**CS 3113 – Intro to Operating Systems – Spring 2025**
**Project One: Due February 4, 20205, 11:59 PM**

**Description**: This project is about simulating process management and execution. This project involves simulating a simplified process management system. Students will implement essential operating system concepts such as:

- Using a **Process Control Block (PCB)** to store and manage process metadata.
- Allocating and managing **main memory** for multiple processes.
- Simulating a **CPU** to execute instructions from processes.
- Implementing a **job queue** and a **ready queue** to manage job states and execution.

The simulation will read job details from an input file, load jobs into main memory, execute instructions, and manage memory allocation. The system will continue until all jobs have been executed.

**Input Description:** The program reads the input file via redirection, and the file must adhere to a specific format. The first line specifies the maximum size of main memory in integers. The second line indicates the total number of processes. Each subsequent line contains the details for an individual process. Each process description begins with a processID, which serves as a unique identifier for the process, followed by its initial state, which is always set to NEW. Next is the maxMemoryNeeded, representing the memory required for the process, and numInstructions, the number of instructions associated with the process. The instructions are encoded as integers. The supported instruction types are: Compute (encoded as 1 iterations cycles, e.g., 1 5 10), Print (encoded as 2 cycles, e.g., 2 3), Store (encoded as 3 value address, e.g., 3 42 250), and Load (encoded as 4 address, e.g., 4 250). Each instruction type specifies the operation and its associated parameters. This structured input format enables the program to parse and manage process information effectively.

Example Input and explanation:

```
2048
2
1 300 3 1 5 10 3 42 250 4 250
2 400 4 2 3 3 100 350 1 2 7 2 5
```

The above input indicates that the main memory has 2048 elements. An integer array of size 2048. The value 2 represents two process that needs to be executed. The instructions to be executed will be compute, store, load, and print. These will be discussed below. **Store and Load operation take one CPU Time.**

Process 1 has a processID of 1 and an initial state of NEW. It requires a maximum of 300 units of memory (int array of size 300) and includes 3 instructions. The instructions are as follows: 1 5 10, which represents a Compute operation with 5 iterations consuming 10 CPU cycles; 3 42 250, which represents a Store operation that saves the value 42 at address 250; and 4 250, which represents a Load operation that retrieves a value from address 250.

Process 2 has a processID of 2 and an initial state of NEW. It requires a maximum of 400 units of memory (int array of size 400) and includes 4 instructions. The instructions are as follows: 2 3, which represents a Print operation consuming 3 CPU cycles; 3 100 350, which represents a Store operation that saves the value 100 at address 350; 1 2 7, which represents a Compute operation with

2 iterations consuming 7 CPU cycles; and 2 5, which represents a Print operation consuming 5 CPU cycles.

**PCB Structure:** The Process Control Block (PCB) is a data structure that contains essential metadata for managing each process. It includes several fields that define the process's identity and operational details. The processID serves as a unique identifier for the process. The state field tracks the current status of the process, which can be one of the following: NEW, READY, RUNNING, or TERMINATED. The programCounter indicates the index of the next instruction to be executed within the process's logical memory. The instructionBase specifies the starting address of the instructions in the logical memory, while the dataBase points to the beginning of the data segment within the logical memory. The memoryLimit defines the total size of logical memory allocated to the process. The cpuCyclesUsed field accumulates the total CPU cycles consumed by the process during its execution. The registerValue is a simulated register used to store intermediate values during load and store operations. The maxMemoryNeeded specifies the maximum memory required by the process as defined in the input file. Finally, the mainMemoryBase denotes the starting address in main memory where the process, including its PCB and logical memory, is loaded. This structure ensures effective management and tracking of process execution within the system.

**Memory Layout**: When a process is loaded into main memory, its Process Control Block (PCB) fields are stored sequentially, beginning at the mainMemoryBase, which is the starting address allocated to the process in main memory. The instructionBase, marking the starting address of the process's instructions, is calculated as mainMemoryBase + PCB size, ensuring that the PCB fields do not overlap with the instructions. Similarly, the dataBase, indicating the starting address of the process's data segment, is determined as instructionBase + numInstructions, allowing logical separation between the instructions and the data. Once these addresses are calculated, the process's logical memory, which includes both the instructions and the data, is copied into the designated space in main memory, ensuring a structured and contiguous layout for efficient execution.

For example, when process 1 gets loaded into main memory you will see main memory structure like this. Note that PCB fields without the logical memory takes up 10 spaces (array locations). Here Address is the array location

| Address | Content | Description |
|---|---|---|
| 0 | 1 | processID |
| 1 | "NEW" (encoded as int) | state |
| 2 | 0 | programCounter |
| 3 | 10 | instructionBase |
| 4 | 13 | dataBase |
| 5 | 300 | memoryLimit |
| 6 | 0 | cpuCyclesUsed |
| 7 | 0 | registerValue |
| 8 | 300 | maxMemoryNeeded |
| 9 | 0 | mainMemoryBase |
| 10 | 1 | Instruction 1: Compute (opcode 1) |
| 11 | 5 | Iterations |
| 12 | 10 | CPU Cycles |
| 13 | 0 | Data (initial value) |
| 14–... | ... | Remaining data |

After process 1 is loaded into main memory, loading of process 2 will be like this:

| Address | Content | Description |
|---|---|---|
| 320 | 2 | processID |
| 321 | "NEW" (encoded as int) | state |
| 322 | 0 | programCounter |
| 323 | 330 | instructionBase |
| 324 | 334 | dataBase |
| 325 | 400 | memoryLimit |
| 326 | 0 | cpuCyclesUsed |
| 327 | 0 | registerValue |
| 328 | 400 | maxMemoryNeeded |
| 329 | 320 | mainMemoryBase |
| 330 | 2 | Instruction 1: Print (opcode 2) |
| 331 | 3 | CPU Cycles for Instruction 1 |
| 332 | 3 | Instruction 2: Store (opcode 3) |
| 333 | 100 | Value for Instruction 2 |
| 334 | 350 | Address for Instruction 2 |
| 335 | 1 | Instruction 3: Compute (opcode 1) |
| 336 | 2 | Iterations for Instruction 3 |
| 337 | 7 | CPU Cycles for Instruction 3 |
| 338 | 2 | Instruction 4: Print (opcode 2) |
| 339 | 5 | CPU Cycles for Instruction 4 |
| 340–... | 0 | Data (initial values) |

**CPU Execution**: The program simulates a CPU that executes instructions for each job loaded into main memory starting at its mainMemoryBase. The CPU fetches instructions by calculating their location in main memory as mainMemoryBase + (instructionBase - logicalMemoryBase), where instructionBase is the address in the process's logical memory. This ensures proper translation of logical memory addresses into main memory locations. The CPU decodes and executes one instruction at a time, updating the corresponding PCB fields during execution. The programCounter is incremented after each instruction to point to the next instruction in logical memory, and the cpuCyclesUsed is updated based on the type of instruction executed.

The process runs to completion without preemption, ensuring all instructions are executed before the CPU moves to the next job. During execution, the program outputs verbose details, including the instruction being executed and the results of the execution, such as values stored or loaded. Once all instructions for the process are completed, the program prints the final PCB details and the total CPU cycles used by the process. This simulation effectively demonstrates the mapping between logical and main memory while simulating process execution in a controlled and systematic manner.

**Structure of the Main Program**:

```
#include <iostream>
#include <queue>
#include <string>
using namespace std;

struct PCB {
    // Define PCB fields as described earlier
};

void loadJobsToMemory(queue<PCB>& newJobQueue, queue<int>& readyQueue,
                      vector<int>& mainMemory, int maxMemory) {

    // TODO: Implement loading jobs into main memory
}

void executeCPU(int startAddress, int* mainMemory) {

    // TODO: Implement CPU instruction execution
}

int main() {
    int maxMemory;
    int numProcesses;
    queue<PCB> newJobQueue;
    queue<int> readyQueue;
    int* mainMemory;

    // Step 1: Read and parse input file
    // TODO: Implement input parsing and populate newJobQueue

    // Step 2: Load jobs into main memory
    loadJobsToMemory(newJobQueue, readyQueue, mainMemory, maxMemory);

    // Step 3: After you load the jobs in the queue go over the main memory
    // and print the content of mainMemory. It will be in the table format
    // three columns as I had provided you earlier.

    // Step 4: Process execution
    while (!readyQueue.empty()) {
        int startAddress = readyQueue.front();
        //readyQueue contains start addresses w.r.t main memory for jobs

        readyQueue.pop();

        // Execute job
        executeCPU(startAddress, mainMemory);

        // Output Job that just completed execution – see example below
    }

    return 0;
}
```

**Output Needed**

1. Print the content of the main memory as described in Step 3 of the program above.
2. Inside the CPU as you execute the instruction output as follows:
    a. If the instruction is print, output the word print
    b. If the instruction is compute, output the word compute
    c. If the instruction is store, if the array location is within bounds say stored, otherwise say store error!
    d. If the instruction is load, if the array location is within bounds say loaded, otherwise say load error!
3. Once a job is completed by the CPU output the details of the process.
   PCB Contents (Stored in Main Memory):
   Process ID: 1
   State: TERMINATED
   Program Counter: 9
   Instruction Base: 10
   Data Base: 13
   Memory Limit: 300
   CPU Cycles Used: 12
   Register Value: 42
   Max Memory Needed: 300
   Main Memory Base: 0
   Total CPU Cycles Consumed: 12

**Rules and Submission Policy**

All projects in this course are **individual assignments** and are not to be completed as group work. The use of **automatic plagiarism detection tools** will be employed to ensure academic integrity. Collaboration with others or the use of outside third parties to complete this project is strictly prohibited. Submissions must be made through **GradeScope**, where automated grading will be conducted. Additionally, manual grading may be performed to ensure correctness and adherence to requirements.

Several input files will be used to evaluate your program. While a subset of these input files will be provided to you for testing, additional files not shared beforehand will also be used during grading. Your score on this project will depend on producing correct results for all input files, including the undisclosed ones used in GradeScope evaluation.

All programs must be written in **C or C++** and must compile successfully using the **GCC** or **GNU C++ compiler**. It is your responsibility to ensure that your program adheres to these requirements.

The course syllabus provides more details on the late and submission policy. You should also go through that.