Data Structures & Algorithms Analysis

CIE 205

Cluster Management

Project Requirements Document

Objectives

By the end of this project, the student should be able to:

- Write a **complete object-oriented C++ program** with **templates** which performs a non-trivial task.
- Use data structures to solve a real-life problem.
- Understand unstructured, natural language problem description and arrive at an appropriate design.
- Intuitively modularize a design into independent components and divide these components among team members.

Introduction

Continuous advancements in the scientific and technological fields have created the need for increasingly capable computational workstations. A computer cluster is a set of computers that work together as one system to achieve some goal(s). The cluster contains several computers (referred to as workers/machines), which can vary in capabilities and peripherals. There are different processes that need to be executed, which are assigned to suitable machines. In this project, we will assume our cluster follows a master/slave architecture, in which there is a master node that divides the workload among the different nodes (workers). The master node acts as the central point which assigns processes to machines and receives notifications from machines after process completion. If we suppose that new processes arrive regularly, we then need to assign the new processes to the different machines that we have available. Using your programming skills and your knowledge of the different data structures, you are going to develop a program that **simulates process assignment** and **calculates** some related **statistics** in order to help improve the overall process.

Project Phases

Project Phase	%
Phase 1	**35%**
Phase 2	**65%**

NOTES:

1. Number of students per team = **3 students.**
2. **The project code must be totally yours. The penalty of cheating any part of the project from any other source is not ONLY taking ZERO in the project grade but also taking MINUS FIVE (-5) from other class work grades; so it is better to deliver an incomplete project than to cheat it. It is totally your responsibility to keep your code private.**
3. At any delivery,

One day late makes you lose **1/2** of the grade.

Two days late makes you lose **3/4** of the grade.

Processes and Machines

There is a number of available machines that can be assigned the different issued processes.

Processes:

The following pieces of information are available for each process:

- **Arrival Time:** the clock cycle when the process arrives and is ready to be assigned to a machine.
- **Process Type:** There are three types of processes: system processes, interactive processes, and computationally intensive processes.
    - **System processes** must be assigned first.
    - **Interactive processes** are processes that require end user interaction. They must be executed by machines equipped with I/O peripherals.
    - **Computationally intensive processes** are jobs that run without end user interaction, and require a large amount of resources. They must be issued to machines equipped with the needed hardware.
- **Dispatch latency:** The number of clock cycles needed to dispatch the process to a machine.
- **Execution time:** The number of clock cycles needed to execute the process (assumed constant regardless of machine type).
- **Priority:** A number representing the importance of the process (the higher the number, the more important the process).

Machines:

At startup, the system loads (from a file) information about the available **machines**. For each machine, the system will load the following information:

- **Machine Type:** There are three types of machines: general-purpose machines, interactive machines, and GPU machines.
    - **General-purpose machines** are machines with standard computing resources and I/O peripherals.
    - **Interactive machines** are machines equipped with highly capable I/O devices.
    - **GPU machines** are machines equipped with graphics processing units (GPUs) that are specialized for intensive computations.
- **Reboot Duration:** The duration (in clock cycles) of system reboot that a machine needs to perform after completing **N** processes.
- **Response time:** The duration of time (in clock cycles) between dispatching the process to the machine, and the time it starts executing the process.

NOTE: Reboot duration and response time are the same for all machines of the same type.

Process Dispatch Criteria

To determine the next process to dispatch (if a machine is available), the following **assignment criteria** should be applied for all the waiting, unassigned processes **at each clock cycle:**

1. First, assign **system processes** to ANY available machine of any type. However, there is a priority based on machine type: first, choose from general-purpose machines THEN interactive machines THEN GPU machines. This means that we do not use interactive machines unless all general-purpose machines are busy, and we do not use GPU machines unless machines of both the other types are busy.

2. Second, assign **computationally intensive processes** using the available GPU machines ***ONLY***. If all GPU machines are busy, wait until one is available.

3. Third, assign **interactive processes** using any type of machines *EXCEPT GPU machines*. First, use the available interactive machines THEN general-purpose machines.

4. If a process cannot be assigned at the current clock cycle, it should wait for the next cycle. At the next cycle, it should be checked whether the process can be assigned now or not. If not, it should wait again, and so on.

NOTES: If processes of a specific type cannot be assigned at the current cycle, try to assign the other types (e.g. if system processes cannot be assigned at the current cycle, this does NOT mean not to assign the interactive processes).

This is how we prioritize the assignment of processes of different types, but how will we prioritize the assignment of processes of **the same type**?

- **For interactive and computationally intensive processes,** assign them based on a first-come-first-served basis.
- **For system processes,** you should design a priority equation for deciding which of the available system processes should be assigned first. System processes with a higher priority are the ones to be assigned first.
  - ☞ You should develop a reasonable **weighted** priority equation depending on at least the following factors: *process arrival time, process dispatch latency, process execution time, and process priority*.

There are some additional services that the master node has to accommodate:

- **For interactive processes ONLY**, a request can be issued to **promote** the process type to become a system process. A request of process **cancellation** could also be issued.
- **For interactive processes ONLY**, if a process waits more than **AutoP** cycles from its arrival time to be assigned to a machine, it should be **automatically promoted** to be a system process. (**AutoP** is read from the input file).

Simulation Approach & Assumptions

You will use incremental **clock cycle** steps. Simulate the changes in the system every **1 processor cycle.**

Some Definitions

- **Arrival Time (AT):**
  The clock cycle at which the process arrives and is ready to be assigned.

- **Waiting Process:**
  A process that has arrived (i.e. process AT < current clock cycle) but has not been assigned yet. At each clock cycle, you should choose the processes to assign from the waiting processes.

- **In-Execution Process:**
  A process that has been assigned to a machine but is not complete yet.

- **Completed Process:**
  A process that has been completed.

- **Waiting Time (WT):**
  The number of clock cycles from the arrival of a process until it starts execution. This includes its waiting time before dispatching and the dispatch latency.

- **Execution Time (ET):**
  The number of clock cycles that a process runs until completion.

- **Completion Time (CT):**
  The clock cycle at which the process is successfully completed.
  (CT = AT + WT + ET)

Assumptions

- If a machine is available at clock cycle t, it can be assigned a new process starting from that cycle.
- More than one process can arrive at the same clock cycle. Also, more than one process can be dispatched and assigned to different machines at the same clock cycle as long as the machines are available.
- A machine can only be executing one process at a time.

Input/Output File Formats

Your program should receive all information to be simulated from an input file and produce an output file that contains some information and statistics about the processes. This section describes the format of both files and gives a sample for each.

The Input File

- First line contains three integers. Each integer represents the total number of machines of each type.
  - **GP:** for general-purpose machines
  - **IO:** for interactive machines
  - **GU:** for GPU machines
- The 2nd line contains three integers:
  - **RGP:** is the response time of all GP machines (clock cycles)
  - **RIO:** is the response time of all interactive machines (clock cycles)
  - **RGU:** is the response time of all GPU machines (clock cycles)
- The 3rd line contains four integers:
  - **N:** is the number of processes the machine completes before rebooting
  - **BGP:** is the reboot duration in cycles for general-purpose machines
  - **BIO:** is the reboot duration in cycles for interactive machines
  - **BGU:** is the reboot duration in cycles for GPU machines
- Then a line with only one integer **AutoP** which represents the number of cycles after which an interactive process is automatically promoted to a system process.
- The next line contains a number ***E*** which represents the number of **events** following this line.
- Then the input file contains ***E*** lines (one line for **each event**). An event can be:
  - Arrival of a new process. Denoted by letter **A**, or
  - Cancellation of an existing process. Denoted by letter **X**, or
  - Promotion of a process to be a computationally intensive process. Denoted by letter **P**.

  NOTE: The input lines of all events are **sorted by the arrival time** (**AT)** in **ascending** order.

Events

- **Arrival event line** has the following information:
  - **A** (letter A at the beginning of the sentence) means a process arrival event.
  - **TYP** is the process type (*S: system, I: interactive, C: computationally intensive*).
  - **AT** is the arrival time.
  - **ID** is a unique sequence number that identifies each process**.**
  - **DL** is the process dispatch latency (in clock cycles).
  - **ET** is the process execution time (in clock cycles).
  - **P** is the process priority.
- **Cancellation event line** has the following information:
  - **X** (Letter X) means a process cancellation event.
  - **AT** is the event arrival time.

- **ID** is the ID of the process to be canceled. This ID must be of a system process.

- **Promotion event line** has the following information:
  - **P** (Letter P) means a process promotion event.
  - **AT** is the arrival time.
  - **ID** is the ID of the process to be promoted to system process. This ID must be of an interactive process.

Sample Input File

```
3  3  2                          no. of machines of each type
1  2  2                          response time of each type (cycles)
3  9  8  7                       no. of processes before checkup and the checkup durations
25                               auto promotion limit
8                                no. of events in this file
A  S  2  1  8  10  5             arrival event example
A  S  5  2  10 15  4
A  I  5  3  3  6   3
A  C  6  4  2  5   4
X  10 1                          cancellation event example
A  I  18 5  2  10  9
P  19 3                          promotion event example
A  C  25 6  9  10  1
```

The Output File

The output file you are required to produce should contain **M** output lines of the following format:
CT ID AT WT ET
which means that the process identified by sequence number **ID** has arrived at cycle **AT.** It then waited for a period **WT** to be assigned. It has then taken **ET** to be completed at the cycle **CT**.
(Read the "Definitions Section" mentioned above)
The output lines **must be sorted** by **CT** in ascending order. If more than one process is completed at the same cycle, **they should be ordered by ET**.

Then the following statistics should be shown at the end of the file:

1. Total number of processes and number of processes of each type
2. Total number of machines and number of machines of each type
3. Average waiting time and average execution time in cycles
4. Percentage of automatically-promoted processes (relative to the total number of interactive processes)

Sample Output File

The following numbers are just for clarification and are not produced by actual calculations.

CT	**ID**	**AT**	**WT**	**ET**
18	**1**	**7**	**5**	**6**
44	**10**	**24**	**2**	**18**
49	**4**	**12**	**20**	**17**
……				
……				
Processes: 124	[S: 100, I: 15, C: 9]			
Machines: 9	[GP: 5, IO: 3, GU: 1]			
Avg Wait = 12.3,	Avg Exec = 25.65			
Auto-promoted: 7%				

Program Interface

The program can run in one of three modes: **interactive, step-by-step,** or **silent mode**. When the program runs, it should ask the user to select the program mode.

1. Interactive Mode: Allows user to monitor the waiting, in-execution and completed processes of each type. The IDs of **[system processes]** are printed within **[]**, the IDs of **(interactive processes)** are printed within **()**, and the IDs of **computationally intensive ones** are printed normally. At each cycle, the program should provide output **similar** to the one below. In this mode, the program pauses for an input from the user (1 for instance) to display the output of the next cycle.

```
Available Machines:      (1) [2] 5 [10]
Waiting Processes:       (5) 6 (8)
In-Execution Processes: [2] (3) 7
Completed Processes:     1 [4]

Current cycle:  6
```

Output Screen Explanation

The IDs of available machines of each type are displayed first. Next, the IDs of waiting, in-execution and completed processes of all types should be displayed.

NOTE: The IDs in the above sample are just for illustration.

After the first 4 lines, the following information should be printed:
- Current simulation cycle number
- Number of waiting processes of each type
- Number of available machines of each type
- Type & ID of **ALL** machine**s** and process**es** that were assigned at the **last** cycle only.
  e.g. **IO6->S3** □ interactive machine #6 assigned to system process #3
- Total number of processes completed so far of each type

2. Step-by-Step Mode is identical to the interactive mode except that after each cycle, the program waits for one second (not for user input) then resumes automatically.

3. In **Silent Mode**, the program produces only an output file (See the "File Formats" section). It does not print anything on the console.

NOTE: No matter what mode of operation your program is running in, **the output file** should be produced.

Project Phases

You are required to write **object-oriented** code with **templates** for data structure classes.

Before explaining the requirement of each phase, all the following are NOT allowed to be used in your project:

- You are not allowed to use **C++ STL** or any external resource that implements the data structures you use. ***This is a data structures course where you should build data structures yourself from scratch.***

- You need to get instructor's approval before making any **custom (new)** data structure.
  NOTE: No approval is needed to use the known data structures.

- ***Do NOT allocate the same process more than once***. Allocate it once and make whatever data structures you choose point to it (pointers). Then, when another list needs an access to the same process, DON'T create a new copy of the same process; just **share** it by making the new list point to it or **move** it from current list to the new one.
  SHARE, MOVE, DON'T COPY…

- You are not allowed to use **global variables** in your code.

- You need to get instructor approval before using **friendships.**

Phase 1

In this phase you should finish implementing ALL **data structures** needed for BOTH phases without implementing logic related to assigning the processes**.** The required parts to be finalized and delivered at this phase are:

1- **Full data members** of Process, Machine and MasterNode classes
2- **Full "template" implementation of ALL data structures (DS)** that you will use to represent **the lists of processes and machines**. All data structures needed for both project phases must be finished in this phase.

   Important: Keep in mind that you are **NOT** selecting the DS that would **work in phase 1 only**. ***You must choose the DS that will work efficiently for both phase 1 & phase 2.***

   When choosing the DS think about the following:
   a. How will you store **waiting processes**? Do you need a separate list for each type?
   b. What about the **machine lists**?
   c. Do you need to store **completed processes**? When should you delete them?
   d. **Which list type** is much suitable to represent each list? You must take into account the **complexity of the main operations** needed for each list (e.g. insert, delete, retrieve, shift, sort …etc.). For example, if the most frequent operation in a list is deleting from the middle,

use a data structure that has low complexity for this operation.

You need to justify your choice for each DS and why you separated or joined lists. Selecting the appropriate DS for each list is the core target of phase 1 and the project as a whole. Most of the discussion time will be about that.

NOTE: You need to read *"File Format"* section to see how the input data and output data are sorted in each file because this will affect the selection of the data structures.

3- **File loading function:** The function that reads input file to:
   a. Load machine data and populate the machine lists(s).
   b. Load events data and populate the events list.

4- **Simple simulator function for phase 1**: The main purpose of this function is to test your data structures and how to move processes and machines between lists. This function should:

- Perform any needed initializations.
- Call file loading function.
- ***At each clock cycle,*** do the following:
   a. Get the events that should be executed at the current cycle.
      i. For the arrival event, generate a process and add it to the appropriate waiting processes list.
      ii. For the cancellation event, delete the corresponding **system process** (**if found**).
      iii. Ignore promotion events.
   b. **Pick one process** from each type and move it to in-execution list(s).
      NOTE 1: The process you choose to delete from each type must be the first process that should be assigned to an available machine in phase 2.
      NOTE 2: NO actual machines check_availability/assignment is required in Phase 1.
   c. Each **5 cycles**, move a process of each type from the in-execution list(s) to the completed list(s).
   d. Print to the console:
      i. The appropriate IDs of machines and processes
      ii. The number of waiting processes of each type
      iii. The number of available machines of each type
   e. The simulation function stops when there are no more events or active processes in the system.

Notes about phase 1:

- No output files should be produced at this phase.
- In this phase, you can go to the next cycle by user input (as in interactive mode).
- No process execution or assigning machines will be done in this phase. However, all the lists of the project should be implemented in that phase.
- Make sure you read **Project Evaluation** and **Individuals Evaluation** section mentioned below.

Phase 1 Deliverables:

Each team is required to submit the following:

- A text file named ***ID.txt*** containing team members' names, IDs, and emails.
- **Phase 1 full code** [Do not include executable files].
- **Three sample input files** (test cases).
- **Workload document:** how the load is divided between members in this phase. **Print** it and bring it with you on the discussion day.
- **Phase 1 document** with 1 or more pages describing:
  - Each process and machine list you chose
  - The DS you chose for each list
  - Your justification of all your choices with the complexity of the most frequent or major operation for each list.

Phase 2

In this phase, you should extend code of phase 1 to build the full application and produce the final output file. Your application should support the different operation modes described in "Program Interface" section.

Phase 2 Deliverables:

Each team is required to deliver the following:

- A text file named ***ID.txt*** containing team members' names, IDs, and emails.
- **Final project code** [Do not include executable files].
- **Six comprehensive sample input files (test cases) and their output files.**
- **Workload document**. Don't forget to **print** it and bring it with you on the discussion day.

Project Evaluation

These are the main points you will be graded on in the project:

- **Successful Compilation:** Your program must compile successfully with zero errors. Delivering the project with any compilation errors will make you lose a large percentage of your grade.

- **Object-Oriented Concepts:**
  - **Modularity:** A **modular** code does not mix several program features within the same unit (module). For example, the code that does the core of the simulation process should be separate from the code that reads the input file which, in turn, is separate from the code that implements the data structures. This can be achieved by:
    - Adding classes for each different entity in the system and each DS used.
    - Dividing the code in each class to several functions. Each function should be responsible for a single job. Avoid writing very long functions that do everything.
  - **Maintainability:** A maintainable code is the one whose modules are easily modified or extended without a severe effect on the other modules.
  - **Separate each class in .h and .cpp** files *(if not a template class).*

  - **Class responsibilities:** No class is performing the job of another class.

- **Data Structures & Algorithms**: After all, this is what the course is about. You should be able to provide a concise description and a justification for: (1) the data structure(s) and algorithm(s) you used to solve the problem, (2) the **complexity** of the chosen algorithm, and (3) the logic of the program flow.

- **Interface modes**: Your program should support the three modes described in the document.

- **Test Cases**: You should prepare different comprehensive test cases (at least 6). Your program should be able to simulate different scenarios not just trivial ones.

- **Coding style**: How elegant and **consistent** is your coding style (indentation, naming convention, ...)? How useful and sufficient are your comments? This will be graded.

Grading

Item	Percentage
Teamwork	70%
Individual Work*	30%

*Each member must be responsible for writing some project modules (e.g. some classes or some functions) and must answer questions showing that he/she understands both the program logic and the implementation details. The workload between team members must be almost equal.

Bonus Criteria (Maximum 10% of Project Grade)

- **[3.5%] Machine response time:** Machines of the same type may have different response times. The machines of a specific type must be sorted by their response time. Shorter response time machines of a certain type should have higher priority to be assigned to processes than longer response time machines of the same type.
- **[3.5%] Machine maintenance:** Machines can sometimes need maintenance apart from their regular checkups. If this happens, they should be unavailable for some time. If the system needs this machine before its maintenance is over, this will cause the machine's response time to be increased until the end of the simulation.
- **[3%] More Process Types:** Think about **two** more process types other than those given in the document. The load of the logic of the two processes must be acceptable (**Needs instructor's approval**).

Appendix A – Guidelines on Project Framework

The main classes of the project should be MasterNode, Process, Machine, Event, and UI (User Interface).

Event Classes:

There are three types of events: Arrival, Cancellation, and Promotion events. You should create a base class called "***Event***" that stores the event time and the related process ID. This should be an abstract class with a pure virtual function "***Execute***". The logic of the Execute function should depend on the Event type.

For each type of the three events, there should be a class derived from ***Event*** class. Each derived class should store data related to its operation. Also, each of them should override the function **Execute** as follows:

1. ArrivalEvent::Execute should create a new process and add it to the appropriate list
2. CancelEvent::Execute should cancel the requested system process (if found and is waiting)
3. PrompteEvent::Execute should move a system process to the computationally intensive list and update the process data (if found and is waiting)

Class MasterNode should have an appropriate list of ***Event*** pointers to store all events loaded from the file at system startup. At each clock cycle, the code loops on the event list to execute all events that should take place at the current cycle.

UI Class

This should be the class responsible for taking in any inputs from the user and printing any information on the console. It contains the input and output functions that you should use to show status of processes at each cycle.

Main members of class UI:

- ***PROG_MODE getProgramMode():***
  - This function should be called at the very beginning to get the program mode from the user.
  - PROG_MODE is an enum containing the different modes of the program.
- ***void printString(std::string text):*** Prints text on the console.
- ***void waitForUser():*** This function will wait for input from the user (needed in the interactive mode).
- ***static void sleep(int milliseconds):*** Block the program for a certain time (needed in the step-by-step mode).

You might want to add other functions like ***void addToWaitingString(std::string text)*** to add to the string of the waiting processes (and similar ones for the rest of the lines that should be printed on the console).