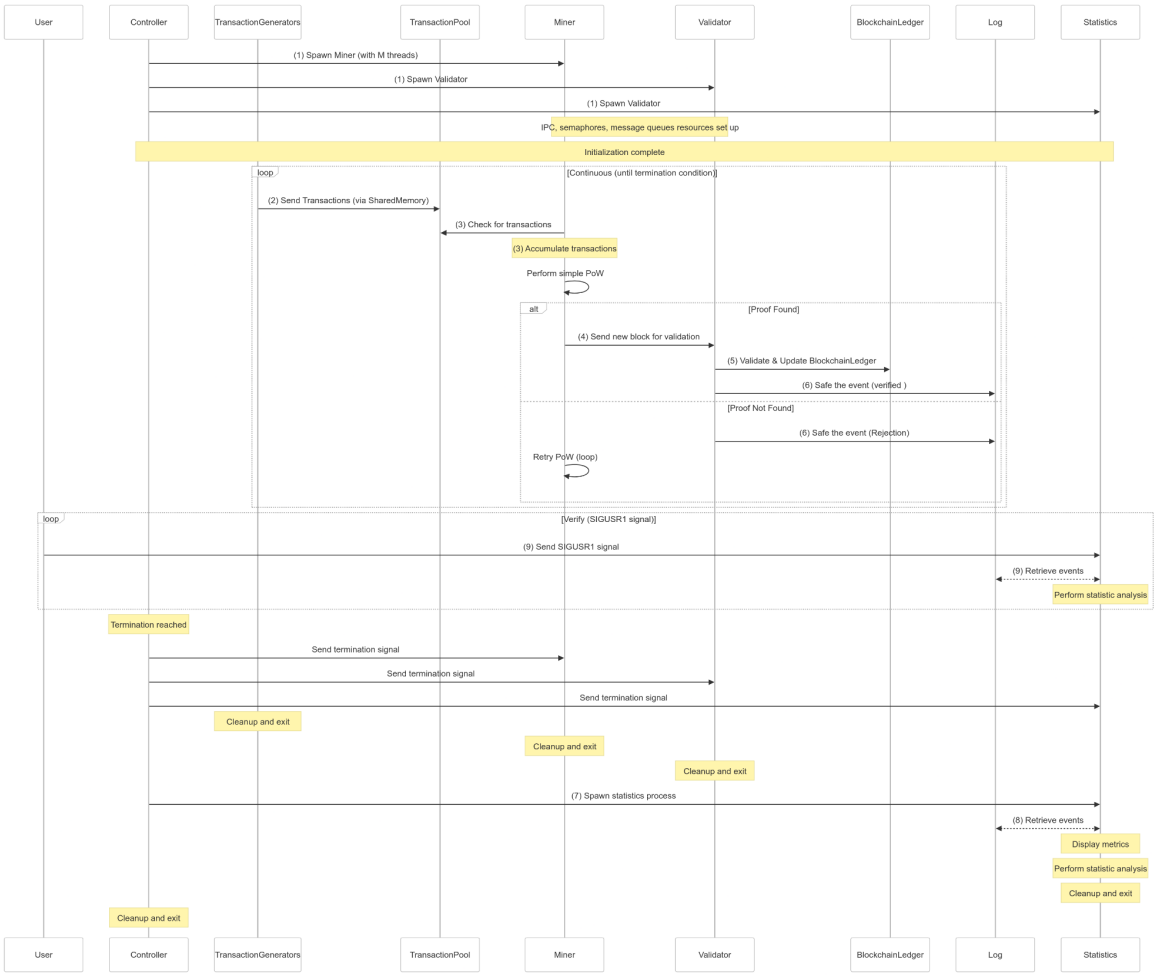

Figure 2. DEIChain workflow simulation

During the **Mining** phase, each **Miner's thread** selects TRANSACTIONS_PER_BLOCK transactions to put in a block. Various strategies can be used by each **Miner's thread** (e.g., it may randomly select transactions, focus on those with higher rewards to maximize earnings, or choose the simpler ones (i.e., with lower rewards) to improve throughput. After grouping the transactions, the **Miner** will solve a simplified **PoW** puzzle. **PoW** is a computational process to secure *Blockchains*, requiring miners to solve a cryptographic puzzle before adding a new block. This process hashes the data block with a numeric *nonce* (**n**umber used **once**) until a hash that starts with a given number of leading zeros (difficulty) is found. Once the **Miner** has found a valid *nonce*, it completes the block-building process by packaging the transactions into a new block, including the *nonce*, and marking the block as ready for validation (more info ahead).

After processing the block, the **Miner's thread** sends the newly created block to the **Validator** that will confirm (or not) the block's validity. The validation process rechecks the block's **PoW** (to verify the **Miner's thread** work), checks that it correctly references the latest accepted block in the **Blockchain Ledger**, typically by matching the *Previous block ID*, and also that the transactions in the block are still in the **Transactions Pool** (i.e., were not already processed and put in the **Blockchain Ledger**). If the block passes all the checks, the **Validator** updates the **Blockchain Ledger** data structure by appending the block to the chain. It also sends the **Miner's thread** *ID* and the credits earned (the sum of the rewards of each transaction) to the **Statistics** process. It must also remove the transactions in the new block from the **Transactions Pool** so no other **Miner's thread** uses them (it marks the slots as available). If the block fails validation (e.g., if its *nonce* does not meet the required puzzle condition or

the transactions are no longer valid in the **Transactions Pool**), the **Validator** rejects it and communicates it to the **Statistics** process. The communication between the **Validator** and **Statistics** process is done via a message queue. The **Validator** is also responsible for increasing the *age* of each transaction each time it touches the **Transactions Pool**. Whenever an increment of 50 is seen in the age of a given transaction, the **Validator** must increment (by one) the transaction's rewards. (i.e., `age mod 50 == 0 → reward++`).

At any time, the **Statistics** process can receive a *SIGUSR1* signal. Upon receiving a *SIGUSR1* signal, it will print all the statistics available to the console (see Statistics Metrics section).

Termination

The system terminates when one of the following stop conditions is met: the maximum size of the *Blockchain* is reached or a SIGINT is received by the **Controller**. Once this occurs, the system must shut down in a controlled manner. Each process should end its current task and clean up properly by releasing resources (e.g., detaching from shared memory, closing semaphores). In the end, the **Statistics** process writes all statistical information on both the screen and the log. The number of transactions not finished should also be written in the log. Once all processes have completed their shutdown procedures, the system will fully exit, releasing its resources and concluding the simulation.

2.3. Additional details

This section provides additional insights on the data structures, files, and on the essential functions of the solution.

Data structures

Some insights regarding the data structures to use:

- Blocks. In your *Blockchain* simulation, a block is a data structure that represents a single unit of data within the ledger.
    - **TBD**
- Transactions. In your *Blockchain* simulation, a transaction represents a discrete piece of data or an operation that needs to be recorded on the ledger (the *Blockchain*). This could involve transferring coins from one party to another in a real-world cryptocurrency context. In this case, a **Transaction Structure** is more straightforward, focusing on the basics of storing a small record of information that your processes can share, pass through a shared memory, and include within a block. The structure of a transaction is detailed below.
    - Transaction ID. A unique integer or string that identifies the transaction (PID of the **Transaction Generator** + incremental number). This is useful for logging and avoiding duplicate records if you replay or reprocess transactions.
    - Reward. Each transaction has an associated reward related to the complexity of completing its PoW. The reward is an integer value, originally from *1 to 3*, where a high value denotes a higher reward. Each valid block mined gives the **Miner** a number of credits corresponding to the sum of the rewards of all the transactions included in the block. The reward is a parameter of the **TxGen**. However, note that a reward greater than 3 can happen because of the aging control mechanism implemented on the **Validator**.

    - Sender and Receiver Fields. To simulate the movement of some resource (e.g., tokens, items, data) between entities, you must store a *sender_ID* and a *receiver_ID*. For example, *sender_id = 101* and *receiver_id = 202*. The sender_id must be the PID of the process, the receiver may be a random number.
    - Value. Represents the quantity associated with this transaction.
    - Timestamp. The time at which the transaction was created. Useful for debugging and for verifying when transactions occurred relative to the block's creation time.
- Transaction Pool: This structure is responsible for holding information on the transactions pending processing. It must have the following fields:
    - *current_block_id*: holds the value of the current block (Block ID).
    - *transactions_list*: Current transactions awaiting validation. Each entry on this list must have the following:
        - *empty*: field indicating whether the position is available or not
        - *age*: field that starts with zero and is incremented every time the Validator touches the Transaction Pool. The Transaction Pool size is defined by the configuration size.
- PoW: **TBD**
- Blockchain Ledger. **TBD**

Statistic Metrics

Some statistics to provide:

- Number of valid blocks submitted by each specific Miner to the Validator
- Number of invalid blocks submitted by each Miner to the Validator
- Average time to verify a transaction (since received until added to the *Blockchain*)
- Credits of each Miner (Miners earn credits for submitting valid blocks)
- Total number of blocks validated (both correct and incorrect)
- Total number of blocks in the *Blockchain*

Configuration file

The configuration file should have the following structure:

```
NUM_MINERS - number of Miners (number of threads in the Miner process)
POOL_SIZE - number of slots on the Transaction Pool
TRANSACTIONS_PER_BLOCK - number of transactions per block (will affect block size)
BLOCKCHAIN_BLOCKS - maximum number of blocks that can be saved in the Blockchain ledger
TRANSACTION_POOL_SIZE - the size of the transaction pool in items (default 10^4)
```

Example:

```
5
50
10
50000
```

Log file

All the application's output must be written in a readable form in the "`DEIChain_log.cfg`" text file. Each entry in this file must always be preceded by showing the same information in the console

so that it can be easily visualized while the simulation is running. You should log all relevant events with their date and time, including:

- Start and end of the program;
- Creation of each process and thread;
- Errors that occurred;
- Validated/invalidated blocks;
- Signals received

3. Checklist

This list only serves as an indication of the tasks to be carried out and marks the components that will be assessed in the mid-term defense. Tasks with '(preliminary)' do not need to be completed in the intermediate defense.

Item	Task	Evaluated in Intermediate defense?
Transaction Generator	Creation of Transaction Generators	S
	Correct reading of command line parameters	S
	Write the transactions in shared memory	
Controller	Booting the system, reading the configuration file, validating the data in the file, and applying the read configurations.	S
	Creation of processes Miner, Validator, and Statistics	S
	Creation of the shared memories (2 shared memories)	S
	Creation of the message queue	
	Create named pipe	
	Capture signal SIGINT, terminate the simulation, and release resources.	S (preliminary)
Miner	Create Miner threads according to the configuration specified in the configuration file.	S
	Get the transactions to mine	
	Process the block and assemble it	
	Send the block to the Validar using the named pipe	
Statistics	Generate statistics upon receiving the signal SIGUSR1	
	Write statistics before closing the simulation	
Log file	Synchronized sending of output to log file and screen.	S
All	Use a makefile	S
	Diagram with architecture and synchronization mechanisms	S (preliminary)
	Concurrency support in all simulation	
	Error detection and handling	
	Synchronization with suitable mechanisms (semaphores, mutexes or condition variables)	S (preliminary)
	Prevention of unwanted interruptions by unspecified signals; providing the appropriate response to the various signals specified in this assignment	
	Controlled termination of all processes and threads, and release of all resources.	

4. Important notes

- Please read the instructions carefully and ask your teachers for clarification.
- Instead of starting to write code straight away, take time to think about the problem and structure your solution appropriately. More efficient solutions that use fewer resources will be valued.
- Include in your solution the code needed to detect and correct errors.
- Avoid busy waiting, synchronize access to data whenever necessary, and ensure clean termination of the server, i.e. with all used resources being removed.

- **Penalties:**
    - Failure to compile the code sent, implies an evaluation of ZERO in the corresponding goal.
    - Busy waiting will be heavily penalized! Use the *top* command in a terminal to ensure that the program does not use more CPU than necessary!
    - Concurrent accesses without synchronization that may lead to data corruption will be heavily penalized!
    - The use of the *sleep* function or similar stratagems to avoid synchronization problems will be heavily penalized!
    - Include debug information to make it easier to monitor the execution of your code, for example using the following approach:

- Include debug information that makes it easier to monitor the program's execution, using an approach such as:

```
#define DEBUG //remove this line to remove debug messages
(...)
#ifdef DEBUG
printf("Creating shared memory\n");
#endif
```

- All work should run on the VM provided or, alternatively, on the student2.dei.uc.pt machine.
- Compilation: the program should be compiled using a makefile; there should be no errors in any of the project deliveries; warnings should also be avoided, except when you have a good justification for their occurrence (rare!).
- The final defense of the work is compulsory, and all group members must participate. Failure to attend the final defense will result in the assignment being graded ZERO.
- The work can be done in groups of up to 2 students (groups with only 1 student should be avoided and groups with more than 2 students are not allowed).
- The students in the group must belong to the same teacher's PL classes. Groups with students from different teachers' classes are exceptions that require prior approval.
- Each defense is individual, so each member of the group may have a different grade.
- Both intermediate and final defenses must be carried out in the same class and with the same teacher.
- Plagiarism, copying parts of code between groups or any other type of cheating will not be tolerated. Attempts to do so will result in a grade of ZERO and consequent failure in the course. Depending on the seriousness, this may also lead to disciplinary action.
- All work will be scrutinized for copies of code.
- To prevent copying, students may not place code in publicly accessible repositories.

- Students are advised to use AI tools in a responsible manner. Do not forget that project authors are responsible for all the code submitted and must be able to justify the structure of the program, the options taken, the advantages of the approach, and to explain all parts of the code. Failure to address any of these aspects of the work will be graded accordingly.

5. Project delivery

Date	**Meta**	
Delivery date on Inforestudante 31/03/2025-09h00	Intermediate delivery	Create an archive in **ZIP** format (**NO OTHER FORMATS WILL BE ACCEPTED**) with all the work files and submit it to Inforestudante: ● The **names** and **student numbers** of the group members must be placed at the beginning of the source code files. ● Include all the necessary source and configuration files as well as a Makefile for compiling the program. ● Do not include any files that are not necessary for compiling or running the program (e.g., directories or version control system files). ● With the code, 1 A4 page must be delivered with the architecture and all the synchronization mechanisms to be implemented described. ● **Email submissions will not be accepted.**
Week starting on 24/03/2025	Demonstration/ Intermediate defense	● The demonstration must cover all the points mentioned in the checklist in this assignment. ● The demonstration/defense will be carried out in PL classes. ● The intermediate defense is worth **20%** of the project grade.
Delivery date on Inforestudante 12/05/2025-09h00	Final delivery	Create an archive in **ZIP** format (**NO OTHER FORMATS WILL BE ACCEPTED**) with all the work files and submit it to Inforestudante: ● The **names** and **student numbers** of the group members must be placed at the beginning of the source code files. ● Include all the necessary source and configuration files as well as a Makefile for compiling the program. ● Do not include any files that are not necessary for compiling or running the program (e.g., directories or version control system files). ● A brief **report** (maximum 2 A4 pages) in **pdf** format (**NO OTHER FORMATS WILL BE ACCEPTED**) must be submitted with the code, explaining the choices made in building the solution. Include a schematic of your program's architecture. Also include information on the total time spent (by each of the two group members) on the project. ● **Email submissions will not be accepted.**
12/05/2025 a 04/06/2025	Final defense	● The final defense is worth **80%** of the project fee and will consist of a detailed analysis of the work presented. ● Group defenses in PL classes. ● Registration is required for the defense.

After submitting, it is advisable to download the files you have submitted to check that you have included everything necessary. If you submit the wrong folder, only part of the files, etc., they will not be evaluated.