****

**Report**

**YASS CPU SIMULATOR**

**Name : Pritam Datta**

**ID No. : 2014-2-60-078**

**Course Code: CSE325**

**Course Title: Operating System**

**Semester: Spring 2017**

**Submitted To: Md. Mezbahul Islam**

**Department of CSE, East West University.**

**Submission Date: 11/04/2017**

**Investigating Synchronization**

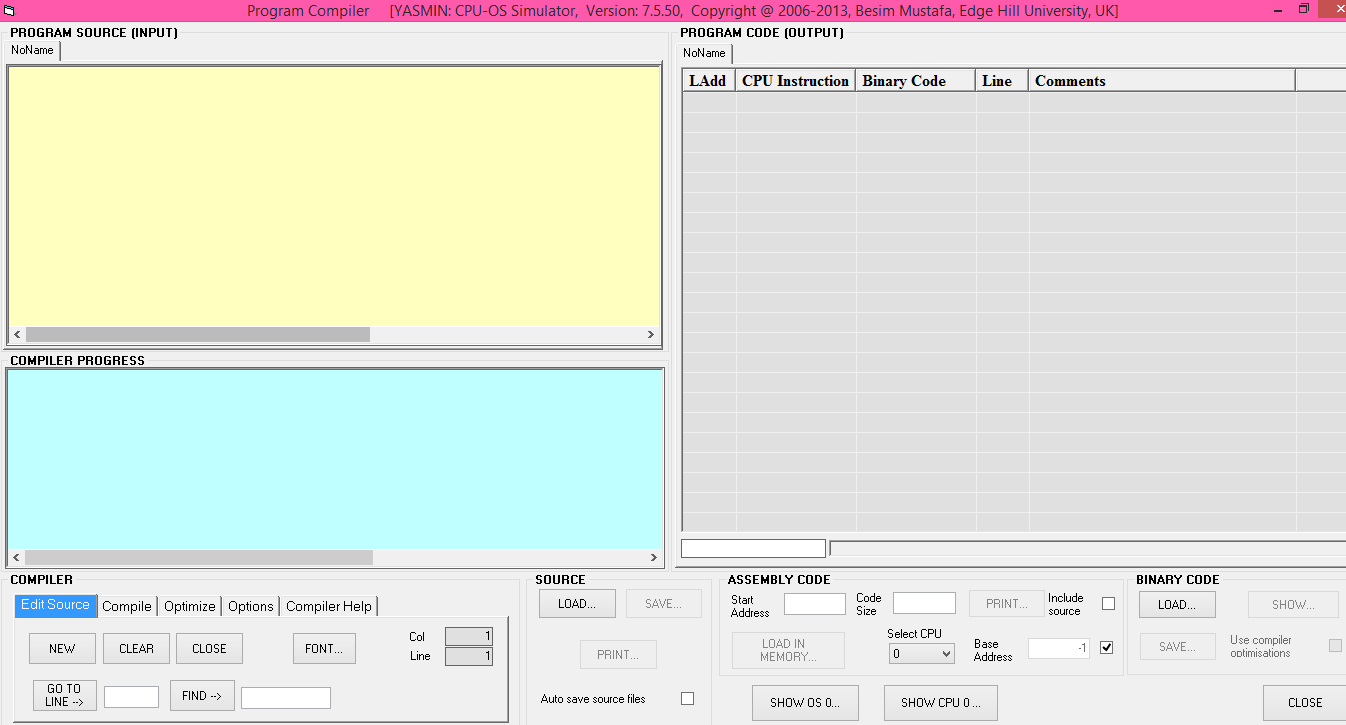
**Implementing Critical Region**

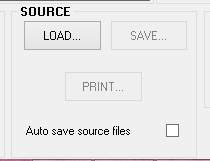
Objectives:

1. Understand shared global memory protection using synchronized threads.
2. Explain how critical regions of code can protect shared global memory areas.
3. Show that memory areas local to threads are unaffected by other threads.
4. Critical regions are often implemented using **semaphores** and **mutexes**. Be able to differentiate these how they differ.
5. Creating critical region and implementing this.

**Investigate and Explore:**

Now we need to create some executable code so that it can be run by the CPU under the control of the OS. In order to create this code, we need to use the compiler which is part of the system simulator. To do this, open the compiler window by selecting the **COMPILER** button in the current window. After opening the compiler button the interface of compiler is shown. Here is the sneak peak :



In the program source section, we can write the code / we can simply load the code from the source option if the code is available to the user from the beginning. 

**Compiling an executable code and Compiling and loading code:**

program CriticalRegion1

var g integer

sub thread1 as thread

writeln("In thread1")

g = 0

for n = 1 to 20

g = g + 1

next

writeln("thread1 g = ", g)

writeln("Exiting thread1")

end sub

sub thread2 as thread

writeln("In thread2")

g = 0

for n = 1 to 12

g = g + 1

next

writeln("thread2 g = ", g)

writeln("Exiting thread2")

end sub

writeln("In main")

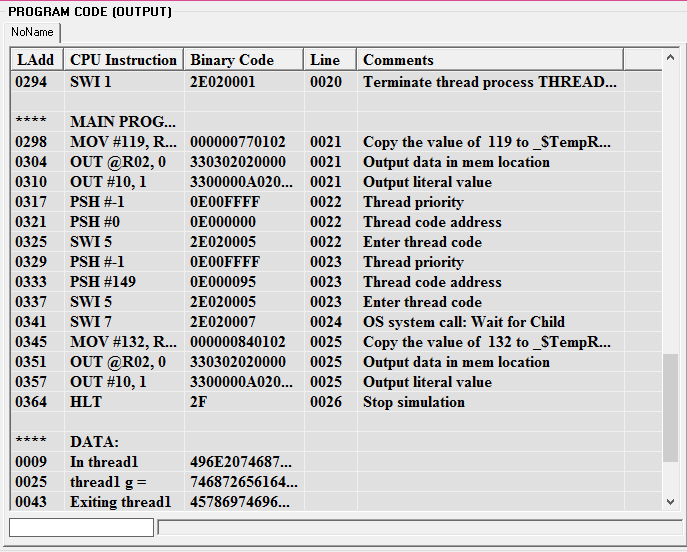
call thread1

call thread2

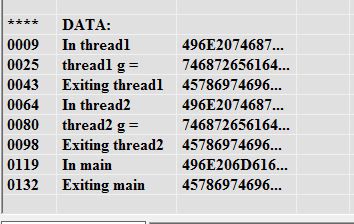
wait

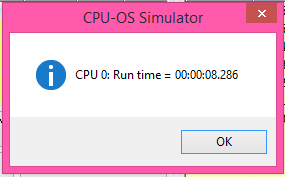
writeln("Exiting main")

end

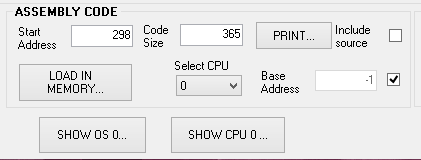
compile the above code using the compile button. From the name of this program code we can simply guess the matter. The above code creates a main program called *CriticalRegion1*. This program creates two threads thread1 and thread2. Each thread increments the value of the global variable **g** in two separate loops. After compilation the output of this code is :

When two thread finished two value of g is displayed on the console when two thread finishes.

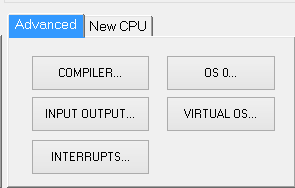


 And it took 08.286 times to compile.

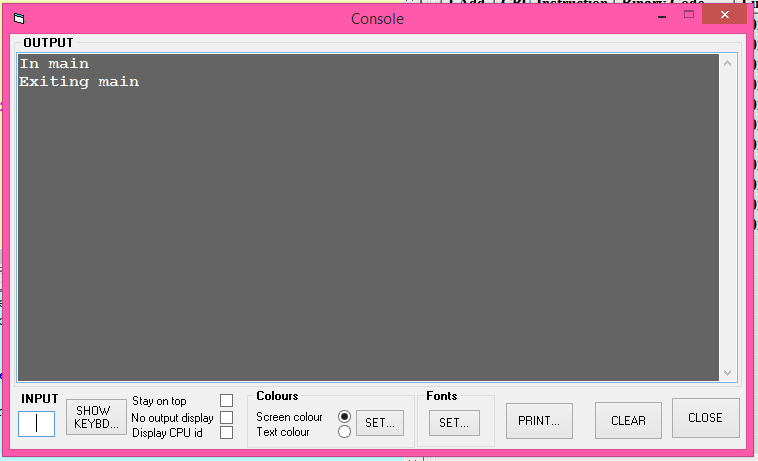
Now we need to load the CPU instruction in memory. We can do so by using the **LOAD IN MEMORY** button**.**

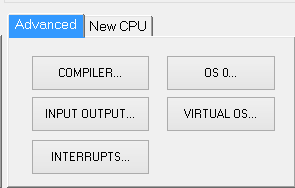


Now Display the console using the **INPUT/OUTPUT** button in CPU simulator. To do this we need head out to the home page. So first click on the INPUT/OUTPUT button on the Advance section which is below the compile button.



After doing this this input/output console is shown :

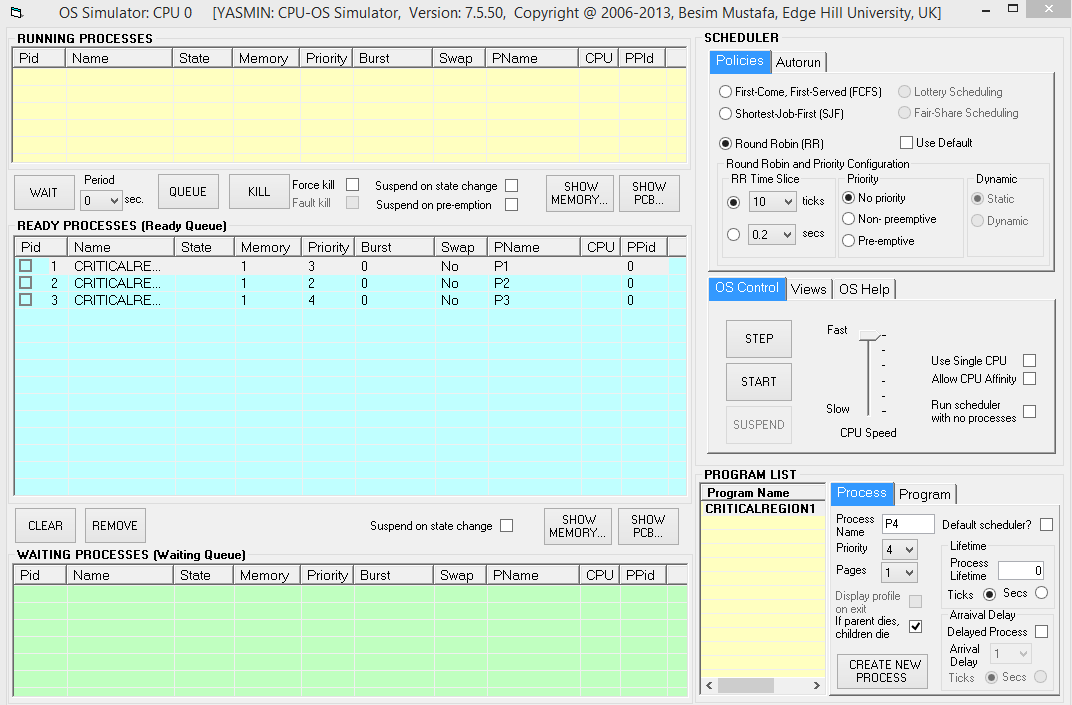


Now click on ‘**stay on the top’** check box. And Enter the OS simulator using the **OS 0…** button in CPU simulator.

After this You can see an entry, titled CriticalRegion1,in the **PROGRAM LIST** view.



Create an instance of this program using the **NEW PROCESS** button. Select **Round Robin** option in the **SCHEDULER/Policies** view. Select **10 ticks** from the drop‐down list in **RR Time Slice** frame. Make sure the console window is displaying . Move the **Speed** slider to the fastest position. Start the scheduler using the **START** button.

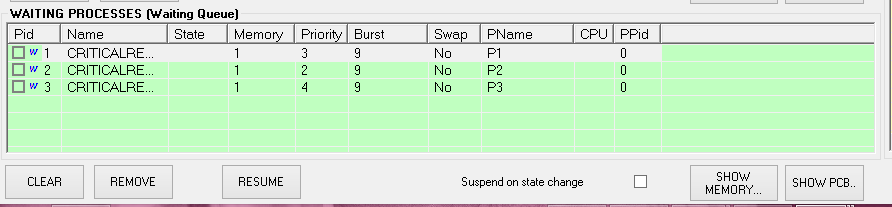


Here,

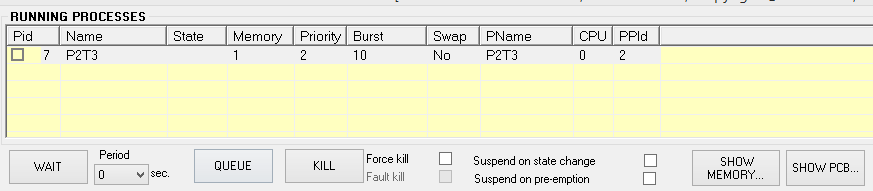
RR time slice =10

**Priority**

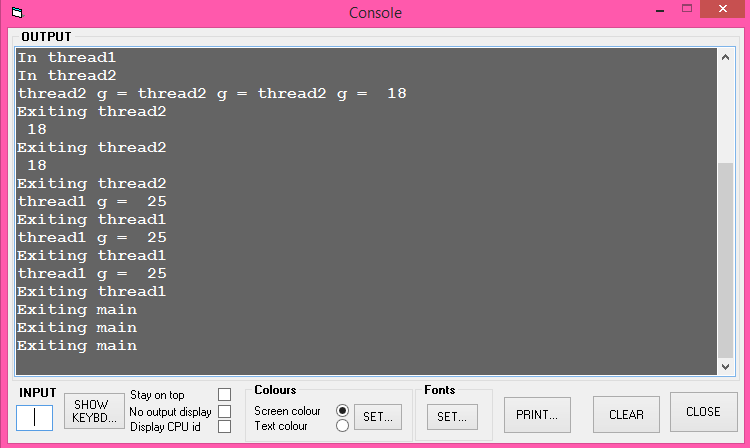
While running there are some process which have to wait while running so the waiting processes displays in the **waiting process** section.



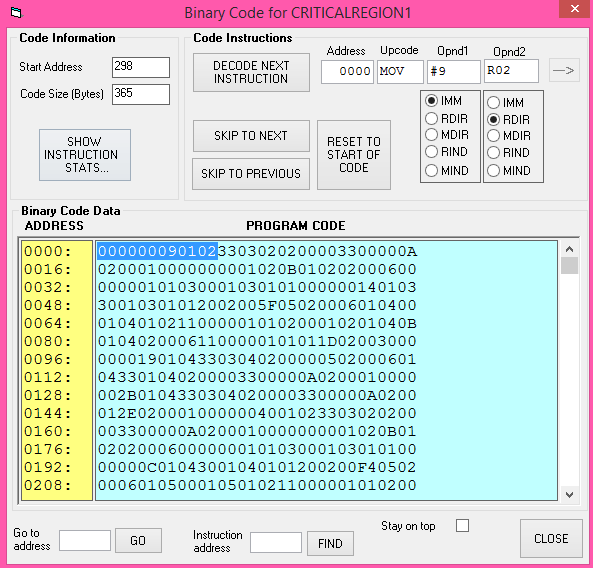
And running processes will show in the **Running process** section.



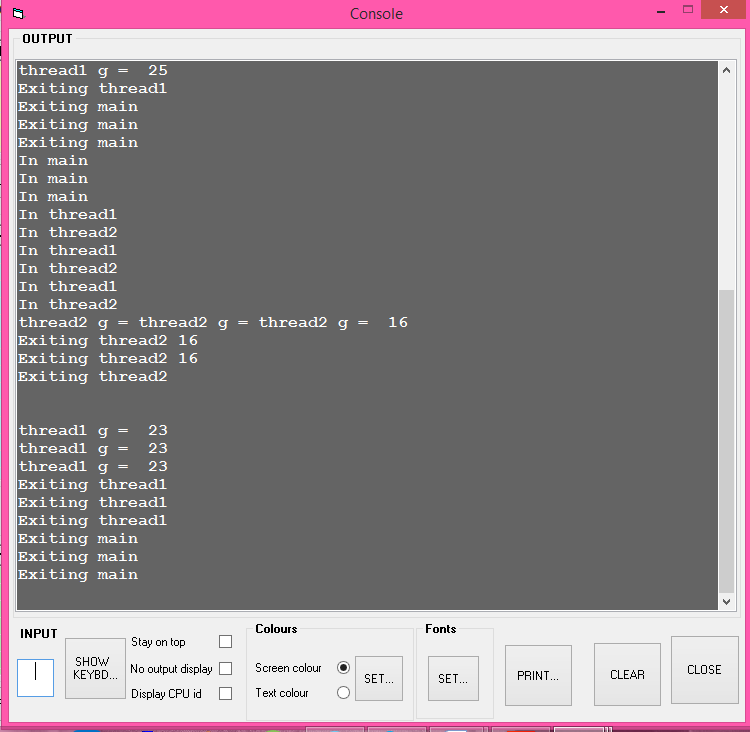
While doing this we have seen the input/output console is showing the thread in the output, and the value of thread1 and thread2 has been changed to another value.



Here is the binary code for critical region1,



Now , we have changed the **RR Time Slice** in the OS simulator window to **5 ticks** and repeated the above run. Now the result is,



Now we can the see that for 5ticks the two value of variable g of thread1 and thread2 are different from the value of 10 ticks.

Now do the same procedure for another code named Critical Region2:

program CriticalRegion2

var g integer

sub thread1 as thread **synchronise**

writeln("In thread1")

g = 0

for n = 1 to 20

g = g + 1

next

writeln("thread1 g = ", g)

writeln("Exiting thread1")

end sub

sub thread2 as thread **synchronise**

writeln("In thread2")

g = 0

for n = 1 to 12

g = g + 1

next

writeln("thread2 g = ", g)

writeln("Exiting thread2")

end sub

writeln("In main")

call thread1

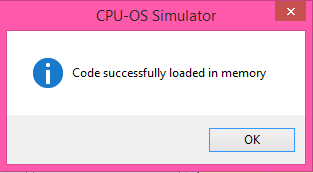
call thread2

wait

writeln("Exiting main")

end

Here, The **synchronise** keyword makes sure that the thread1 and thread2 code are executed mutually exclusively (i.e. not at the same time). After compiling the above program we have to load it in memory as before.



**Input/output console**

Here we can see that the value has not changed, it executed successfully without any change.

After that, we have modified this program for the second time. The new additions are in bold and underlined. Remove the two **synchronise** keywords. Renamed it as CriticalRegion3.

program CriticalRegion3

var g integer

sub thread1 as thread

writeln("In thread1")

**enter**

g = 0

for n = 1 to 20

g = g + 1

next

writeln("thread1 g = ", g)

**leave**

writeln("Exiting thread1")

end sub

sub thread2 as thread

writeln("In thread2")

**enter**

g = 0

for n = 1 to 12

g = g + 1

next

writeln("thread2 g = ", g)

**leave**

writeln("Exiting thread2")

end sub

writeln("In main")

call thread1

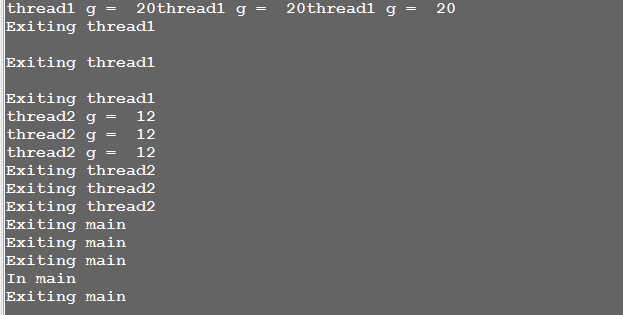
call thread2

wait

writeln("Exiting main")

end

NOTE: The **enter** and **leave** keyword pair protect the program code between them. This makes sure the protected code executes exclusively without sharing the CPU with any other thread. Now compile,load it in meomory and click os and create process then click run.



from this we can see that the value of variable g has changed so are the thread1 and thread2.

Now compile another program to see the difference. Modify this program for the third time. And generate the new additions in bold and underlined for our better understanding. Now remove the global variable **g, enter** and **leave** keywords. And we renamed it *CriticalRegion4*.

program CriticalRegion4

sub thread1 as thread

**var g integer**

writeln("In thread1")

g = 0

for n = 1 to 20

g = g + 1

next

writeln("thread1 g = ", g)

writeln("Exiting thread1")

end sub

sub thread2 as thread

**var g integer**

writeln("In thread2")

g = 0

for n = 1 to 12

g = g + 1

next

writeln("thread2 g = ", g)

writeln("Exiting thread2")

end sub

writeln("In main")

call thread1

call thread2

wait

writeln("Exiting main")

end

and now compile and load it in memory and click os and create process and the click start button and then click run and see the input/output console. And from the input/output console we can see that the thread1 became the value of thread 2 of previous code and the thread1 have became the value of thread2 of previous code.

