Processor Execution Simulator

Atypon Task-Report

By: Farah Jamal

Instructor: Fahed Jubair

What exactly is the processor execution?

- Almost all programs have some alternating cycle of CPU number crunching and waiting for I/O of some kind. (Even a simple fetch from memory takes a long time relative to CPU speeds.)
- In a simple system running a single process, the time spent waiting for I/O is wasted, and those CPU cycles are lost forever.

- A scheduling system allows one process to use the CPU while another is waiting for I/O, thereby making full use of otherwise lost CPU cycles.
- The challenge is to make the overall system as "efficient" and "fair" as possible, subject to varying and often dynamic conditions, and where "efficient" and "fair" are somewhat subjective terms, often subject to shifting priority policies.

Problem Domain:

The main problem is to build a simulator that simulates processor execution for processes.

If we have P processors, consider the following notes about the processor execution:

- A processor can only execute one task at a time.
- The number of processors is fixed during the whole simulation.
- All processors are synchronized, i.e., they all have the same clock.
- Processors are given unique ids as follows: P1, P2, P3, P4, etc.
- Tasks are given unique ids as follows: T1, T2, T3, T4, etc.
- Clock cycles are given unique ids as follows: C1, C2, C3, C4, etc.
    - A task is defined by the following:
    - Creation time: the clock cycle in which the task was created.
    - Requested time: the number of clocks cycles the task needs to execute till completion.
    - Completion time: the clock cycle in which the task was completed.
    - Priority: which can be either high or low.
    - State: which can be either one of three states

Important Definitions:

1- **creation Time:** creation time is the point of time at which a process enters the ready queue.

2- **Waiting Time**: Waiting time is the amount of time spent by a process waiting in the ready queue for getting the CPU.

3- **Requested Time:**
   - Requested time is the amount of time required by a process for executing on CPU.
   - It is also called as **execution time** or **running time.**
   - Requested time of a process cannot be known in advance before executing the process.
   - It can be known only after the process has executed.

4- Turnaround Time:
   - Turn Around time is the total amount of time spent by a process in the system.
   - When present in the system, a process is either waiting in the ready queue for getting the CPU or it is executing on the CPU.

Turn Around time = Requested time + Waiting time

OR

Turn Around time = Completion time – Creation time

** While discussing the above definitions,

- We have considered that the process does not require an I/O operation.
- When process is present in the system, it will be either waiting for the CPU in the ready state or it will be executing on the CPU.

Visualization:

There is many scheduling types happens on the CPU in this task we require to implement the Priority Scheduling

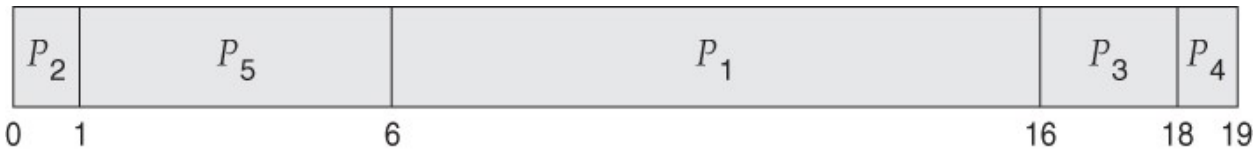
- Priority scheduling is a more general case of SJF(shortest job first), in which each job is assigned a priority and the job with the highest priority gets scheduled

first. (SJF uses the inverse of the next expected burst time as its priority - The smaller the expected requested , the higher the priority.)

- Note that in practice, priorities are implemented using integers within a fixed range, but there is no agreed-upon convention as to whether "high" priorities use large numbers or small numbers. This book uses low number for high priorities, with 0 being the highest possible priority.

- For example, the following Gantt chart is based upon these processes requested times and priorities, and yields an average waiting time of 8.2 ms:

• Process	Requested Time	Priority
P1	10	3
P2	1	1
P3	2	4
P4	1	5
P5	5	2



- Priorities can be assigned either internally or externally. Internal priorities are assigned by the OS using criteria such as average burst time, ratio of CPU to I/O activity, system resource use, and other factors available to the kernel. External priorities are assigned by users, based on the importance of the job, fees paid, politics, etc.
- Priority scheduling can be either preemptive or non-preemptive.
- Priority scheduling can suffer from a major problem known as ***indefinite blocking***, or ***starvation***, in which a low-priority task can wait forever because there are always some other jobs around that have higher priority.

- If this problem is allowed to occur, then processes will either run eventually when the system load lightens (at say 2:00 a.m.), or will eventually get lost when the system is shut down or crashes. (There are rumors of jobs that have been stuck for years.)
- One common solution to this problem is ***aging***, in which priorities of jobs increase the longer they wait. Under this scheme a low-priority job will eventually get its priority raised high enough that it gets run.

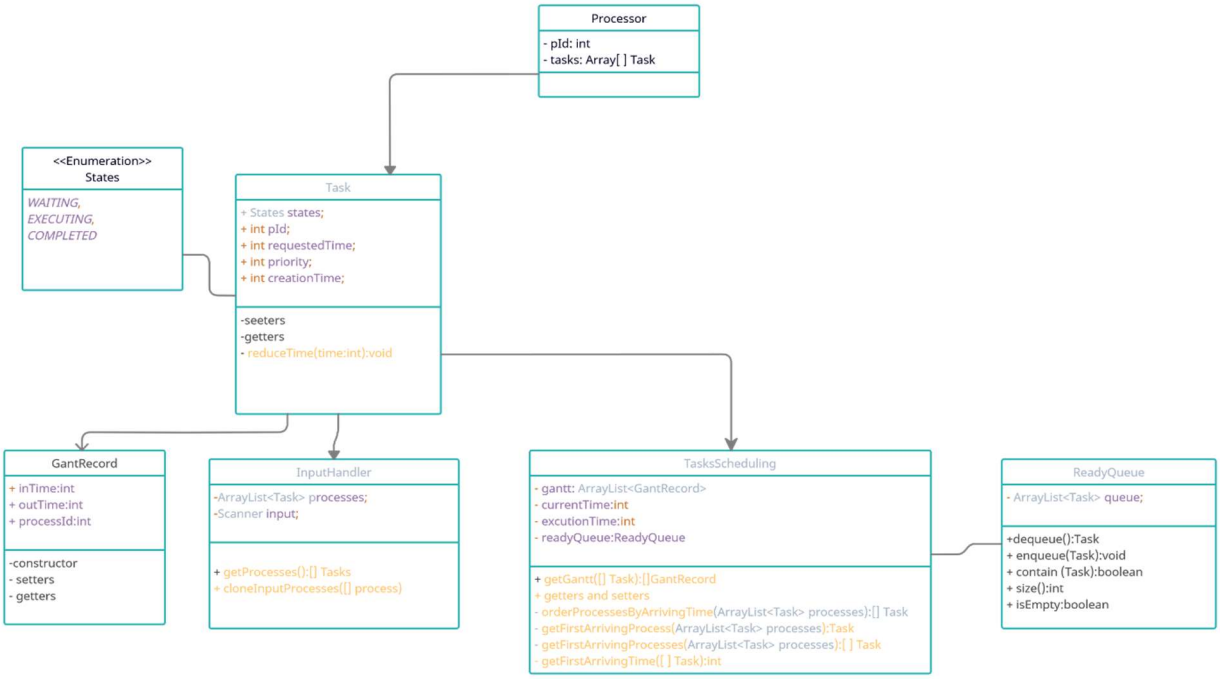
Steps through the solution:

First Step:

Investigate and read on computer operating system topics and trying to collect enough data to solve the problem.

Second Step:

Draw the UML class Diagram to determine what exactly I will have classes and function on my code I use draw.io to draw my UML diagram.



Explain the UML Class diagram:

- **Processor class may have many tasks**
- **Each task has five main properties**
    - **States which are Enum class may has one of three states**
    - **Process/task Id**
    - **Requested Time**
    - **Creation Time**
    - **Priority**
- **Tasks must be scheduled using task scheduler class**
- **Gant Record to draw the task priority execution**
- **Input Handler has all tasks properties input by user manually or using file**
- **Ready Queue helps to add tasks and get tasks**

Third Step:

Start the coding and the logic of my program

The structure of my project is:

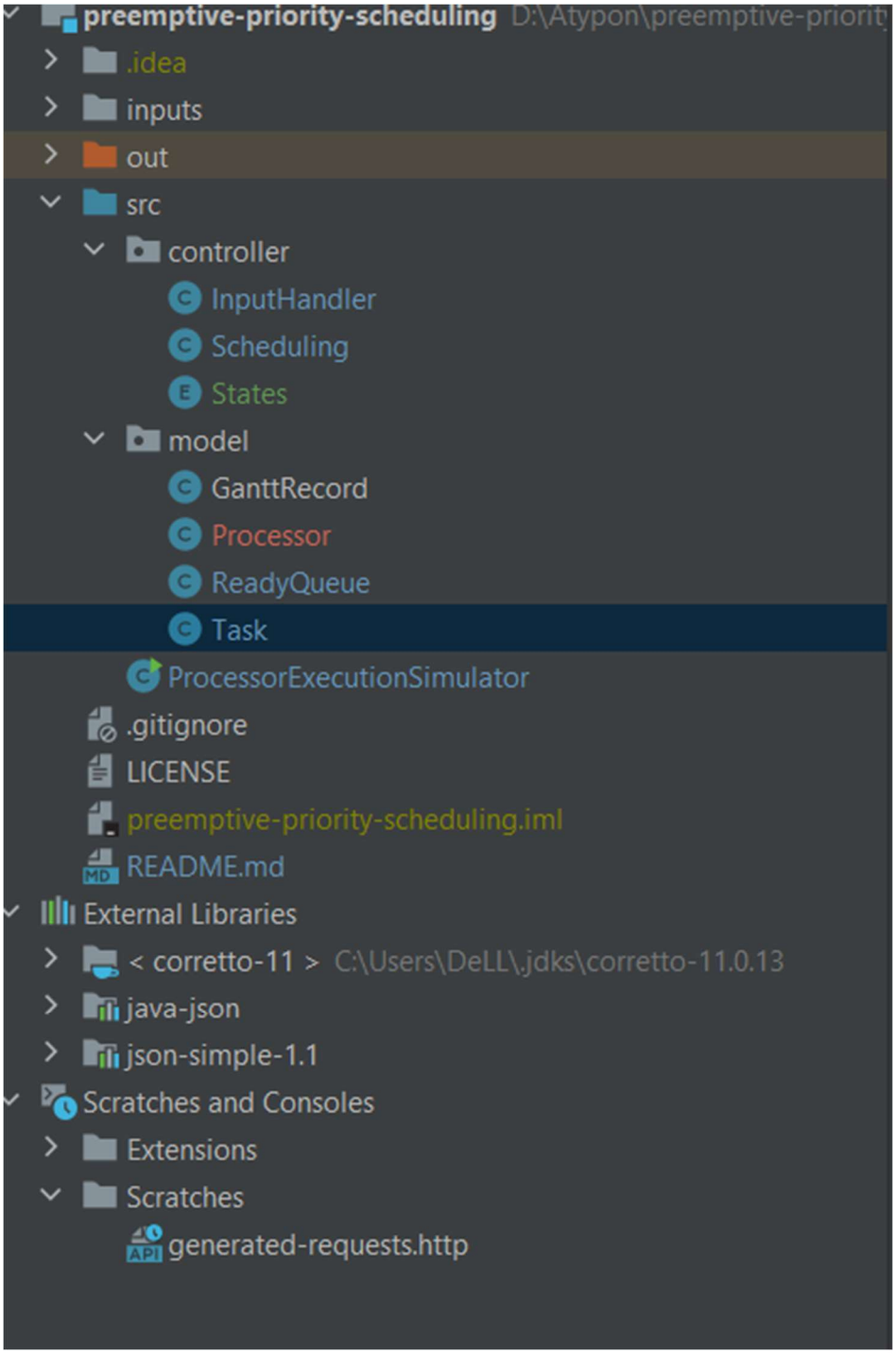


Figure 1 project structure

Let’s start by describe everything inside this project:-

1- Controller Package

a. InputHandler Class

i. This class simple class take the user input and handle it inside processor and tasks constructor

```
   private ArrayList<Task> tasks;
   private Scanner input;
Processor processor=new Processor();
   public InputHandler(){
      tasks = new ArrayList<>();
      input = new Scanner(System.in);
   }
```

ii. Have two functions one for input process

iii. Second one for iterate over tasks array and make copy from it

b. Scheduling Class

- The whole logic stands here focus ^^
- We have initial values:

```
•
    private ArrayList<GanttRecord> gantt;
    private int currentTime;
    private ReadyQueue readyQueue;
```

- (gantt) array helps you to build Gantt chart see below:-

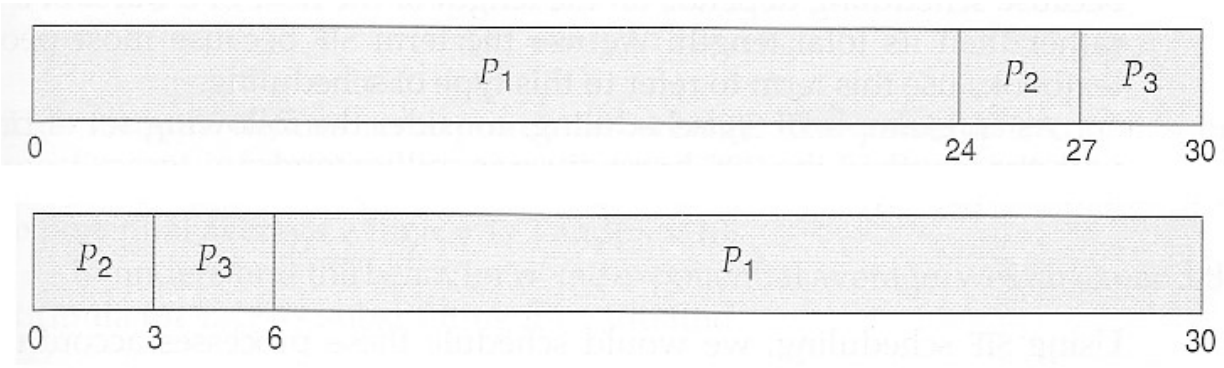


Figure 2:Gantt chart

- Current Time help processor to calculate the time for each process
- readyQueue help the processor to control the tasks execution
- constructor to control the data initialization

Functions:

1- getGantt function takes arrays of tasks and return arrays of ganttRecord

```
public ArrayList<GanttRecord> getGantt(ArrayList<Task> tasks){
```

initialization of creation times

```
currentTime = this.getFirstCreationTime(tasks);
int in ,out = currentTime;
```

the first processes in the queue are ready

```
ArrayList<Task> processes1 = this.getAllCreatedProcesses(tasks);
```

the increase in the queue is almost based on priority so the highest priority will remove from Ready Queue

```
for(Task task : processes1){
    readyQueue.enqueue(task);
    tasks.remove(task);
}
```

the remaining processes are listed depending on the creation time

```
ArrayList<Task> orderedByArrivingTime = this.orderProcessesByCreationTime(tasks);
```

if the queue is not almost empty then deals with the process with the highest priority from the turn ready

```
while(!readyQueue.isEmpty()){
    Task task = readyQueue.dequeue();
```

Notice that we have **Two** cases to consider, the first is when we have new processes coming and the other case is when new processes do not come but those that are ready are treated.

```
if(orderedByArrivingTime.size() > 0) {
    //Handle of all arriving processes while one process has the control of the Processor

    for (int i = 0; i < orderedByArrivingTime.size(); i++) {
        Task p = orderedByArrivingTime.get(i);

        // the new process that comes when the Processor is busy compares the priority and if the priority is
with
        // decreases that the priority of the process that has control then the new process is added to the
ready queue

        if (p.getCreationTime() >= task.getCreationTime()
```

```
            && p.getCreationTime() < (task.getRequestedTime() + currentTime)
            && p.getPriority() >= task.getPriority()) {
        readyQueue.enqueue(p);
        orderedByArrivingTime.remove(p);
        i--;
      }

      // the new process that comes when the Processor is busy compares the priority and if the priority is
with
      // greater than the priority of the process that has control then the old process is added to the ready
queue
      // with my requested time reduced, Processor control is taken over by the new process

       else if (p.getCreationTime() >= task.getCreationTime()
            && p.getCreationTime() < (task.getRequestedTime() + currentTime)
            && p.getPriority() < task.getPriority()) {
        in = currentTime;
        currentTime = p.getCreationTime();
        task.reduceTime(currentTime - in);
        out = currentTime;
        readyQueue.enqueue(task);
        GanttRecord gR = new GanttRecord(in, out, task.getTaskID());
        gantt.add(gR);

        readyQueue.enqueue(p);
        orderedByArrivingTime.remove(p);
        i--;

        break;
      }
```

if you check the entire list of new processes coming up and none of them are valid to take control of the Processor, the process that has the control continues with the time it needs

```
if (i == orderedByArrivingTime.size() - 1) {
    in = currentTime;
    currentTime += task.getRequestedTime();
    out = currentTime;
    gantt.add(new GanttRecord(in, out, task.getTaskID()));

    // checks if at the end of the uninterrupted execution of a process we have a new process that
    // comes and goes in the ready row

    if(orderedByArrivingTime.size() > 0
            && readyQueue.isEmpty()) {
        readyQueue.enqueue(orderedByArrivingTime.get(0));
    }
}
```

the other case when we do not have new processes that follow but only those that are in the queue are treated

```
else{
    in = currentTime;
    currentTime += task.getRequestedTime();
    out = currentTime;
    gantt.add(new GanttRecord(in, out, task.getTaskID()));
}
```

then return gantt arraylist

```
return gantt;
```

- getCompletionTime function used to help for getting the completion time for each process/task

```
public static int getCompletionTime(Task p, ArrayList<GanttRecord>
gantt) {
```

```
    int completionTime = 0;
    for(GanttRecord gR : gantt){
        if(gR.getProcessId() == p.getTaskID())
            completionTime = gR.getOutTime();
        p.setStates(States.EXECUTING);
    }
    return completionTime;
}
```

- getTurnArroundTime helps to calculate the waiting time

```
public static int getTurnAroundTime(Task p, ArrayList<GanttRecord> gantt)
{
    int completionTime = Scheduling.getCompletionTime(p,gantt);
    return completionTime-p.getCreationTime();
}
```

- getWaitingTime helps to show the waiting time for each process/task

```
public static int getWaitingTime(Task p, ArrayList<GanttRecord> gantt)
{
    int turnAroundTime = Scheduling.getTurnAroundTime(p,gantt);
    return turnAroundTime-p.getRequestedTime();
}
```

- orderProcessesByCreationTime function help to order processes by it's creation time

```
private ArrayList<Task> orderProcessesByCreationTime(ArrayList<Task> tasks){
    ArrayList<Task> newTasks = new ArrayList<>();
    while(tasks.size() != 0) {
        Task p = this.getFirstCreatedProcess(tasks);
        tasks.remove(p);
        newTasks.add(p);
    }
```

```
    return newTasks;
}
```

- getFirstCreatedProcess helps orderProcessesByCreationTime function by ordering the tasks

```
private Task getFirstCreatedProcess(ArrayList<Task> tasks){
  int min = Integer.MAX_VALUE;
  Task task = null;
  for(Task p : tasks){
      if(p.getCreationTime() < min){
          min = p.getCreationTime();
          task = p;
      }
  }
  return task;
}
```

- when process is created and ready this function will called when the first processes in the queue are ready

```
private ArrayList<Task> getAllCreatedProcesses(ArrayList<Task> tasks){
  int min = this.getFirstCreationTime(tasks);
  ArrayList<Task> processes1 = new ArrayList<>();
  for(Task p : tasks){
      if(p.getCreationTime() == min){
          processes1.add(p);
      }
  }
```

```
    return processes1;
}
```

- getFirstCreationTime helps to get the tasks by first creation time

```
private int getFirstCreationTime(ArrayList<Task> tasks){
    int min = Integer.MAX_VALUE;
    for(Task p : tasks){
        if(p.getCreationTime() < min){
            min = p.getCreationTime();
        }
    }
    return min;
}
```

- **states Enum:**
    - **contains states for every task**

```
package controller;

public enum States {
    WAITING,
    EXECUTING,
    COMPLETED

}
```

2- Model Package

a. GanttRacord Class

i. Simple class contains data needed for drawing the gantt chart

```
private int inTime;
private int outTime;
private int processId;
```

ii. And contains setters and getters for every variable in addition to constructor

iii. toString method to print data

```
                public String toString() {
    return "|" + inTime + "| --P" + processId + "-- |" + outTime + "|";
}
```

b. Processor Class

i. Class just for take bunch of tasks

```
package model;

import java.util.ArrayList;

public class Processor extends ArrayList<Task> {
    public  ArrayList<Task> tasks=new ArrayList<>();
    public  int processorID;

    public ArrayList<Task> getTasks() {
        return tasks;
    }

    public void setTasks(ArrayList<Task> tasks) {
        this.tasks = tasks;
    }

    public int getProcessorID() {
        return processorID;
    }

    public void setProcessorID(int processorID) {
        this.processorID = processorID;
    }
}
```

c. ReadyQueue

i. Queue data structure implemented to handle process/tasks in out for processor

Dequeue remove task from queue

```
public Task dequeue()
{
    Task p = null;
    if (!isEmpty())
    {
        p = queue.get(0);
        queue.remove(p);
    }
    return p;
}
```

enqueue add task to the queue

```
    public void enqueue(Task task)
    {
//the case when the queue is empty
      if (queue.isEmpty()) {
            queue.add(task);
        }
        else if(!this.contain(task)){
            int i;
            for (i = 0; i < queue.size(); i++) {
                if (queue.get(i).getPriority() > task.getPriority()) {
                    queue.add(i, task);
                    break;
                }
// the case when the priority of the added process is higher than of all
processes
            if(i == queue.size() ){
                queue.add(task);
            }
```

Contain to prevent duplicate tasks in the queue

```
private boolean contain(Task task){
    for(Task p : queue){
        if(p.getTaskID() == task.getTaskID())
            return true;
        return false;
    }
    return false;
}
```

d. Task Class

i. Contains setter and getters

ii. Constructor

```
public class Task {
    private int taskID;
    private int priority;
    private int creationTime;
    private int requestedTime;
    private States states;
```

iii. reduceTime help requested time reduced So Processor control is taken over by the new process

```
public void reduceTime(int time) {
    if(requestedTime >= time)
        requestedTime = requestedTime - time;
}
```

3- Main class

a. Print data

i. I see it's important to the user to see

1. List of all processes.
2. GANTT chart
3. Completion Time for each process/task

4. Waiting time
5. Turnaround time
6. Average waiting time
7. Average Turnaround time

4- Inputs folder: contains 2 suggested files as json file to execute the program.

- To run this program, you need to
  - cd out/
  - java ProcessorExecutionSimulator
- To Compile this project, you must run
  - javac -sourcepath src -d out src/ rocessorExecutionSimulator.java

Output Sample:

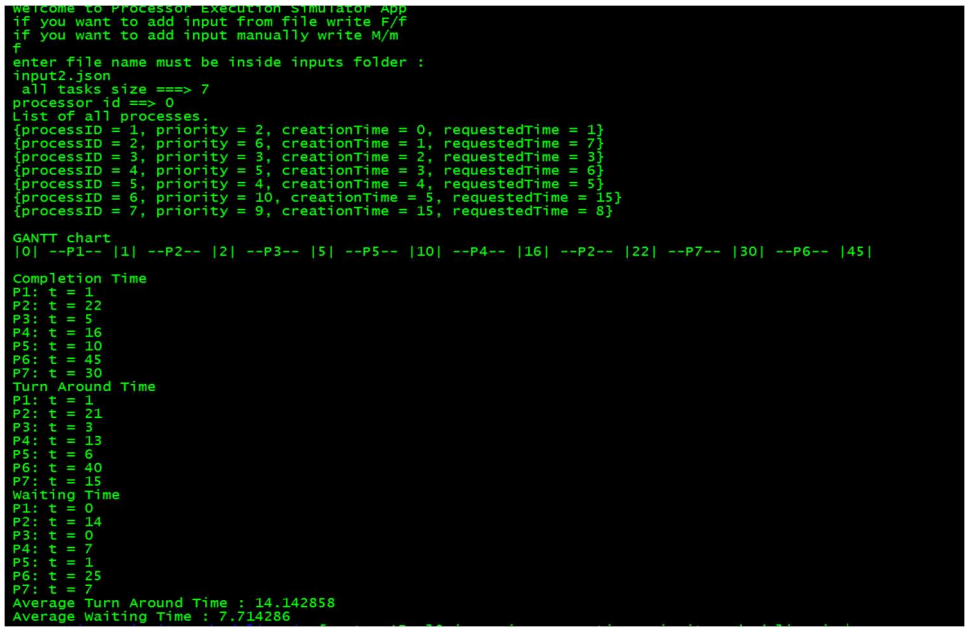


Figure 3 input by file

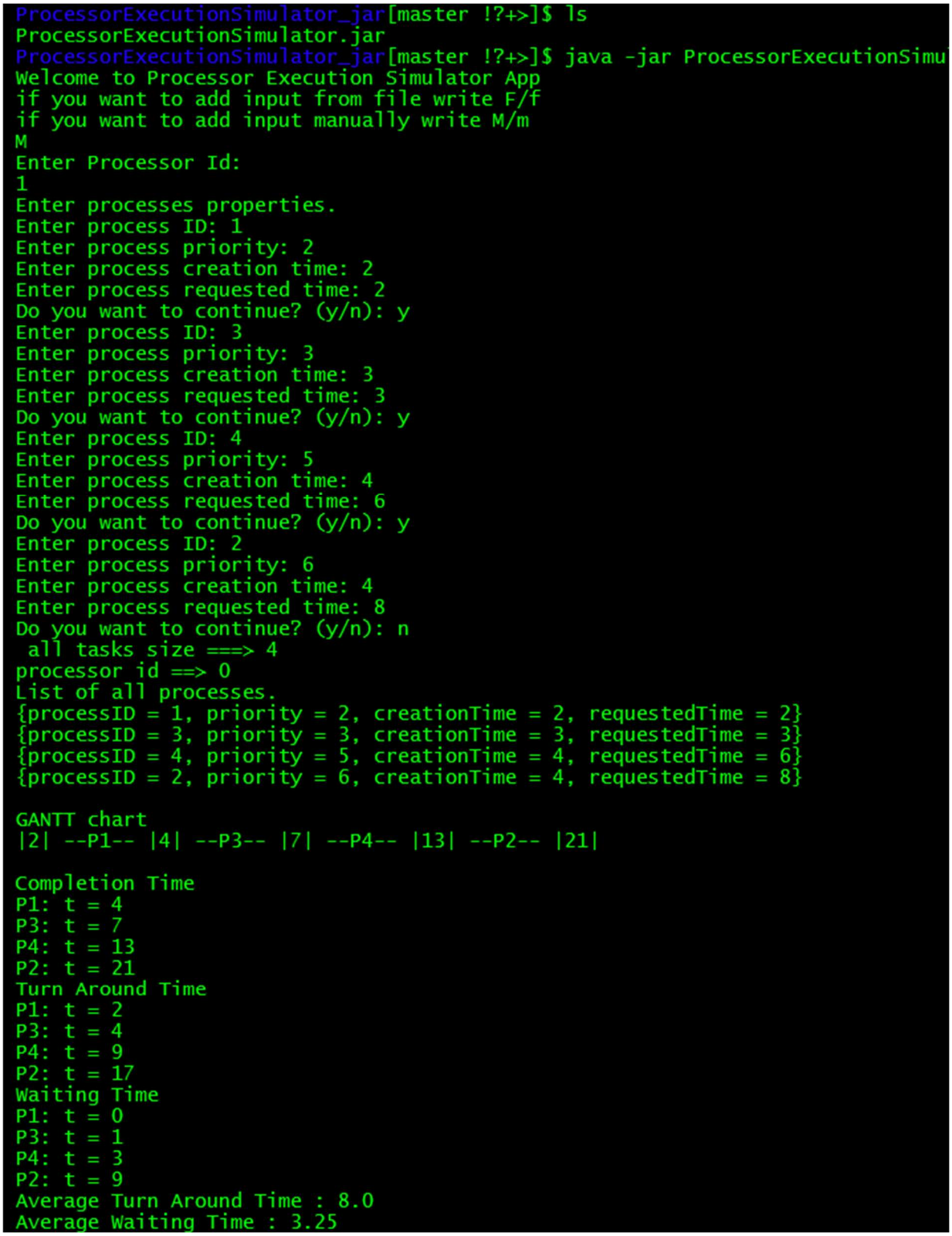


Figure 4 input by user manually