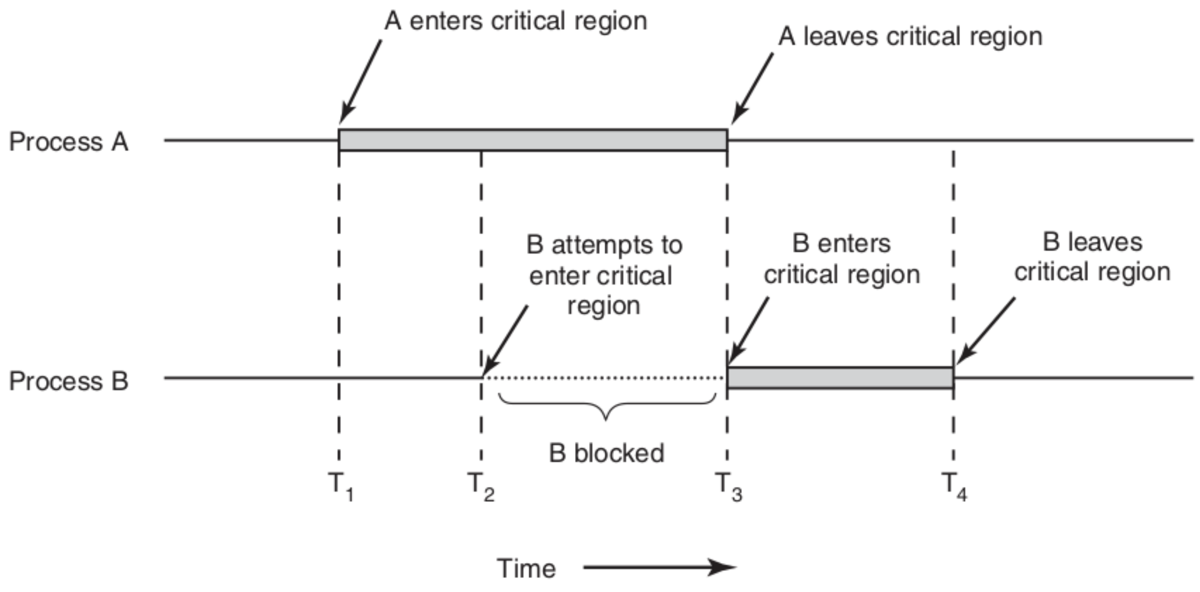
Homework #2
Mutual Exclusion Method Simulator

1. Introduction

Processes often need to communicate with each other, pass information between them or access some shared resources. This homework deals with the last case and how to prevent processes from creating a **Race Condition**, a state where two or more processes are reading or writing some shared data and the final result depends on who runs precisely when. To prevent a race condition, the **Mutual Exclusion** method is used which makes sure that only one process accesses a shared resource at a time, this is known as the **Critical Region** of the process, and the mutual exclusion method keeps any two processes from being in their critical regions at the same time. The mutual exclusion method has four conditions:

1. No two processes may be in their critical regions at the same time.
2. No assumptions may be made about speed or the number of CPUs.
3. No process running outside its critical region may block any other process.
4. No process should have to wait forever to enter its critical region.

The following diagram presents the mutual exclusion method in action at T_2 by preventing Process B from entering its critical region due to the fact that Process A is currently in its own critical region.



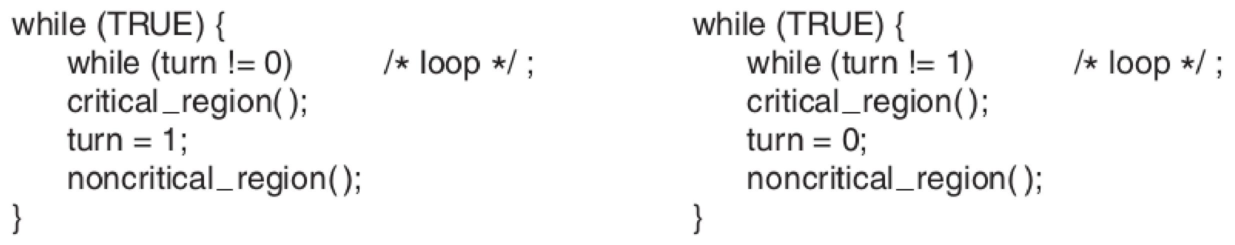
There are many different ways to implement the mutual exclusion method such as Disabling Interrupts, Lock Variables, Strict Alternation, Peterson's Solution, and The TSL Instruction just to name a few. However, my program only focuses on the Lock Variable and the Strict Alternation implementations.

2. Program Description

My program was written on Java and uses JavaFX for the graphic user interface. It is a simulation of the Lock Variable and Strict Alternation implementations of the mutual exclusion method. The scheduling algorithm used is the **Round Robin** algorithm implemented with a queue data structure in order to know which process' turn is next.

The **Lock Variable** implementation uses a Boolean variable as the "lock" for the critical region. When a process wants to enter its critical region, it first checks the lock. If the lock is not set, meaning the variable is false, the process sets the lock (variable is set to true) and it enters its critical region. When the process is done with its critical region, it removes the lock so the next process can enter its critical region if it needs to. If a turn is over, meaning that the quantum time/CPU time given to the process is over, and the process is still in its critical region, it will stay there blocking any other processes that want to enter their critical regions and resume execution of its critical region once it is its turn again.

The **Strict Alternation** implementation uses an integer variable to keep track of whose turn it is to receive quantum time. With this implementation, processes are constantly testing the turn variable to see if it is their turn to execute instructions. The constant checking is achieved with a while loop that does nothing but keep the process in a **Busy Waiting** state, a state when a program is continuously testing a variable for some value to appear. The following code demonstrates how the Strict Alternation algorithm is implemented:

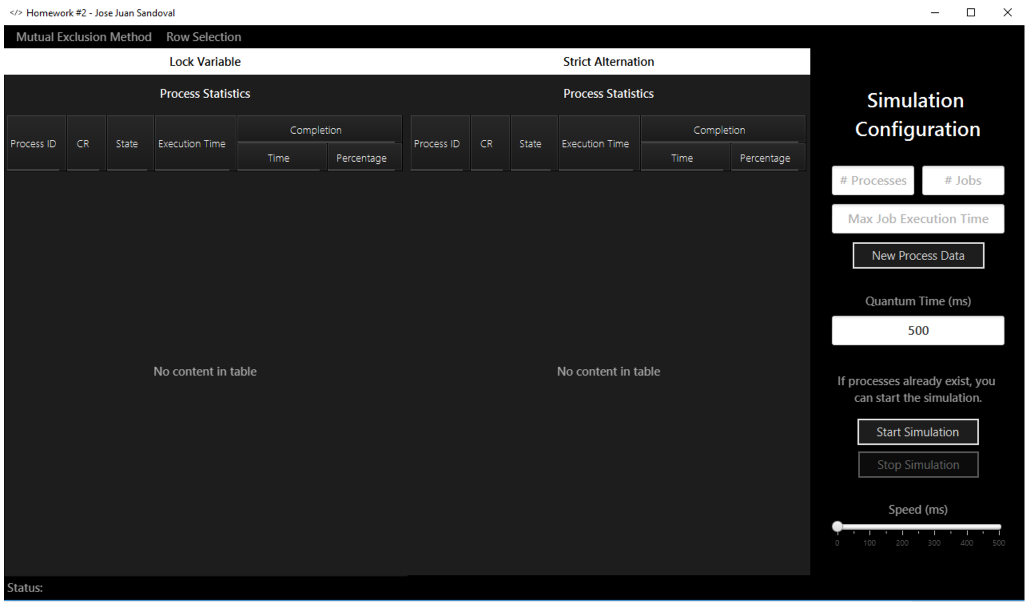


```
while (TRUE) {                                  while (TRUE) {
    while (turn != 0)      /* loop */ ;             while (turn != 1)      /* loop */ ;
    critical_region( );                             critical_region( );
    turn = 1;                                       turn = 0;
    noncritical_region( );                          noncritical_region( );
}                                               }
```

(a) (b)

As it can be seen, Process A (on the left) waits for the turn to become 0 in order to exit the busy waiting state. Process B does the same but instead waits for the turn to be 1. Once a process is out of its waiting state, it executes its critical region and when it is done it hands control over to the next process. It then goes to execute its non-critical region and goes back to the busy waiting state. With this implementation, I believe the quantum time is not used because the process who enters its critical region is actually hogging the CPU time until its done. The Strict Alternation implementation is often avoided because the busy waiting loop wastes CPU cycles and this is noticeable in my program when there are too many processes. Beware that running the simulation with this implementation activated will drive your CPU utilization high and it may even lock your OS. I tested with 10 processes.

3. Program Execution



No content in table

No content in table

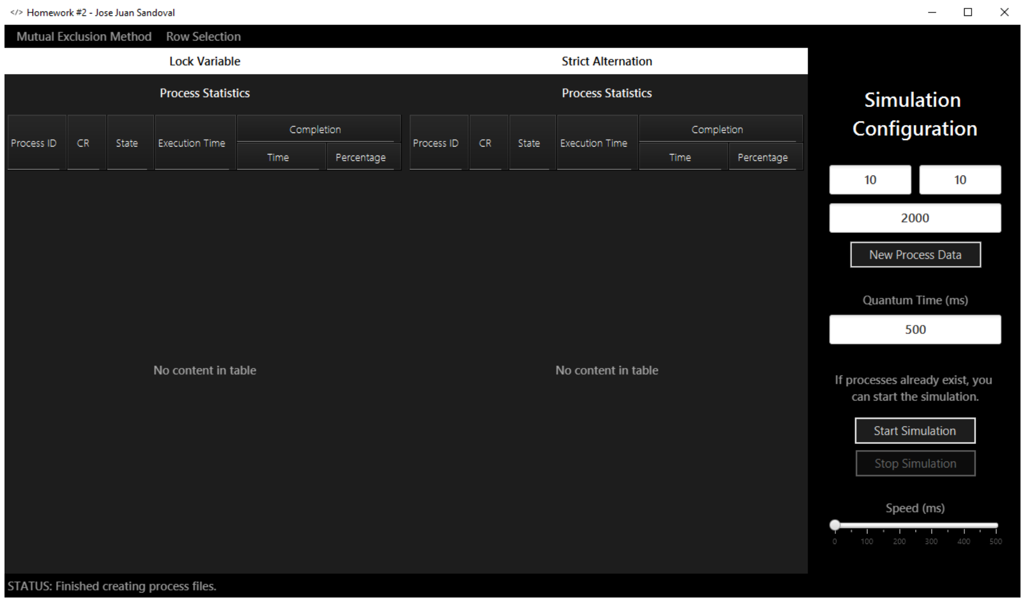
Quantum Time (ms)

500

If processes already exist, you can start the simulation.

Status:

When you launch the program this is the screen that will open up. On the left you will see the Process Statistics for each process with each Mutual Exclusion implementation side by side.



No content in table

No content in table

10 10

2000

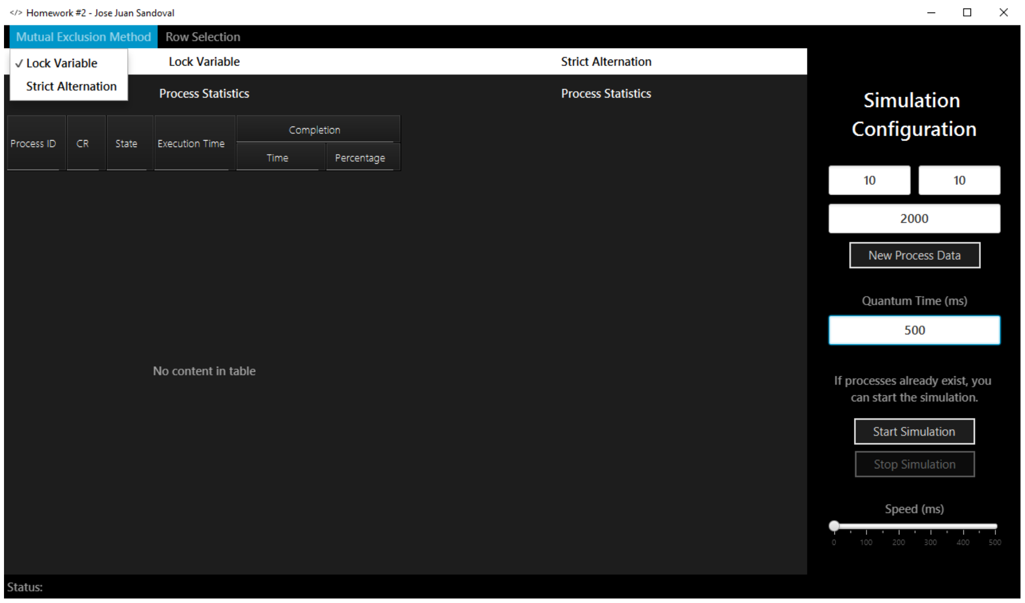
Quantum Time (ms)

500

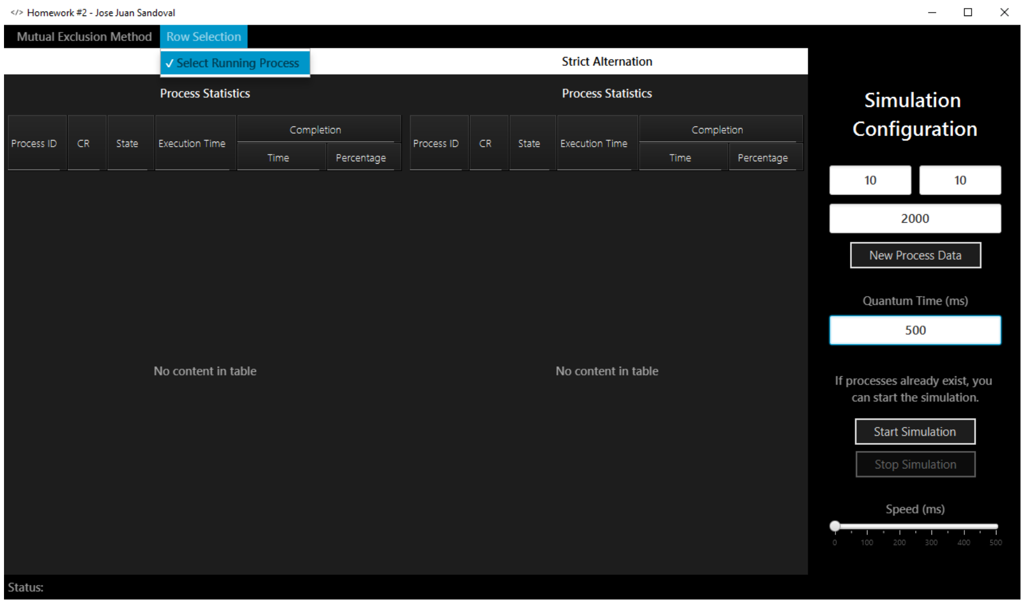
If processes already exist, you can start the simulation.

STATUS: Finished creating process files.

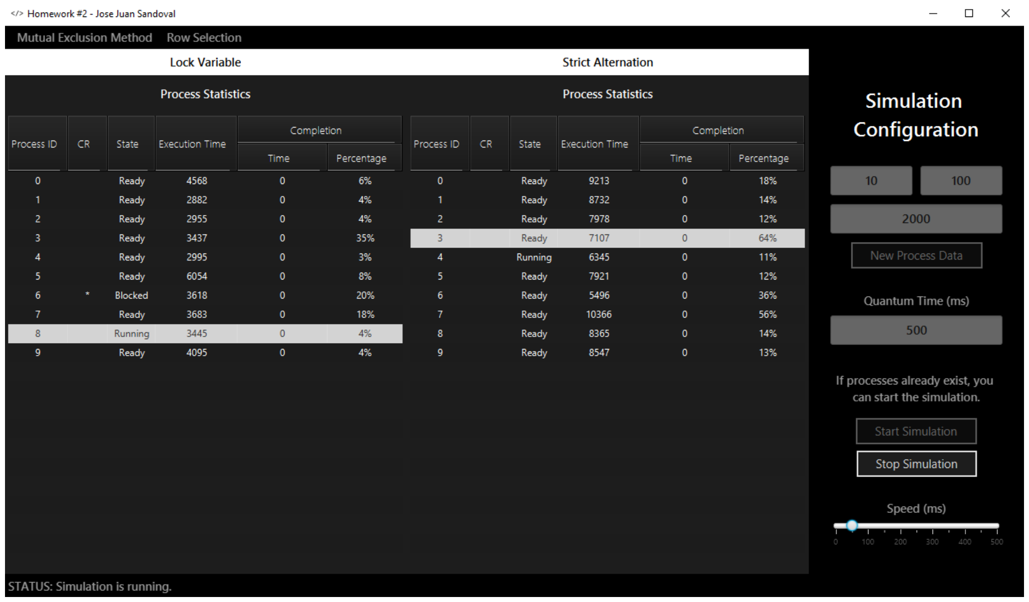
On the right you will see controllers for creating new process data for the simulation and controllers to configure the CPU's Quantum Time and the Speed of the simulation in real milliseconds. On the screenshot above, you can see that I created 10 processes, with a maximum of 10 jobs/instructions per process and with a maximum of 2000 milliseconds as the job's execution time.



On the top you can choose which Mutual Exclusion implementation you would like to simulate. The two options are the Lock Variable and Strict Alternation implementations as mentioned before.



Also on the top, you can choose whether to disable row selection for the currently running process.



Lock Variable

Process Statistics

Process ID	CR	State	Execution Time	Completion	
				Time	Percentage
0		Ready	4568	0	6%
1		Ready	2882	0	4%
2		Ready	2955	0	4%
3		Ready	3437	0	35%
4		Ready	2995	0	3%
5		Ready	6054	0	8%
6	*	Blocked	3618	0	20%
7		Ready	3683	0	18%
8		Running	3445	0	4%
9		Ready	4095	0	4%

Strict Alternation

Process Statistics

Process ID	CR	State	Execution Time	Completion	
				Time	Percentage
0		Ready	9213	0	18%
1		Ready	8732	0	14%
2		Ready	7978	0	12%
3		Ready	7107	0	64%
4		Running	6345	0	11%
5		Ready	7921	0	12%
6		Ready	5496	0	36%
7		Ready	10366	0	56%
8		Ready	8365	0	14%
9		Ready	8547	0	13%

If processes already exist, you can start the simulation.

STATUS: Simulation is running.

Finally, you can click the Start Simulation button and the simulation will start with the configuration provided. When the simulation is running the configuration controllers will be disabled and re-enabled once the simulation has finished. In the screenshot above, you can see the simulation running and with data being updated on the tables. You can use the Speed slider to speed up or slow down the simulation. Setting the slider to 0 will execute the program immediately. If you need to abort the simulation, you may use the Stop Simulation button. On the bottom you may see status updates or errors marked in red.

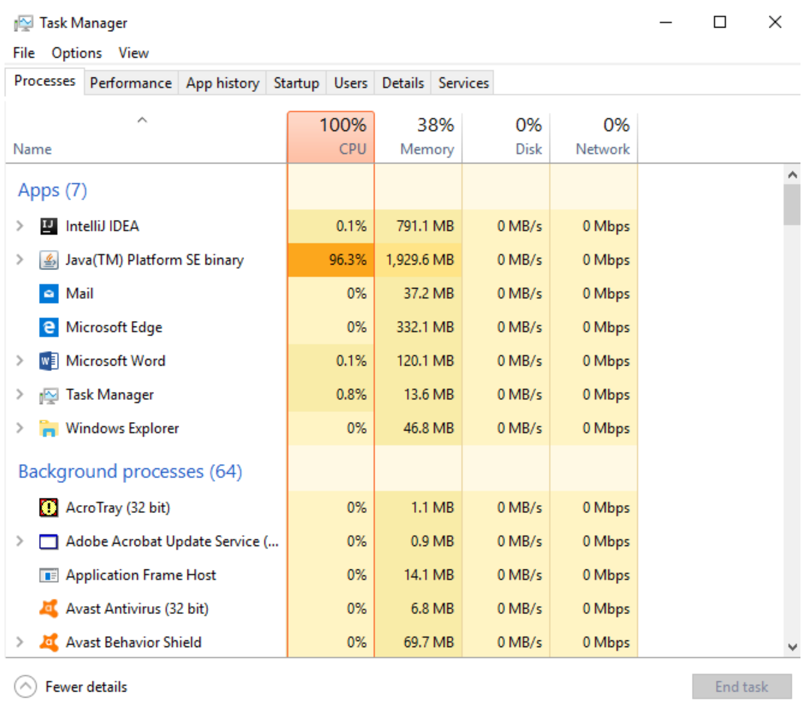
4. Programming Approach

To simulate the mutual exclusion methods, I followed the instructions provided in the lectures and implemented them as explained in section two. However, because I created a GUI for this program, the simulation is run in a separate thread from the GUI. The Lock Variable implementation runs in a single thread. The Strict Alternation implementation, however, runs each process in its own thread because I did not know how to instantiate many busy waiting while loops and keep them all in the same thread. I do not even know if it is possible at all due to sequential execution. So if you have 10 processes, 10 threads will be created to run the Strict Alternation simulation. The turn variable is marked as static and volatile so all threads can have access to it.

Besides multithreading, I also implemented the Observer design pattern to know when all threads had finished running so the GUI could be updated by re-enabling the required buttons.

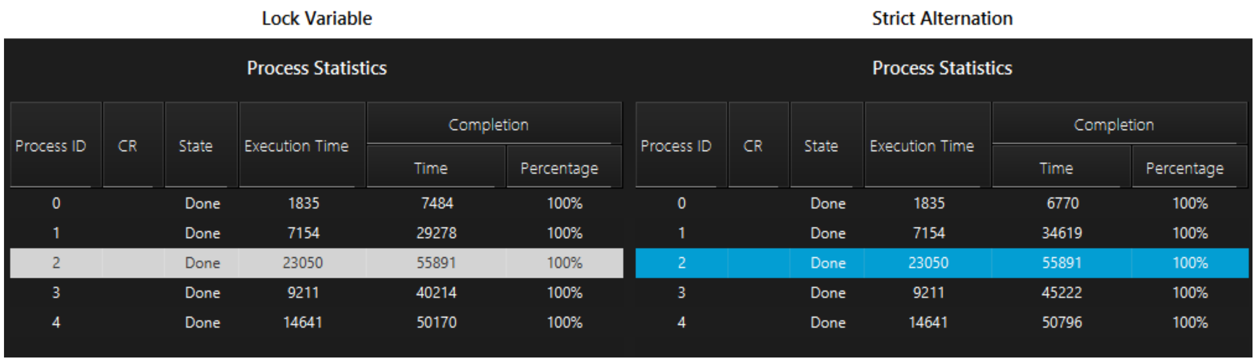
5. Findings/Outputs

I found out the hard way why the Strict Alternation method is often avoided. The busy waiting loop does waste so much CPU cycles that my CPU utilization shot up to 100% when I was trying to simulate many big processes. I let it run like that and after a few minutes my Windows crashed with a Driver Power State Failure error. I was running the program in my laptop and it was not connected to an outlet. I believed the CPU required too much power and the laptop was not able to provide it making the OS crash. The screenshot below shows the CPU utilization at its highest. So BEWARE I do not recommend running the simulation with the Strict Alternation implementation enabled if you are testing many big processes but if your are testing a few small processes then it works fine. One way that the performance of this simulation could be improved can be by implementing the Observer pattern for the turn variable so instead of wasting CPU cycles in the busy waiting loop, the processes can just wait to be notified that it is their turn but I will implement this later.


Task Manager
File Options View
Processes Performance App history Startup Users Details Services
Fewer details
End task

Name	100% CPU	38% Memory	0% Disk	0% Network
Apps (7)				
IntelliJ IDEA	0.1%	791.1 MB	0 MB/s	0 Mbps
Java(TM) Platform SE binary	96.3%	1,929.6 MB	0 MB/s	0 Mbps
Mail	0%	37.2 MB	0 MB/s	0 Mbps
Microsoft Edge	0%	332.1 MB	0 MB/s	0 Mbps
Microsoft Word	0.1%	120.1 MB	0 MB/s	0 Mbps
Task Manager	0.8%	13.6 MB	0 MB/s	0 Mbps
Windows Explorer	0%	46.8 MB	0 MB/s	0 Mbps
Background processes (64)				
AcroTray (32 bit)	0%	1.1 MB	0 MB/s	0 Mbps
Adobe Acrobat Update Service (...	0%	0.9 MB	0 MB/s	0 Mbps
Application Frame Host	0%	14.1 MB	0 MB/s	0 Mbps
Avast Antivirus (32 bit)	0%	6.8 MB	0 MB/s	0 Mbps
Avast Behavior Shield	0%	69.7 MB	0 MB/s	0 Mbps

We were required to create files with data that would create a race condition but I do not think it is possible to do so because the program’s purpose is to prevent that. So you can see processes get blocked but the program continues execution because the Mutual Exclusion methods are actually working and doing what they are supposed to.



Lock Variable

Process Statistics

Process ID	CR	State	Execution Time	Completion	
				Time	Percentage
0		Done	1835	7484	100%
1		Done	7154	29278	100%
2		Done	23050	55891	100%
3		Done	9211	40214	100%
4		Done	14641	50170	100%

Strict Alternation

Process Statistics

Process ID	CR	State	Execution Time	Completion	
				Time	Percentage
0		Done	1835	6770	100%
1		Done	7154	34619	100%
2		Done	23050	55891	100%
3		Done	9211	45222	100%
4		Done	14641	50796	100%

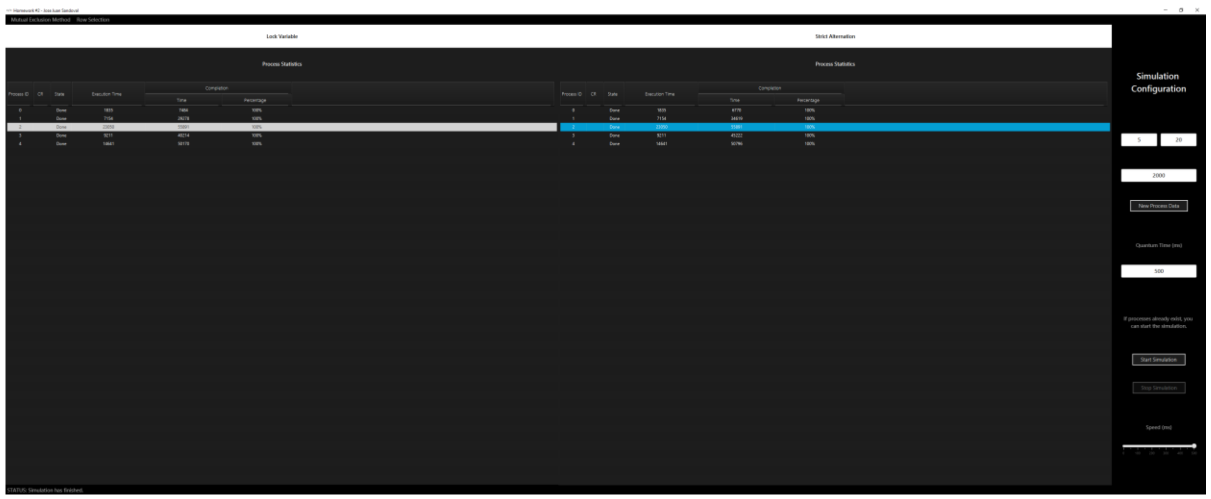
On average I found that the Lock Variable implementation produces lower completion times than the Strict Alternation implementation. From the five processes simulated and shown above, the Lock Variable had processes 1 and 3 that were completed considerably faster than their counterparts in the Strict Alternation table. Only process 0 completes faster in the Strict Alternation method. The last process to complete always takes the same amount of time because it is the same data. And process 4 is about the same in both implementations.

6. Future Plans and Bugs to Fix

I plan to expand this to include more scheduling algorithms besides Round Robin and more Mutual Exclusion methods. I also want to add more statistics like wait time for each process and average for all. And finally I would like to add more visualizations like charts, maybe a bar at the bottom that shows the total CPU time and is broken into the execution times for each process with different colors kind of something like this:

Three bugs I found that I would like to fix are:

1. The ghost column that appears when I expand the window.



Homework #2 - Jose Juan Sandoval

Mutual Exclusion Method Row Selection

Lock Variable

Process Statistics

Process ID	CR	State	Execution Time	Completion	
				Time	Percentage
0		Done	1835	7484	100%
1		Done	7154	29278	100%
2		Done	23050	55891	100%
3		Done	9211	40214	100%
4		Done	14641	50170	100%

Strict Alternation

Process Statistics

Process ID	CR	State	Execution Time	Completion	
				Time	Percentage
0		Done	1835	6770	100%
1		Done	7154	34619	100%
2		Done	23050	55891	100%
3		Done	9211	45222	100%
4		Done	14641	50796	100%

Simulation Configuration

5 20

2000

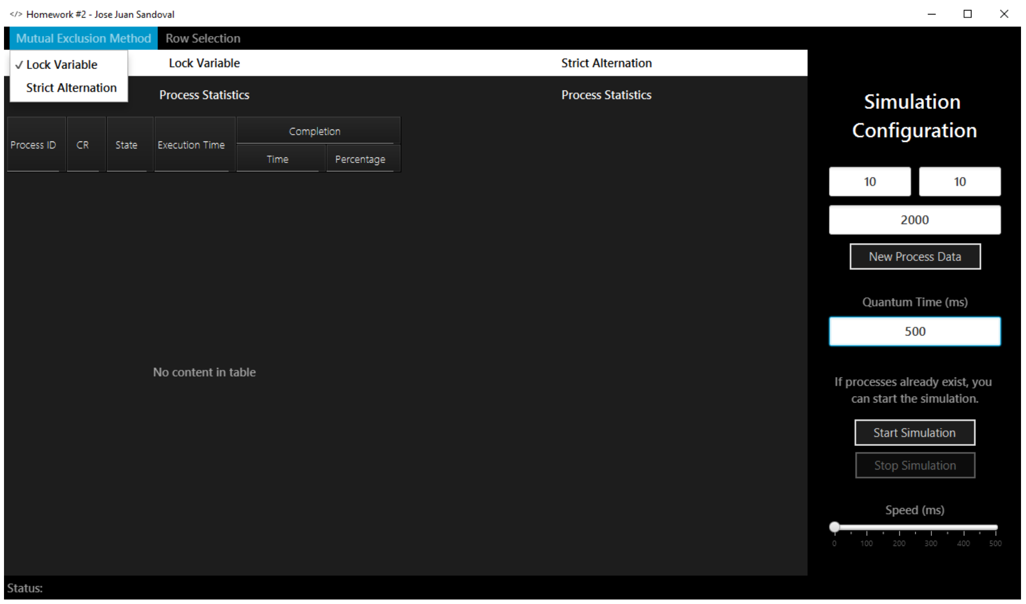
Quantum Time (ms) 500

If processes already exist, you can start the simulation.

Speed (ms)

STATUS: Simulation has finished.

2. Making the TableViews responsive so they fill out the whole space when one mutual exclusion method is disabled.



3. I saw the execution times wrapped around, they became negative numbers so I need to use a bigger data type for that.

7. Notes

I am turning in this project late because I underestimated how long it would take me to learn JavaFX to create good looking GUIs. Also I had to learn more about multithreading and creating thread safe algorithms. I also liked how I finally was able to implement the Observer pattern in something where it actually benefits the application. I spent a lot of time working on this and I am actually proud of it. This is my last year in school and I am aiming to create applications that are worth showing to future employers. Now if you have time Professor Gershman, I would like you to rip my application to shreds. Tell me what you think is wrong with it and suggest how I can improve it.

Like for example, JavaFX forces you implement the MVC pattern by using a Controller class. I wanted to turn it into a Singleton but it wouldn't let me, I can't remember what was the exception it was throwing. Anyway what I did instead was save a static reference on my Main class and access it from my ProcessSimulator class using the line Main.controller.someMethod() but I don't know if that is a good way to do it or if it would had been better to pass the reference to the ProcessSimulator. I did not want to pass it around because maybe I had to pass it to all threads I was instantiating that needed to access it, so keeping it as a static object in my Main class seemed a better option.

Also to end threads I found that you should not kill them because that could make your JVM unstable, that instead you should gracefully let them know that they should end. So I created a static boolean in my ProcessSimulator class that is checked to see if it should end or not. So sometimes you click the Stop Simulation button and the GUI stops updating after a little while because the threads do not end immediately. So what is a good way to end them immediately or is this the best way? I believe that’s it for now. I almost wrote you a 10-page report. By the way a .JAR file has been provided with the final program in case the project cannot be built because of the JavaFX stuff.