**GROUP PROJECT**

Instructor: Dr. Karne

COSC 519

**************************************************************

# PROCESS MANAGEMENT SIMULATION

**************************************************************

**GROUP MEMBERS**

## A. INTRODUCTION

Process Management is an essential component of modern day Operating Systems. It is responsible for managing processes from start to finish. A process is a program in execution. An Operating System with the help of Process Management, allocates resource for a process, enables sharing and protection of data, and puts it in a job queue using a job scheduler. Once a process comes in the job queue, it needs be assigned to a CPU for it to be able to complete its tasks. A CPU scheduler will fetch the process from the queue and feed to the CPU. Which process gets selected by CPU scheduler depends on the scheduling algorithm implemented.

One of the components that can directly impact Operating System's performance is the CPU. Process Management's goal is to keep the CPU occupied to make it seem like all processes are running concurrently. How an Operating System utilizes affects the degree of multiprogramming. In this project, we will try to understand Process Management in an environment with multiple CPUs.

## B. Objectives and Description

The goal of this project is to simulate the Process Management component of the Operating System in a multi-CPU environment. Since this project is about the simulation of Process Management in a multi-CPU environment, many other aspects of the Process Management will be not implemented. We will focus solely on how processes are assigned to available CPUs. This simulation will utilize two CPUs. The simulator has following components:

1) Process (fixed number of process)
2) Process Generator
3) Ready Queue
4) Dispatcher
5) CPU (2)

Multiple processes will be generated using the Process Generator, and those processes will be placed in the Ready Queue. The Dispatcher will grab a process from the Ready Queue and feed to the available CPU. Process selection will be based on First-Come-First-Serve principle. Each process will have a maximum execution (burst) time associated with it. A process will be considered terminated once it reaches its maximum execution time. The Dispatcher will take the terminated process away from the CPU, and mark it complete.

Simulation application is written in C++. Each component of the simulator is represented by a class/object, and the main function will drive the simulator.

## D. Analysis and Design

### 1. Scope and limitations

The purpose of the current project is to simulate the operating system process management. That is we do not intend to reproduce the entire process management system in all details. Instead we aim to implement and learn the life cycle of a process from creation to complete execution highlighting multiple interactions that occurs between operating system components i.e., dispatcher, ready queue, processor, and memory. The virtual memory, cache memory, and I/O activities are not considered in the present simulation. The processes actually do not perform any specific job.

## 2. Assumptions and Requirements

- For simulation purpose as stated above, we made the following assumptions:
  a. the number of process created is limited.
  b. Our System is a dual processor system.
  c. The only memory involved is the ready queue.
  d. there is no Read/Write Conflict between processors and the process generator
  e. The process scheduling is a First In First Out (FIFO) schedule
  f. All processes or jobs are executed by the processor for a preset and fixed burst time after what the process is either complete and terminated or sent back to the ready queue.
  g. the multiprocessing is set to synchronous meaning the processors are equal peers and share the same memory.

- Requirements:

**Processor:**

1. Processor runs the job sent by the dispatcher for a fixed time after what the job is either terminated or sent back at the end the ready queue.
2. When done, the processor set an availability flag to 1 and waits for the next job.

**Job Generator**

1. The Process generator creates jobs sequentially and adds them to the queue until the preset number of job is reached.

**Dispatcher**

- The dispatcher checks availability of the processors and sent the job at the top of the queue to the available one.
- The dispatcher should first check that the ready queue is not empty.

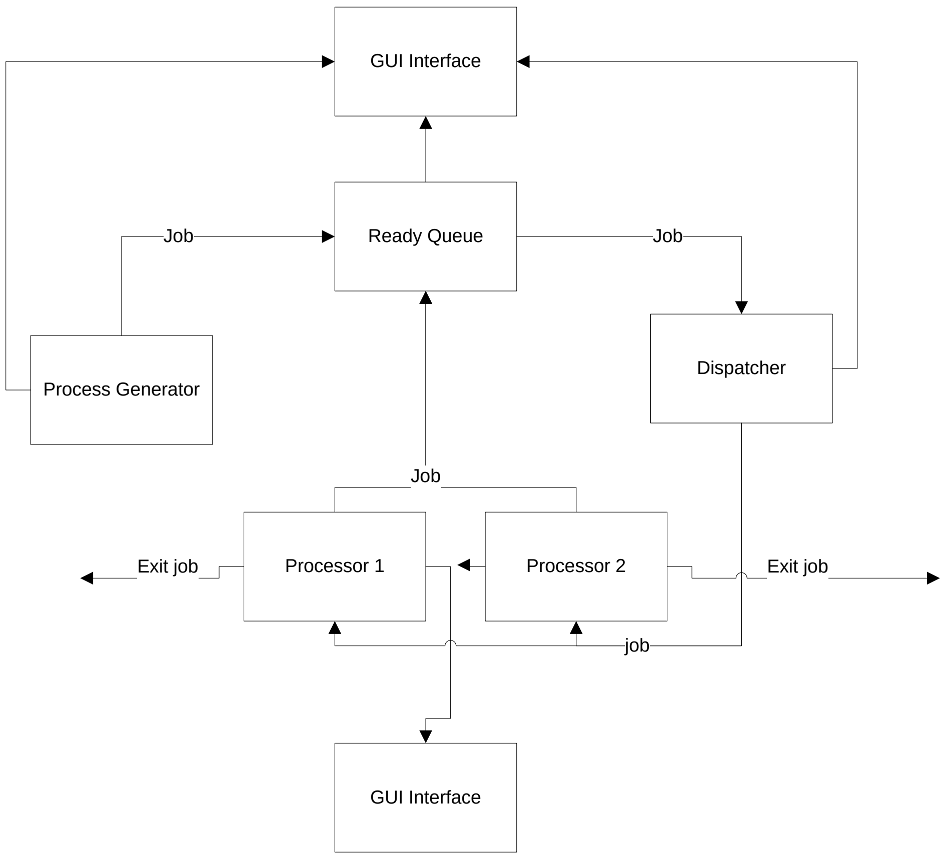
**Job**

1. a job is defined by a burst time, an ID,
2. Each time a job is executed, his burst time is decremented by one

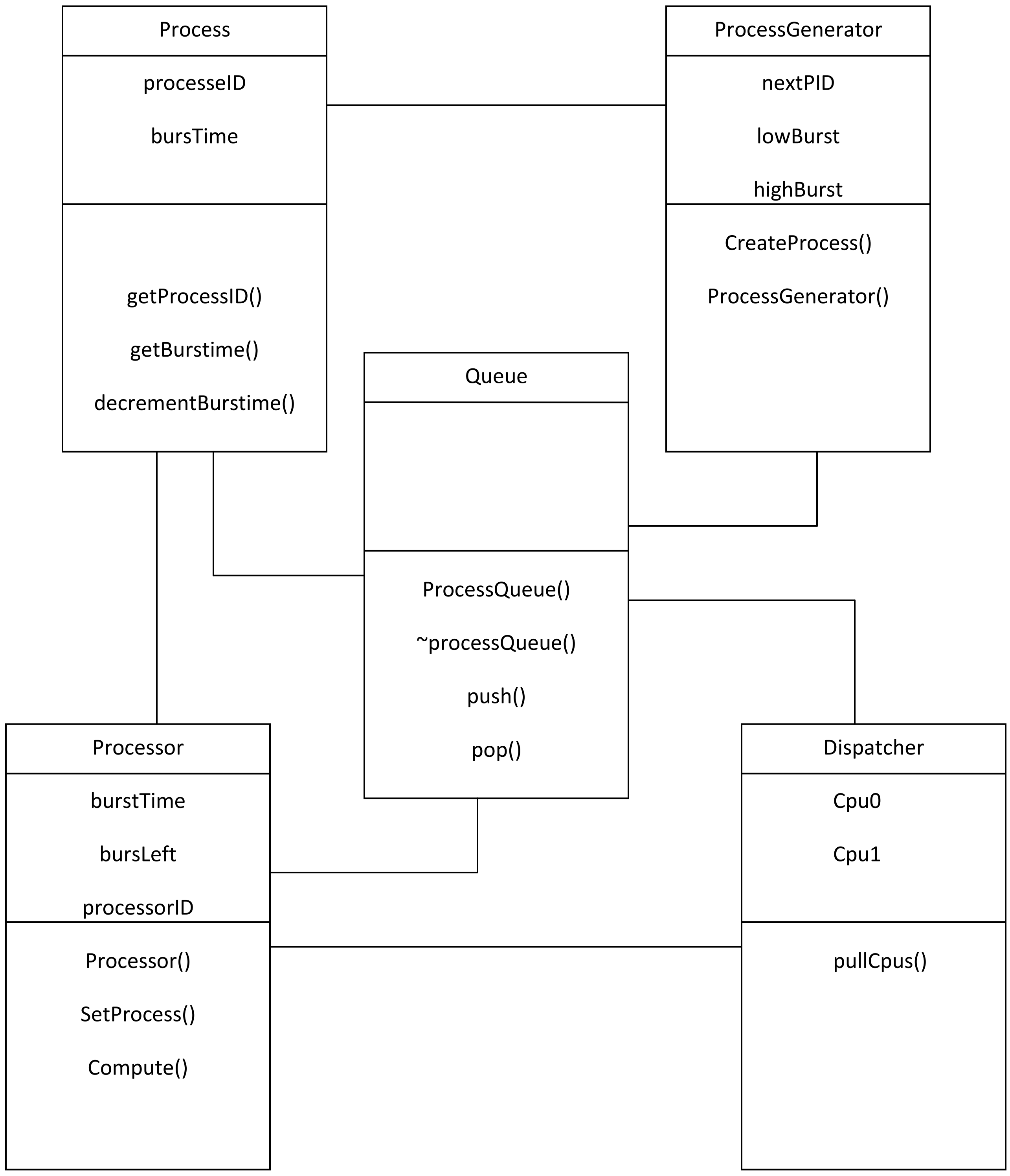
**Ready Queue**

1. incoming jobs are added to the tail of the queue
2. Jobs leave the ready queue from the top
3. the size of the queue is infinite

## 3. **Architectural Design**



## 4. Data / Class Design

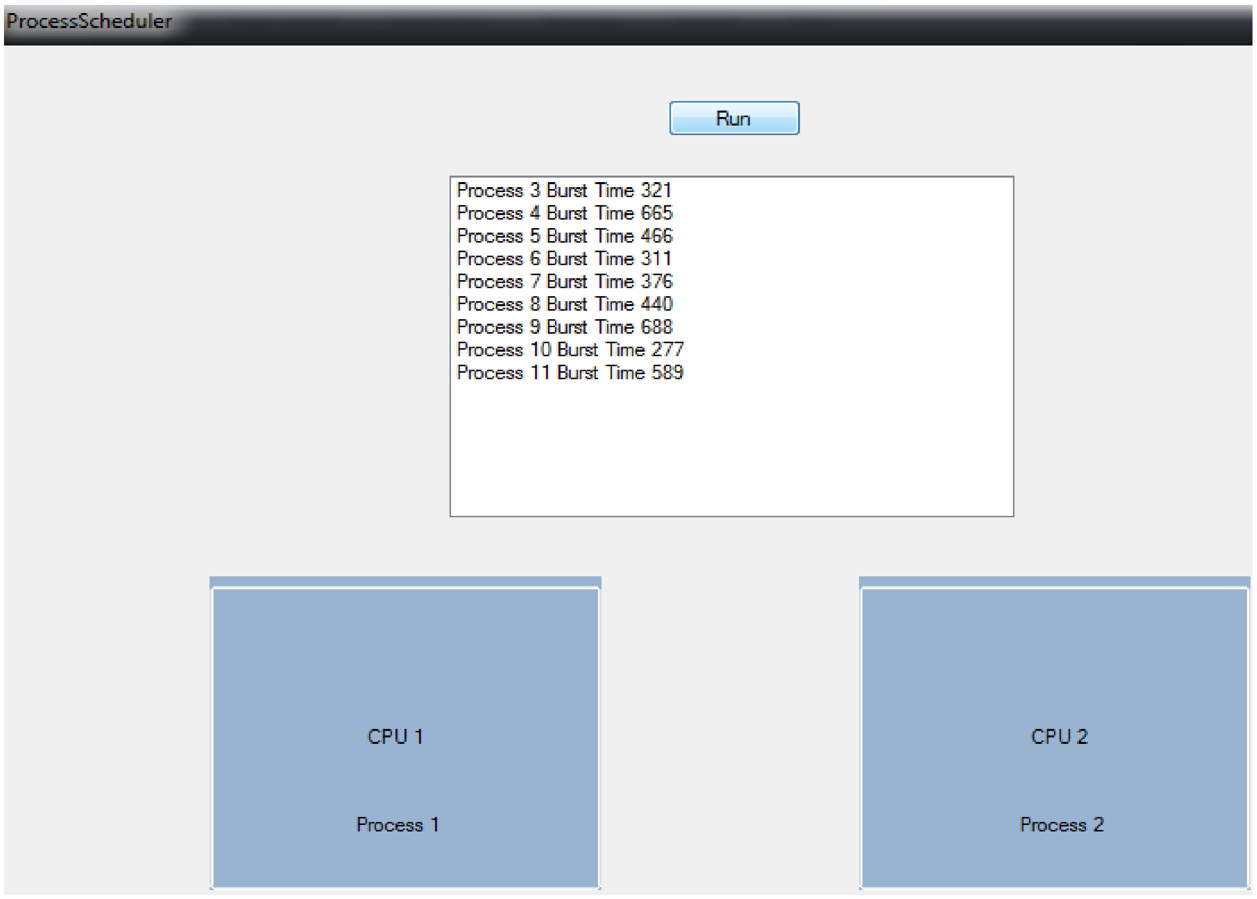


| **Class** | **Attributes** | **Methods** |
|---|---|---|
| **Process** | - **processeID**: type integer; identifies a unique process.<br>- **BurstTime**: type integer, define how long a process requires the CPU | **- GetProcessID ():** return a process ID<br>- **GetBurstTime** (): return process burst time<br>- **decrementBurstTime** (): decrement process burst time by 1<br>- **finished ():** Boolean method, check if a process is complete (burst time = 0) |
| **ProcessGenerator** | - **nextPID**: type integer<br>- **lowBurst**: type integer, minimum process burst time.<br>- **highBurst**: type integer, maximum process burst time | - **CreateProcess**(): create a process<br>- **ProcessGenerator**() |
| **Processor** | - **burstTime**: type integer, maximum time the cpu run each process<br>- **bursLeft**: cpu burst time left<br>- **processorID**: type integer, identifies a unique processor | - **Processor**(): create a processor<br>- **SetProcess**(): set the processor status<br>- **Compute**(): execute a process |
| **Dispatcher** | - **Cpu0**: type integer; reference to processor 1.<br>- **Cpu1**: type integer; reference to processor 2 | - **pullCpus**(): assign a process to a processor |
| Queue | | - ProcessQueue()<br>- ~processQueue()<br>- **push**(): add a process to the queue<br>- **pop**(): remove a process from the queue |

## E. Code design and Implementation

In this section of the paper, we will be discussing the design and implementation of the code for this project. The first step of the design and implementation was to choose a programming language on which we would build our project. We easily narrowed it down to java and c++, but decided to go with c++ for its simplicity and ease of design. Once we decided on c++, we needed to make a plan.

We then decided what type of output we were looking for. We decided that we wanted to have a user interface to visualize what was happening within our simulation of a dual core processor system. Since the scope of our simulation included the processors, the queue, and the means to get manage the processes, we decided that we needed to simply visualize the process queue and the 2 processors. The process queue is easily shown using a list box. The processes in the queue are shown in the order they are to be run, so the top left process is next and the bottom right process is last. We also decided that it would be beneficial to be able to tell which process is currently in which processor. For this we decided that a simple box containing the process number would suffice. With this, we were able to design the interface:



```
Process 3 Burst Time 321
Process 4 Burst Time 665
Process 5 Burst Time 466
Process 6 Burst Time 311
Process 7 Burst Time 376
Process 8 Burst Time 440
Process 9 Burst Time 688
Process 10 Burst Time 277
Process 11 Burst Time 589
```

We started the coding portion of the project by making a plan. With our plan we decided what classes we would need, what data we would need to have and create, and how everything would work together. First we wrote most of our ideas down on paper, drew some simple diagrams and pictures, and then got right into it. We created our header files first, giving us the basic structure for the program. This allowed us to decide what methods we needed, what parameters they needed to be passed, and also allowed us to split up the code amongst our group members.

We decided that our project would require 5 classes. We needed a class for the process (Proc.h), the processor (Processor.h), a process generator (ProcessGenerator.h), a dispatcher (Dispatcher.h), and the user interface (ProcessScheduler.h). I will first discuss each of these classes, going over what they do and how they do it. We also have 2 additional header files, which are not classes. The first is stdafx.h, which holds references to any header files we need within our project. This allows us to keep the #includes within the source files short. There is also a targetver.h file, which simply tells Windows what type of form to provide to the user. This is a built in function, and we did not modify it.

The first class I would like to discuss is the processor class. We decided to name it “Proc,” since process is reserved for system use. The proc class contains a couple of attributes and a few methods. The first attribute is an int called pid. This stores the process ID of the process and is set in the constructor. The second attribute is also an int and is named burstTime. Where a real process would have instructions to execute, our processes have the burst time. The burst time represents the number of CPU clock cycles it will take to complete. We also have several methods, getPID(), getBurstTime(), decrementBurstTime(), and finished(). The first two methods simply return their respective attributes. The decrementBurstTime() method subtracts 1 from the burst time. It is called once for each clock cycle that it is the process currently running in one of the processors. The last method, finished(), returns true if the burst time = 0 and false otherwise. We use this to check if the process has finished so we know to discard it or set it back on the queue.

```
class Proc
{
public:
    Proc();
    Proc(int processID, int burst);
    int getPID();;
    int getBurstTime();
    void decrementBurstTime();
    bool finished();

    int pid;
    int burstTime;
};
```

The next class is the processor, “Processor.h”. The processor has a few different attributes and methods. The first attribute is the processorID. This lets us know which processor it is. The boolean, ready, is set to true any time a process either completes or the burst time is surpassed. The processor also has a burstTime, which is the maximum amount of time a process can run before it must either complete or be sent back to the queue. This goes along with burstLeft, which is decremented each clock cycle and when it reaches 0, the CPU knows to pass the current process out. The attributes *q and p hold a reference to the process queue and the current process respectively. The first method is the constructor, which takes a reference to the process queue, along with the processor ID. There are also a few getter and setter methods, isFree(), setReady(), and setProcess(). These are used get or change the ready attribute, and to set the current process respectively. The last method is compute(). The compute method is what really does the work for our processor. It first decrements the process’ and its own burst times. Then it checks if either has reached 0. If its own burstLeft has reached 0, it will set itself to ready and send the process to the queue. If the process has completed it will simply set itself to ready, therefore discarding the current process when the new process is received.

```
class Processor
{
public:
    void setProcess(Proc incomingProc);
    System::String^ compute();
    bool isFree();
    void setReady(bool sr);;

    bool ready;
    int burstTime;
    int burstLeft;
    int processorID;
    queue<Proc> *q;
    Proc p;
    Processor(queue<Proc> *qu,int CPUID);
};
```

We next had the need for the process generator, “ProcessGenerator.h”. This class is very simple, it has 3 attributes and 3 methods. The first three attributes are references to the processors and the third attribute is a reference to the process queue. The first method is the constructor. The constructor is passed the above attributes and sets them appropriately. The second and third methods are pollCPU0() and pollCPU1(). These methods check if the processor is free, if it is not free, they return false.  If the CPU is free, the methods will first set the current process for the CPU to the first item in the process queue, then return true.

```
class Dispatcher
{
public:
    Processor *cpu0;
    Processor *cpu1;
    queue<Proc> *q;

    Dispatcher(queue<Proc> *qu, Processor *process0, Processor *process1);
    bool pollCPU0();
    bool pollCPU1();
};
```

The fourth class we needed is the process generator.  This class is used to create the processes which will be sent to the queue.  It has 4 attributes and 2 methods.  The first attribute is nextPID.  This holds the PID that will be used to create the next process.  The second attribute is lburst.  This is the low end of the process burst time that we will randomly generate.  The third is hburst.  As you can guess, it is the high end of the process burst time that we will randomly generate.  The last attribute is a reference to the process queue, since we will need to add the processes after creation.  The first method is the constructor.  It takes attributes 2-4 and assigns them.  It also seeds our random number generator with the time, which ensures some randomness within our program.  The other method is createProcess().  This method creates a process with the nextPID and a random number between lburst and hburst.  Once the process is created, this method increments nextPID and add the new process to the process queue.

```
class ProcessGenerator
{
public:
    int nextPID;
    int lburst;
    int hburst;
    queue<Proc> *q;

    ProcessGenerator(int lowBurst, int highBurst, queue<Proc> *qu);
    System::String^ createProcess();

};
```

Finally the last class created was Process Scheduler. This class functions as the main driver for the program as well as the GUI creation. In the header file ProcessScheduler.h, a form is created and populated with the components for the form itself. All of the components are created, allocated and defined on the header file and then manipulated in the .cpp file.  In the .cpp file lies the driver for the program in the function runButton_Click(sender, args), in this function we start by creating two processors via the Processor class. Next, we create the dispatcher and process generator as well. We populate the process queue with 10 processes to start and then run a series of checks on the processors. The checks we do are, if cpu0 is free then send it a process from the queue, and if cpu1 is free then send it a process from the queue. After that, we loop a while loop until the queue is empty where we periodically create new processes, run the CPU operations and give the processors new processes if they have completed their current process. Finally we account for when there is only 1 item left to process and one processor will be free while the other finishes the job. During the driver itself there are many updates to the GUI as well.

- **Source code:**

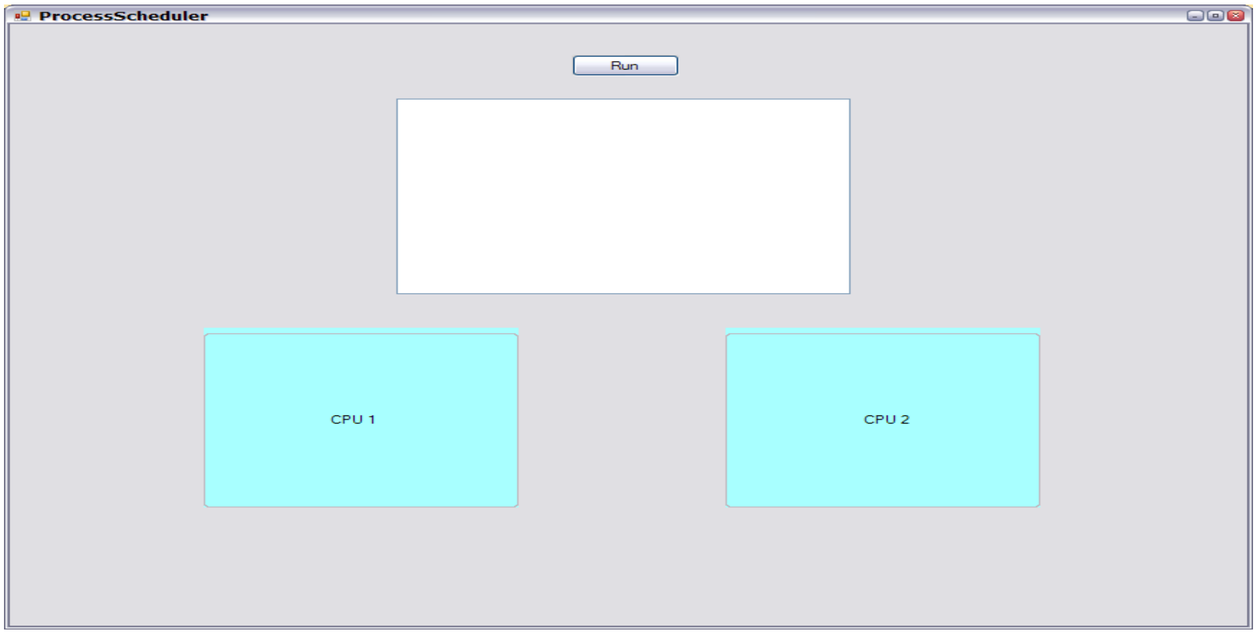
```
   Void main(){
   //create the 2 processors, the dispatcher, and the process generator
Processor cpu0 = Processor(&q,0);
   Processor cpu1 = Processor(&q,1);
   Dispatcher disp = Dispatcher(&q,&cpu0,&cpu1);
   ProcessGenerator pg = ProcessGenerator(250,850,&q);

   //create 10 processes to begin
   for(int i=0;i<10;i++){
                String^ temp = pg.createProcess();
                ListViewItem^ lv =  gcnew ListViewItem(temp);//gui
   qList->Items->Add(lv);//gui
   }
   this->Update();

   //gives each CPU a process
   bool cpu0Free = disp.pollCPU0();
   bool cpu1Free = disp.pollCPU1();

   if(cpu0Free)//gui
   {
   System::String^ temp = Convert::ToString(qList->Items[0]);
   int parseVal = temp->IndexOf("Process",0);
   System::String^ cpuPID = temp->Substring(23, 2);
   cpu1Lbl->Text = System::String::Concat("Process ", cpuPID);
   qList->Items->RemoveAt(0);
   this->Update();
   }
   if(cpu1Free)//gui
   {
   System::String^ temp = Convert::ToString(qList->Items[0]);
   int parseVal = temp->IndexOf("Process",0);
   System::String^ cpuPID = temp->Substring(23, 2);
   cpu2Lbl->Text = System::String::Concat("Process ", cpuPID);
   qList->Items->RemoveAt(0);
   this->Update();
   }
int counter = 0;
   //runs while the q has items
   while(!q.empty()){
   //generate process every 500 time, while the counter < runtime
   if(counter < RUNTIME){
                if(counter%300 == 0){
                             String^ temp = pg.createProcess();
                             ListViewItem^ lv =  gcnew ListViewItem(temp);//gui
                             qList->Items->Add(lv);//gui
                             this->Update();//gui
                }
   }
                //run 1 clock cycle, each cpu runs 1 "instruction"
   //and the dispatcher polls the CPU and sends them
                //a new process to run if necessary
   System::String^ cpu0Tmp = cpu0.compute();
   System::String^ cpu1Tmp = cpu1.compute();
   if(cpu0Tmp != "0")//gui
   {
   ListViewItem^ lv =  gcnew ListViewItem(cpu0Tmp);
   qList->Items->Add(lv);
   this->Update();
   }
```

```
		if(cpu1Tmp != "0")//gui
		{
		ListViewItem^ lv =  gcnew ListViewItem(cpu1Tmp);
		qList->Items->Add(lv);
		this->Update();
		}

		bool cpu0Free = disp.pollCPU0();
		bool cpu1Free = disp.pollCPU1();
		if(cpu0Free)//gui
		{
		System::String^ temp = Convert::ToString(qList->Items[0]);
		int parseVal = temp->IndexOf("Process",0);
		System::String^ cpuPID = temp->Substring(23, 2);
		cpu1Lbl->Text = System::String::Concat("Process ", cpuPID);
		qList->Items->RemoveAt(0);
		this->Update();
		}
		if(cpu1Free)//gui
		{
		System::String^ temp = Convert::ToString(qList->Items[0]);
		int parseVal = temp->IndexOf("Process",0);
		System::String^ cpuPID = temp->Substring(23, 2);
		cpu2Lbl->Text = System::String::Concat("Process ", cpuPID);
		qList->Items->RemoveAt(0);
		this->Update();
		}
		counter += 1;
		//slows down the output so it is readable.
		Sleep(10);
		}

		//allows cpu's to finish the processes they are running
		//once the q is empty
		while(!cpu0.isFree() || !cpu1.isFree()){
			if(!cpu0.isFree()){
				cpu0.compute();
			}
			if(!cpu1.isFree()){
				cpu1.compute();
			}
			Sleep(10);
		}
}
```

## F. Simulation and Results

In this section of the paper, we will be running the program for simulation purposes and discuss about the results of the simulation. To achieve our goals, we will be taking screen shots of the running program at different phases of execution and discuss the obtained results. For this simulation, we run the program on a workstation powered by Intel® Core™2 processors with vPro™ technology, with 3Gbytes of memory,  and running on Windows XP. Note that our program is self driven because no input is provided to the program. As mentioned earlier, we do not intend to reproduce the entire process management system in all details. In that way we focus on the life cycle of a process from creation to complete execution highlighting multiple interactions that occurs between the dispatcher, the ready queue, the two processors, and memory.

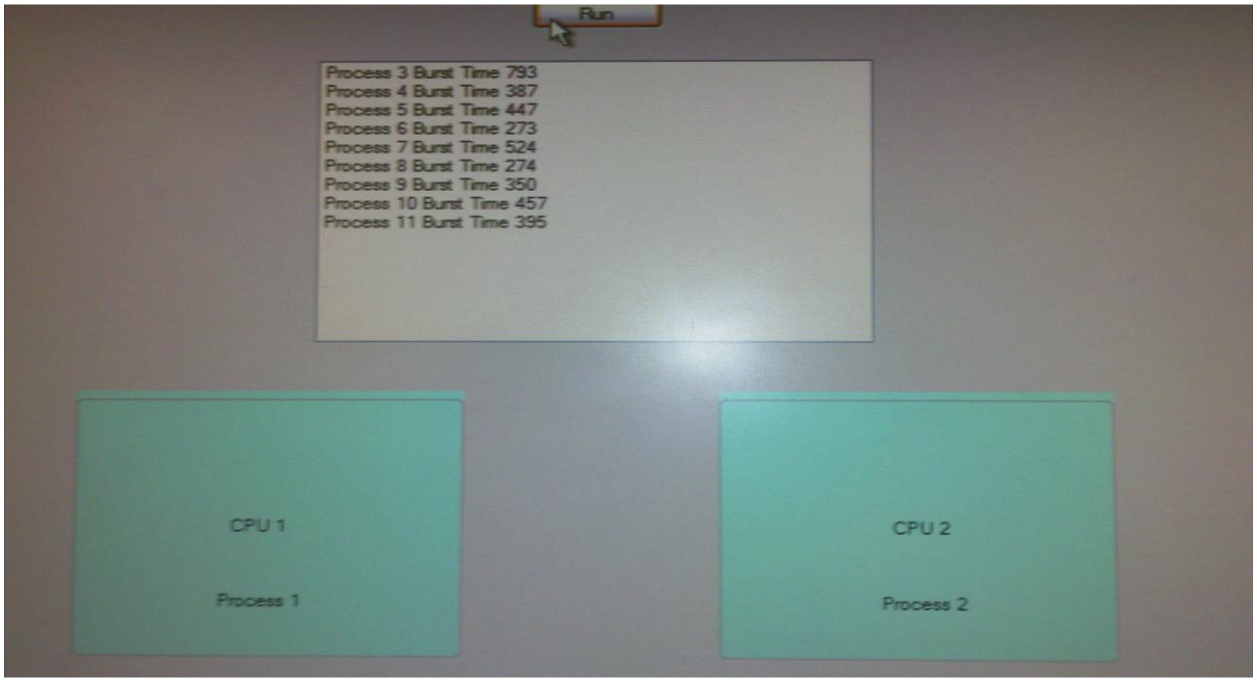
- Let’s start our simulation with a screen shot of the GUI interface without running the program:



*****1st screen shot*****

The GUI interface is a representation of our ProcessScheduler and it is keeping track of the contents of our ReadyQueue (the queue that holds all processes waiting to be assigned to a CPU for processing), and two CPUs.

- Next, we take a screen shot of the GUI interface showing what is our program doing when we first launch it:

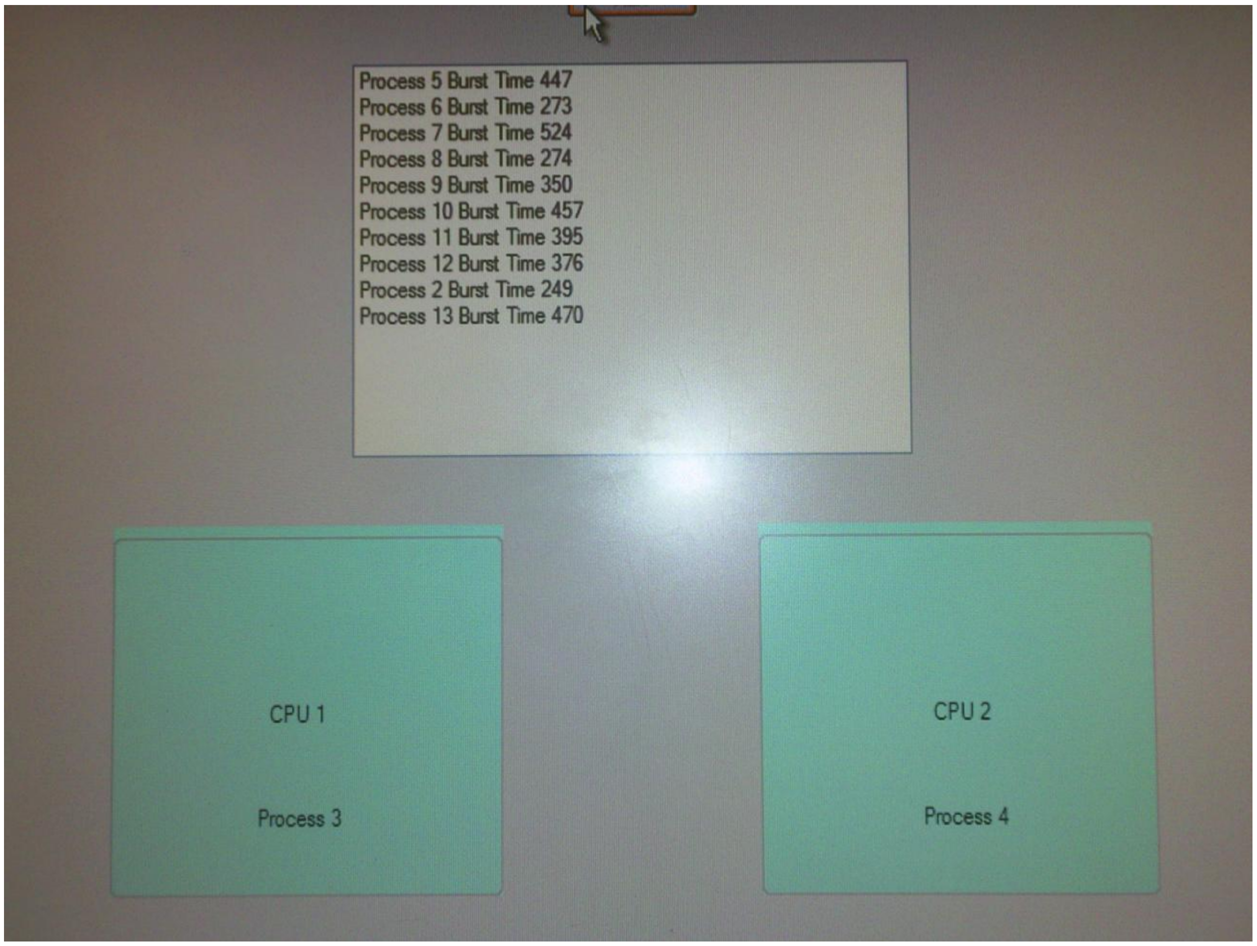


```
Process 3 Burst Time 793
Process 4 Burst Time 387
Process 5 Burst Time 447
Process 6 Burst Time 273
Process 7 Burst Time 524
Process 8 Burst Time 274
Process 9 Burst Time 350
Process 10 Burst Time 457
Process 11 Burst Time 395
```

*****2nd screen shot*****

At start, the program automatically generates 10 processes with random burst time values between 250 and 850 to ensure that our processes don't have the same burst time values. You can see from the screen shot that Process 1 and Process 2 are automatically assigned to CPU 1 and CPU 2 respectively since they are both free initially so each of them can take one process. Note that the ProcessScheduler will generate a new process and add it to the ReadyQueue every 500 of unit time while the RUNTIME (declared to be 5000 in this simulation) of our program is not equal to zero. That is the reason why we have a Process 11 at the bottom of the ReadyQueue.

- Next, we take a screen shot of the GUI interface after Process 1 and Process 2 are done executing on CPU 1 and CPU 2:



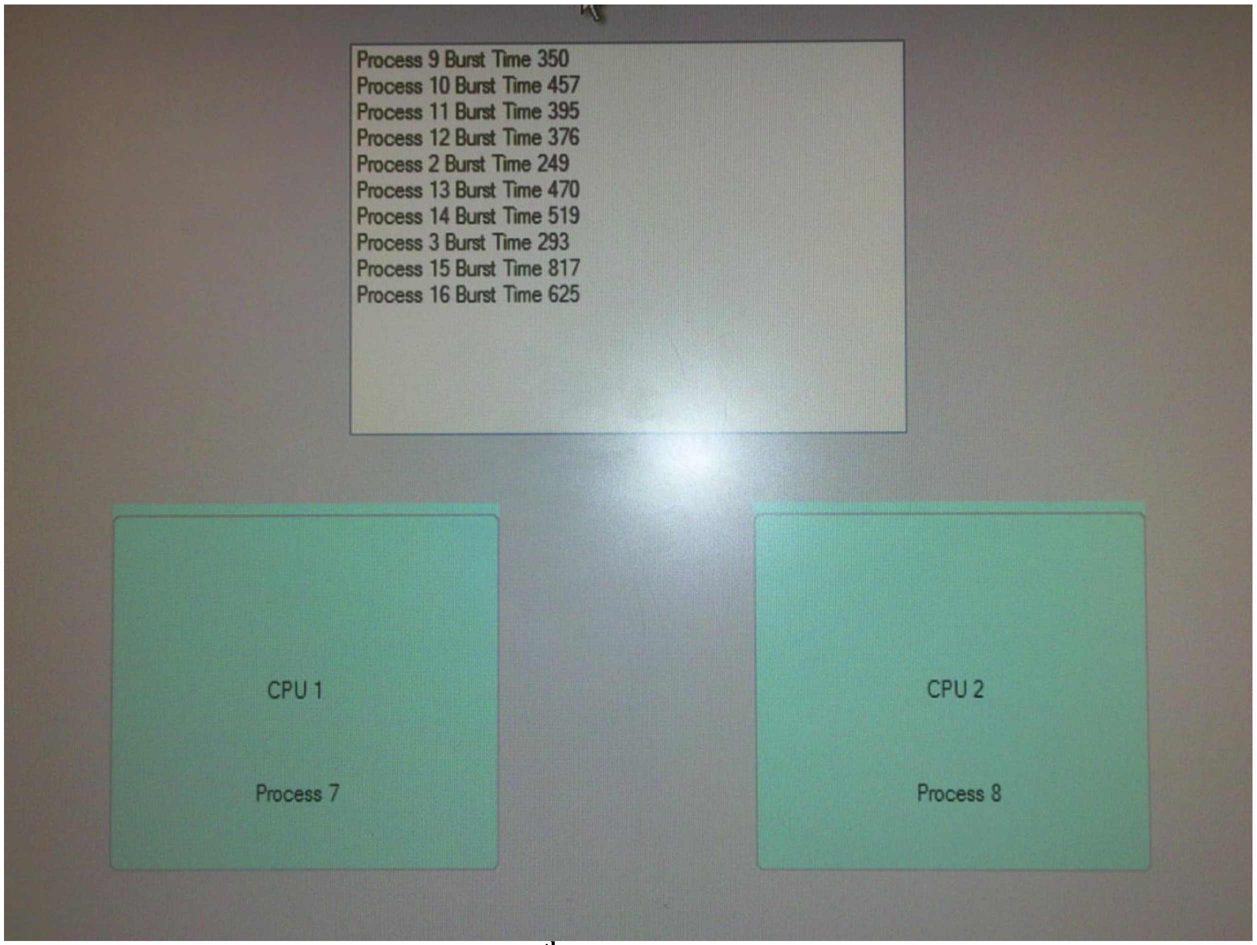
```
Process 5 Burst Time 447
Process 6 Burst Time 273
Process 7 Burst Time 524
Process 8 Burst Time 274
Process 9 Burst Time 350
Process 10 Burst Time 457
Process 11 Burst Time 395
Process 12 Burst Time 376
Process 2 Burst Time 249
Process 13 Burst Time 470
```

*****3rd screen shot*****

From the screen shot, we can see that Process 3 and Process 4 have been assigned to CPU 1 and CPU 2 respectively. This means that the burst time for Process 1 was less than the burst time for Process 2 that is why CPU 1 was assigned Process 3 by the Dispatcher because CPU 1 was free to take the top process available from the ReadyQueue before CPU 2 has done processing Process 2. In that way, when CPU 2 was done executing Process 2, it was assigned Process 4 by the Dispatcher because it was the top process available from the ReadyQueue at that time.

From the screen shot, we can also see that Process 2 is also listed in the ReadyQueue with a burst time of 249. This means that Process 2 did not finish processing while in CPU 2 because it had a burst time greater than 500 which is a limit we have set for processes' computation time in the CPUs. In that way, Process 2 which had originally a burst time of 749 (500 + 249), has to be taken out from CPU 2 because it has been processed for 500 unit time and added to the tail of the ReadyQueue with a new burst time of 249 (749 – 500). We can also see that new processes are kept being added to the ReadyQueue after each 500 of unit time where the presence of Process 12 and Process 13.

- Next, we take a screen shot of the GUI interface after Process 5 and Process 6 are done executing:



```
Process 9 Burst Time 350
Process 10 Burst Time 457
Process 11 Burst Time 395
Process 12 Burst Time 376
Process 2 Burst Time 249
Process 13 Burst Time 470
Process 14 Burst Time 519
Process 3 Burst Time 293
Process 15 Burst Time 817
Process 16 Burst Time 625
```

CPU 1

Process 7

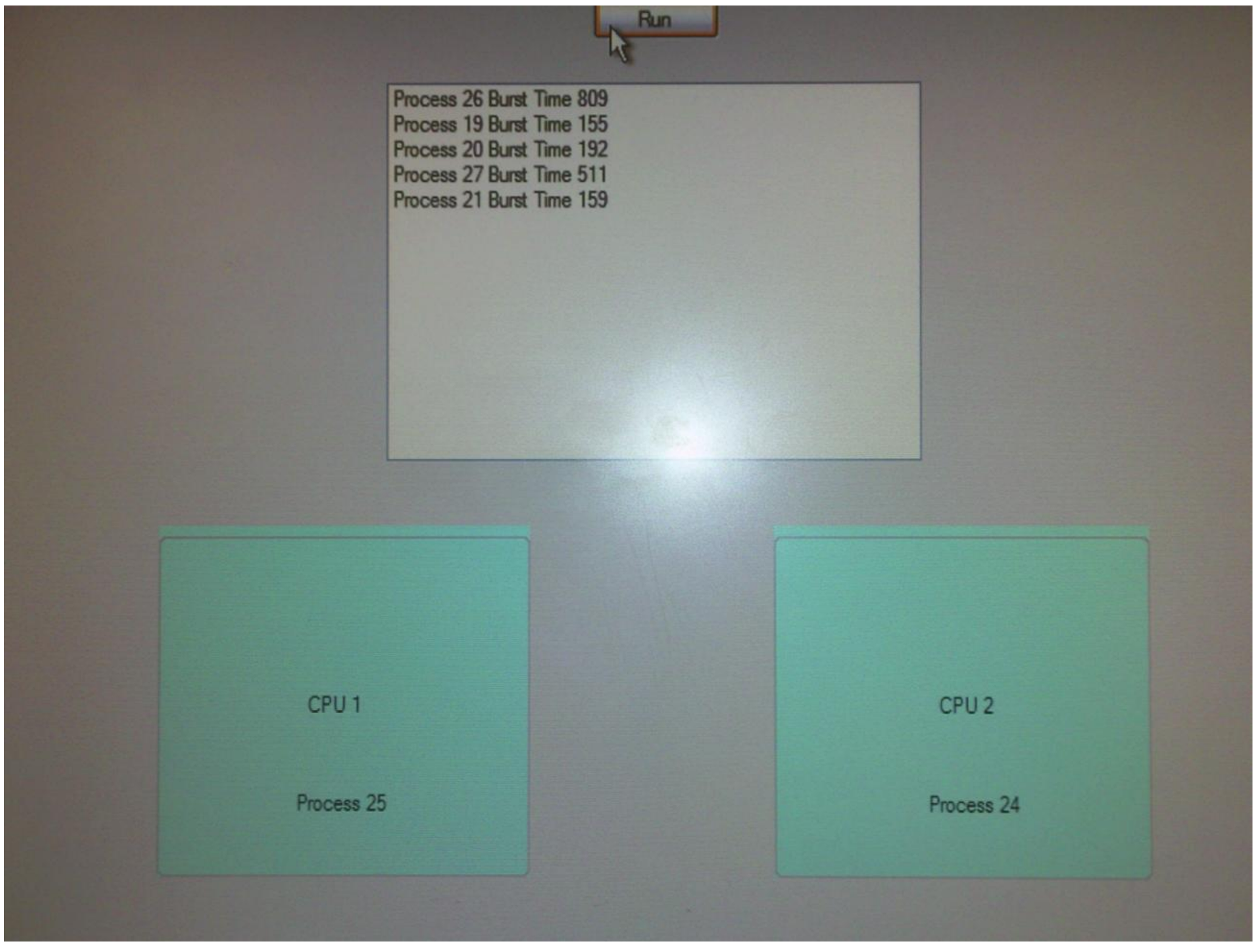
CPU 2

Process 8

*****4th screen shot*****

From the screen shot, we can see that Process 7 and Process 8 have been assigned to CPU 1 and CPU 2 respectively and that the ProcessScheduler keeps adding new processes after each 500 of unit time where the presence of Process 14, Process 15, and Process 16 in the ReadyQueue. Just like we had to take out Process 2 from CPU 2 because its total burst time was greater than 500 and could not complete its execution within that period of time while in CPU 2 and thus got added back to the ReadyQueue; the same happened with Process 3. It was originally processed in CPU 1 but had to be taken out and put back to the ReadyQueue with a new burst time of 293 (we can see from the first screen shot that the original burst time of process 3 was 793). Following this logic, all processes that have a burst time greater than 500 will have to be swapped out from the CPU and put back into the ReadyQueue.

- Next, we take a screen shot of the GUI interface after the RUNTIME of the program has reached 0:

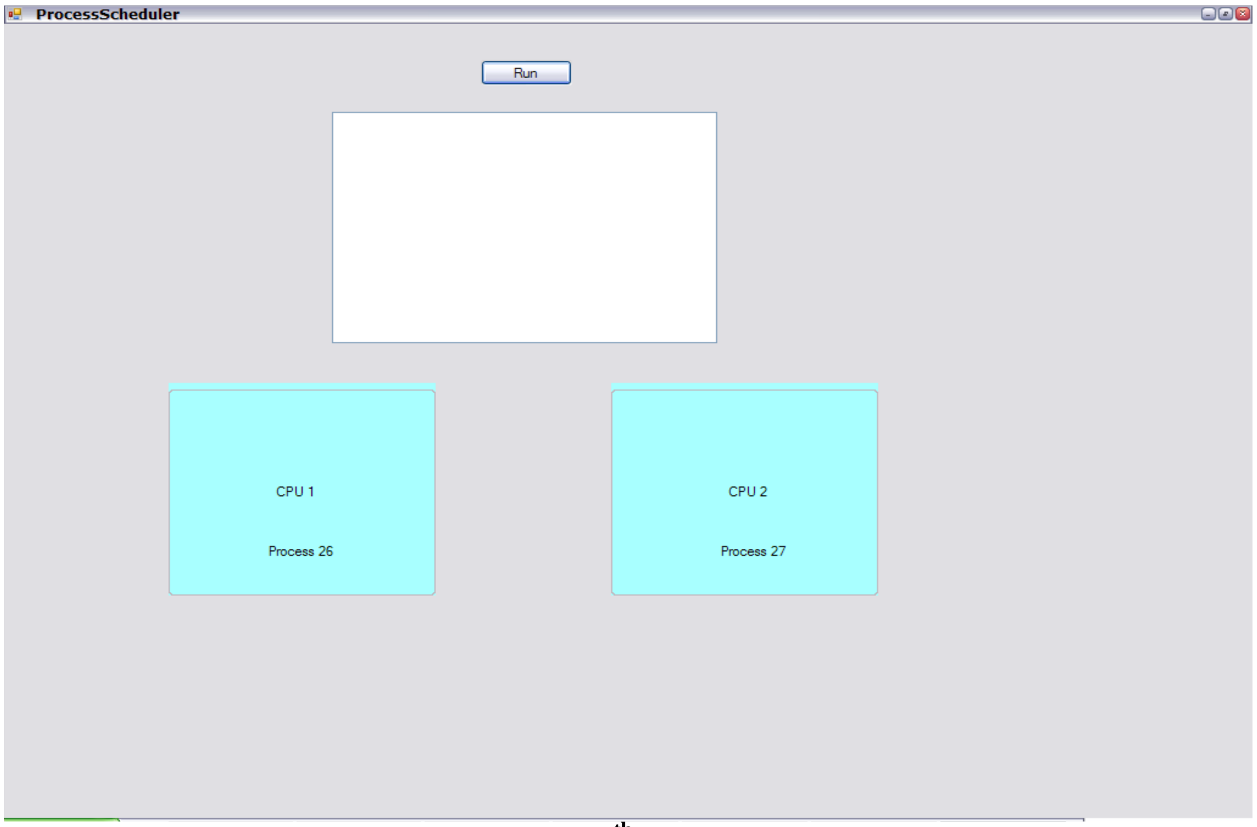


```
Process 26 Burst Time 809
Process 19 Burst Time 155
Process 20 Burst Time 192
Process 27 Burst Time 511
Process 21 Burst Time 159
```

*****5th screen shot*****

From the screen shot, we can see that only five processes are left in the ReadyQueue. This means that the RUNTIME of the program has reached 0 and that the ProcessScheduler will no longer be producing any more processes and that only the processes left in the ReadyQueue need to be processed by the CPUs. But note that even though the ProcessScheduler has stopped producing processes at this time, all the processes remaining in the ReadyQueue that have a burst time greater than 500 like Process 26 (burst time = 809) and Process 27 (burst time = 511) will have to be brought back to the ReadyQueue after their execution time in the CPUs expires after 500. In that way, the new burst time after first execution in CPUs of Process 26 and Process 27 will be 309 and 11 respectively.

- Next, we take a screen shot of the GUI interface after the ReadyQueue got empty and the simulation stops:



*****6th screen shot*****

From the screen shot, we can see that the ReadyQueue is now empty and that the last processes to be processed were Process 26 by CPU 1 and Process 27 by CPU 2 making them the last processes available from the ReadyQueue just like we have predicted in the discussion of the results we got from screen shot 5.

## G. Conclusion

By doing this project, we became familiar with the implementation and life cycle of a process from creation to complete execution through the interactions that occur between operating system components. Created from scratch and without any reference, our simulation turned out to be a success in terms of our goals because we were able to demonstrate how an OS manages processes in terms of scheduling. The processes do not actually perform any specific job but our program can be extended to support IPC in order to ensure Mutual Exclusion and avoid Deadlocks and Starvation in case any particular job need to be executed and requires some resources.