OS Experiment - 8

Name: Kartik Jolapara **SAP ID:60004200107** **Batch: B1**

Aim- Using the CPU-OS simulator to analyze and synthesize the following:

a. Process Scheduling algorithms.
b. Thread creation and synchronization.
c. Deadlock prevention and avoidance.

Problem Statement:

1) Install CPU-OS simulator
2) Perform the following steps

a) Process Scheduling algorithms

Loading and Compiling Program

You need to create some executable code so that it can be run by the CPU simulator under the control of the OS simulator. In order to create this code, you need to use the compiler which is part of the system simulator. This compiler is able to compile simple high-level source statements similar to Visual Basic. To do this, open the compiler window by selecting the **COMPILER…** button in the current window. You should now be looking at the compiler window. In the compiler window, enter the following source code in the compiler's source editor window (under **PROGRAM SOURCE** frame title):

```
program LoopTest
      i = 0  for n = 0
      to 40  i = i + 1
      next  end
```

Now you need to compile this in order to generate the executable code. To do this, click on the **COMPILE…** button. You should see the code created on the right in **PROGRAM CODE** view. Make a habit of saving your source code.

Click on the button **SHOW…** in **BINARY CODE** view. You should now see the **Binary Code for LOOPTEST** window. Study the program code displayed in hexadecimal format.

Now, this code needs to be loaded in memory so that the CPU can execute it. To do this, first we need to specify a base address (in **ASSEMBLY CODE** view): uncheck the box next to the edit box with label **Base Address**, and then enter 100 in the edit box. Now, click on the **LOAD IN MEMORY…** button in the current window. You should now see the code loaded in memory ready to be executed. You are also back in the CPU simulator at this stage. This action is

equivalent to loading the program code normally stored on a disc drive into RAM on the real computer systems.

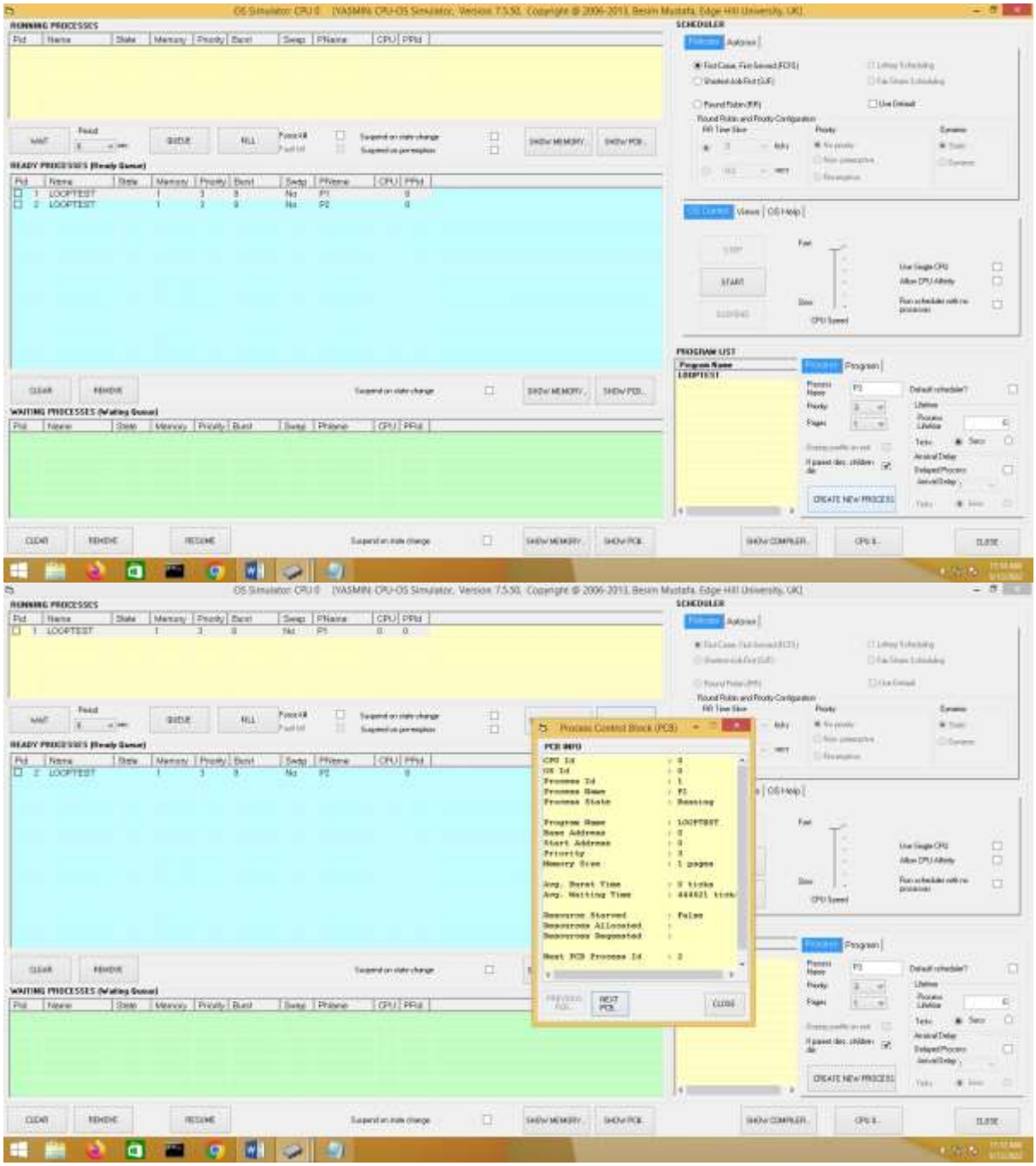
Creating processes from programs in the OS simulator.

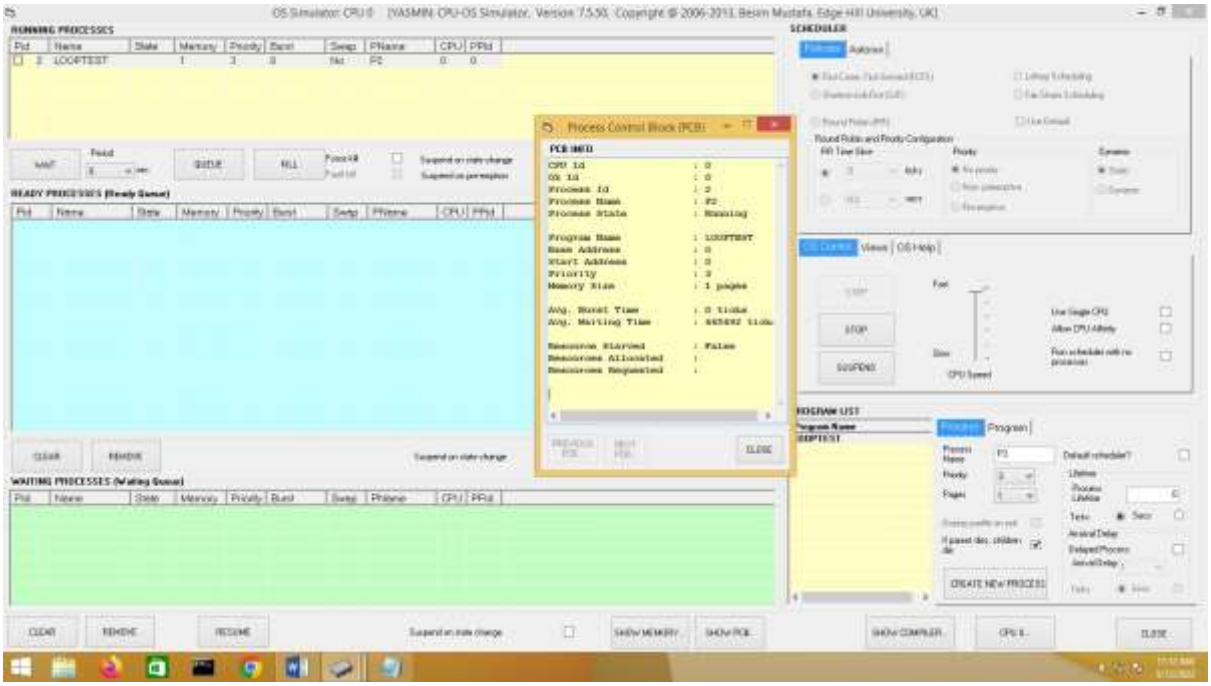
We are now going to use the OS simulator to run this code. To enter the OS simulator, click on the **OS 0…** button in the current window. The OS window opens. You should see an entry, titled **LoopTest**, in the **PROGRAM LIST** view. Now that this program is available to the OS simulator, we can create as many instances, i.e. processes, of it as we like. You do this by clicking on the **CREATE NEW PROCESS** button. Repeat this four times. Observe the four instances of the program being queued in the ready queue which is represented by the **READY PROCESSES** view.

NOTE: it is very important that you follow the instructions below without any deviation. If you do, then you must re-do the exercise from the beginning as any follow-up action(s) may give the wrong results.

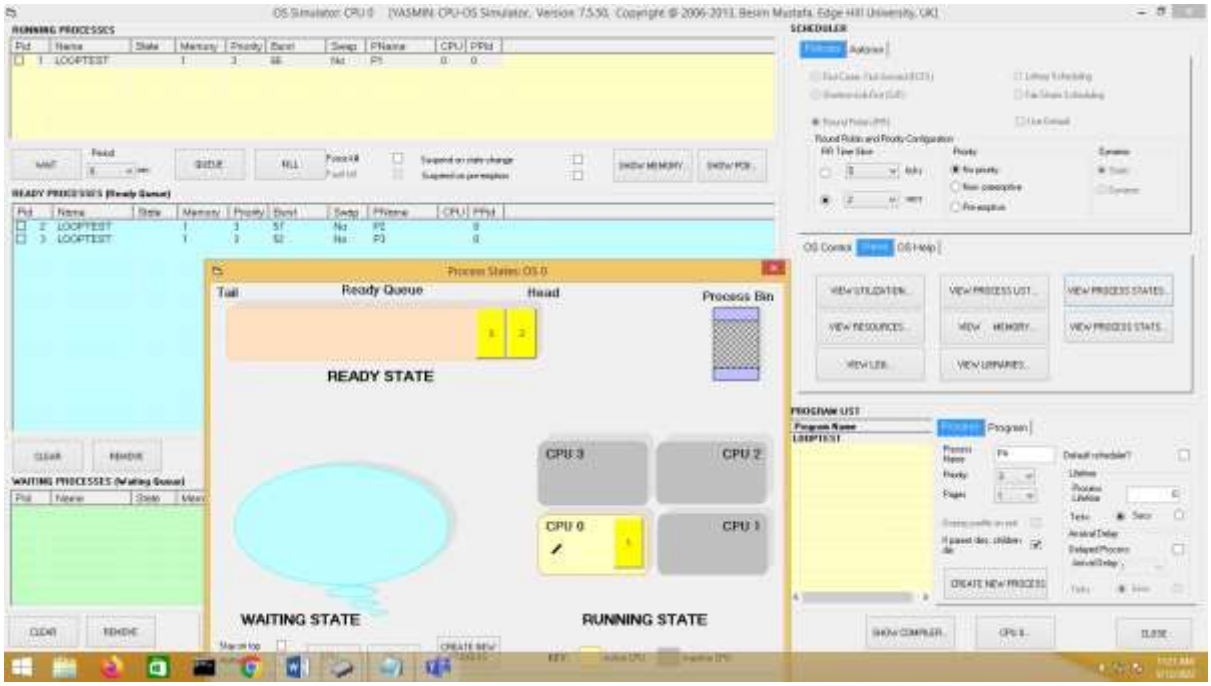
Selecting different scheduling policies and run the processes in the OS simulator

Make sure the **First-Come-First-Served (FCFS)** option is selected in the **SCHEDULER/Policies** view. At this point the OS is inactive. To activate, first move the **Speed** slider to the fastest position, then click on the **START** button. This should start the OS simulator running the processes. Observe the instructions executing in the CPU simulator window. Make a note of what you observe in the box below as the processes are run (you need to concentrate on the two views: **RUNNING PROCESSES** and the **READY PROCESSES** during this period).

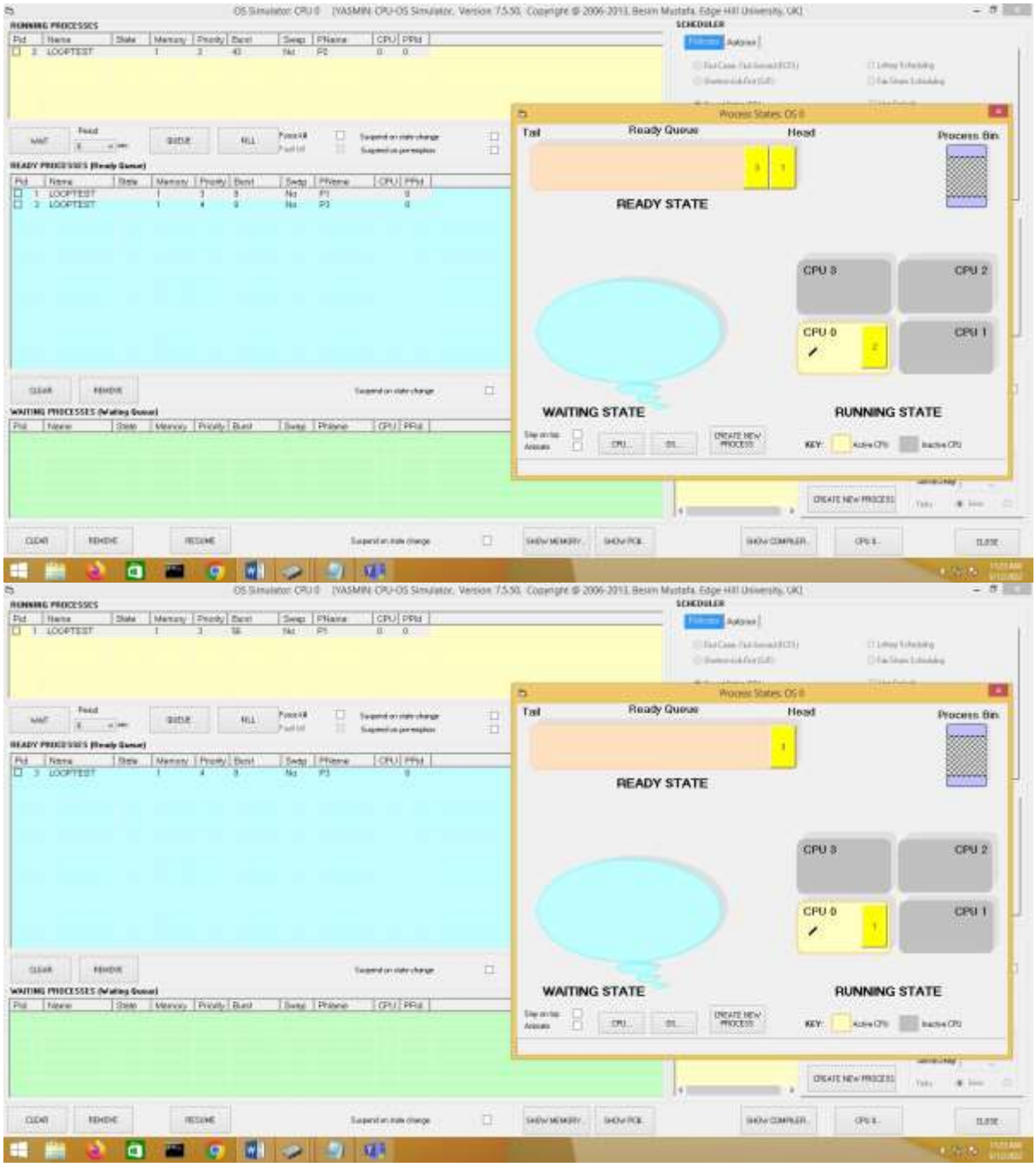

OS Simulator: CPU 0 [YASMIN: CPU-OS Simulator, Version 7.5.50, Copyright @ 2006-2013, Besim Mustafa, Edge Hill University, UK]
RUNNING PROCESSES
Pid
Name
State
Memory
Priority
Burst
Swap
PName
CPU
PPid
READY PROCESSES (Ready Queue)
1 LOOPTEST
2 LOOPTEST
WAITING PROCESSES (Waiting Queue)
SCHEDULER
Round Robin and Priority Configuration
OS Control
Views
OS Help
CPU Speed
PROGRAM LIST
Program Name
LOOPTEST
CREATE NEW PROCESS
CLEAR
REMOVE
RESUME
SHOW MEMORY
SHOW PCB
SHOW COMPILER
CLOSE
OS Simulator: CPU 0 [YASMIN: CPU-OS Simulator, Version 7.5.50, Copyright @ 2006-2013, Besim Mustafa, Edge Hill University, UK]
Process Control Block (PCB)
PCB INFO
CPU Id : 0
OS Id : 0
Process Id : 1
Process Name : P1
Process State : Running
Program Name : LOOPTEST
Base Address : 0
Start Address : 0
Priority : 3
Memory Size : 1 pages
Avg. Burst Time : 0 ticks
Avg. Waiting Time : 444621 ticks
Resources Starved : False
Resources Allocated :
Resources Requested :
Next PCB Process Id : 2
PREVIOUS PCB
NEXT PCB
CLOSE



When all the processes finish, do the following. Select **Round Robin (RR)** option in the **SCHEDULER/Policies** view. Then select the **No priority** option in the **SCHEDULER/Policies/Priority** frame. Create three processes. Click on the **START** button and observe the behaviors of the processes until they all complete. You may wish to use speed slider to slow down the processes to better see what is happening. Make a note of what you observed in the box below and compare this with the observation in step 1 above.

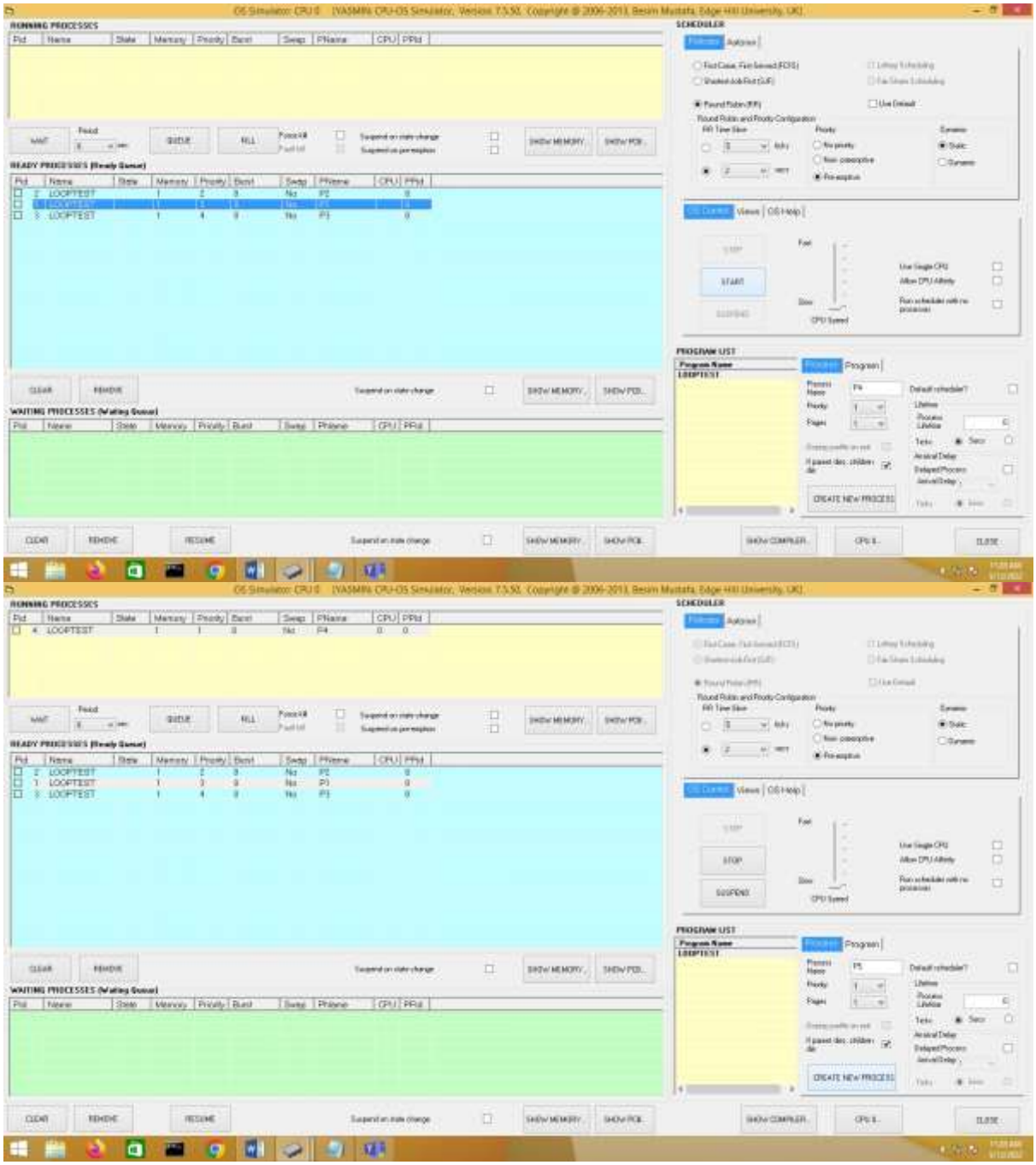


Then select the **Non-preemptive** priority option in the **SCHEDULER/Policies/Priority** frame. Create three processes with the following priorities: 3, 2 and 4. Use the **Priority** drop down list to select priorities. Observe the order in which the three processes are queued in the ready queue represented by the **READY PROCESSES** view and make a note of this in the box below (note that the lower the number the higher the priority is).



Slide the **Speed** selector to the slowest position and then hit the **START** button. While the first process is being run do the following. Create a fourth process with priority 1. Make a note of what you observe (pay attention to the **READY PROCESSES** view) in the box below.

Now kill all four processes one by one as they start running. Next, select the **Pre-emptive** option in the **SCHEDULER/Policies/Priority** frame. Create the same three processes as in step 3 and then hit the **START** button. While the first process is being run do the following. Create a fourth process with priority 1. Make a note of what you observe (pay attention to the **RUNNING PROCESSES** view). How is this behavior different than that in step 4 above?



Thread creation and synchronization.

Loading and Compiling a Program

In the compiler window enter the following source code:

```
program ThreadTest1  sub
     thread1 as thread
     writeln("In thread1")
     while true  wend
     end sub  sub thread2
     as thread  call
     thread1 writeln("In
```

```
        thread2")  while true
        wend
        end sub
        call thread2
        writeln("In main")  do
        loop
end
```

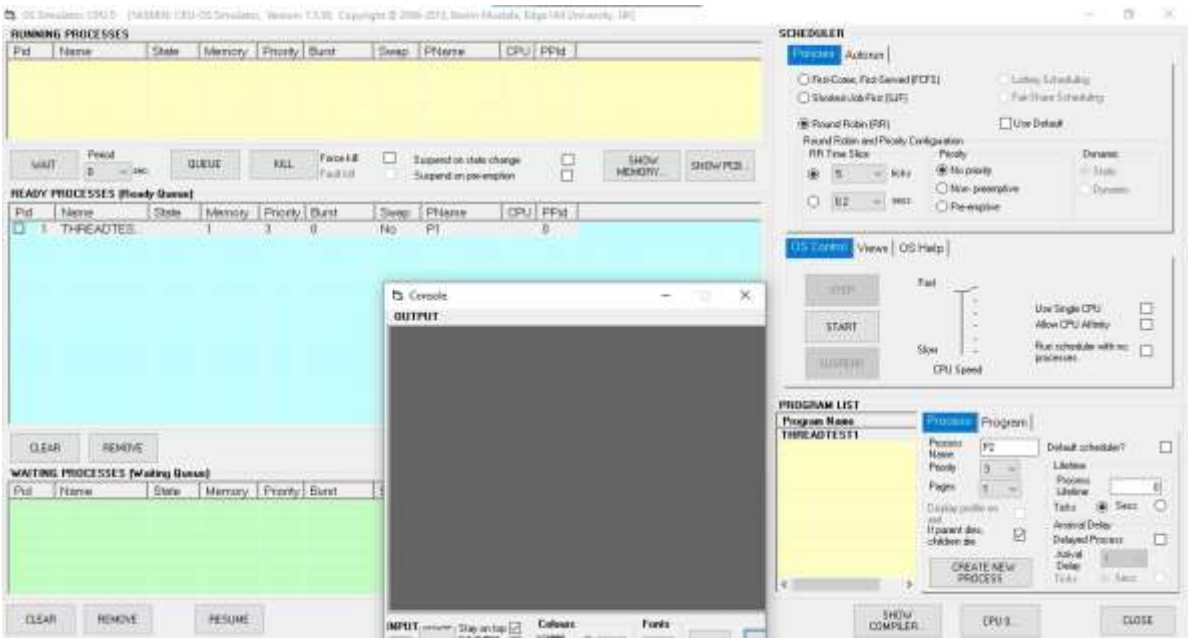
Compile the above source and load the generated code in memory.

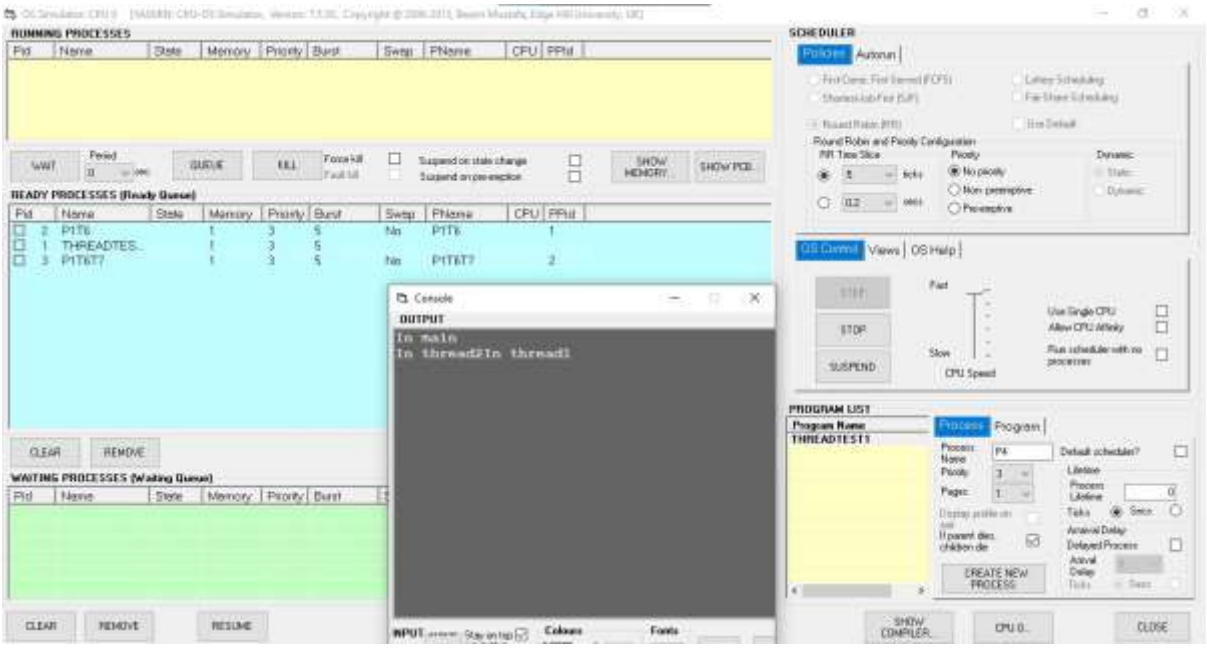
Make the console window visible by clicking on the **INPUT/OUTPUT…** button. Also make sure the console window stays on top by checking the **Stay on top** check box.

Now, go to the OS simulator window (use the **OS…** button in the CPU simulator window) and create a single process of program *ThreadTest1* in the program list view. For this use the **CREATE NEW PROCESS** button.

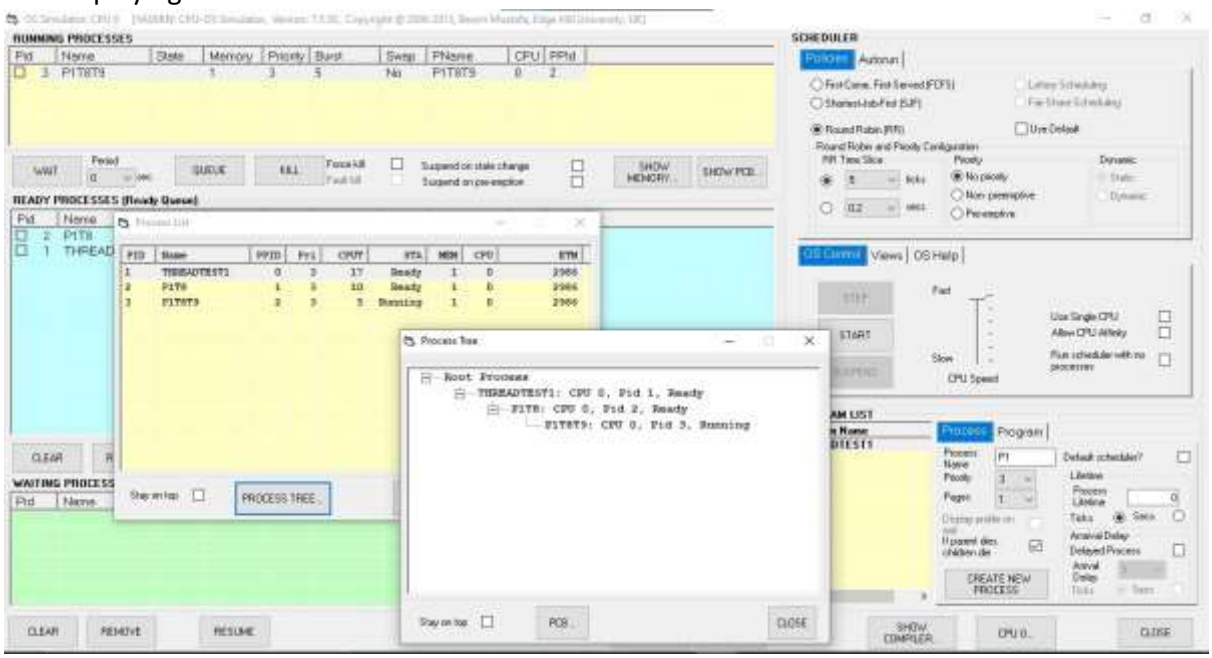
Make sure the scheduling policy selected is **Round Robin** and that the simulation speed is set at maximum.

Hit the **START** button and at the same time observe the displays on the console window. Briefly explain your observations and the no. of processes created in the box below.





Now, click on the **Views** tab and click on the **VIEW PROCESS LIST…** button. Observe the contents of the window now displaying.
In the Process List window hit the **PROCESS TREE…** button. Observe the contents of the window now displaying.



Stop the running processes by repeatedly using the **KILL** button in the OS simulator window.

Synchronization

Loading and Compiling a Program

In the compiler window, enter the following source code in the compiler source editor area (under **PROGRAM SOURCE** frame title). Make sure your program is exactly the same as the one below (best to use copy and paste for this).

```
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
```

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.

- Compile the above code using the **COMPILE…** button.
- Load the CPU instructions in memory using the **LOAD IN MEMORY** button**.**
- Display the console using the **INPUT/OUTPUT…** button in CPU simulator.
- On the console window check the **Stay on top** check box.

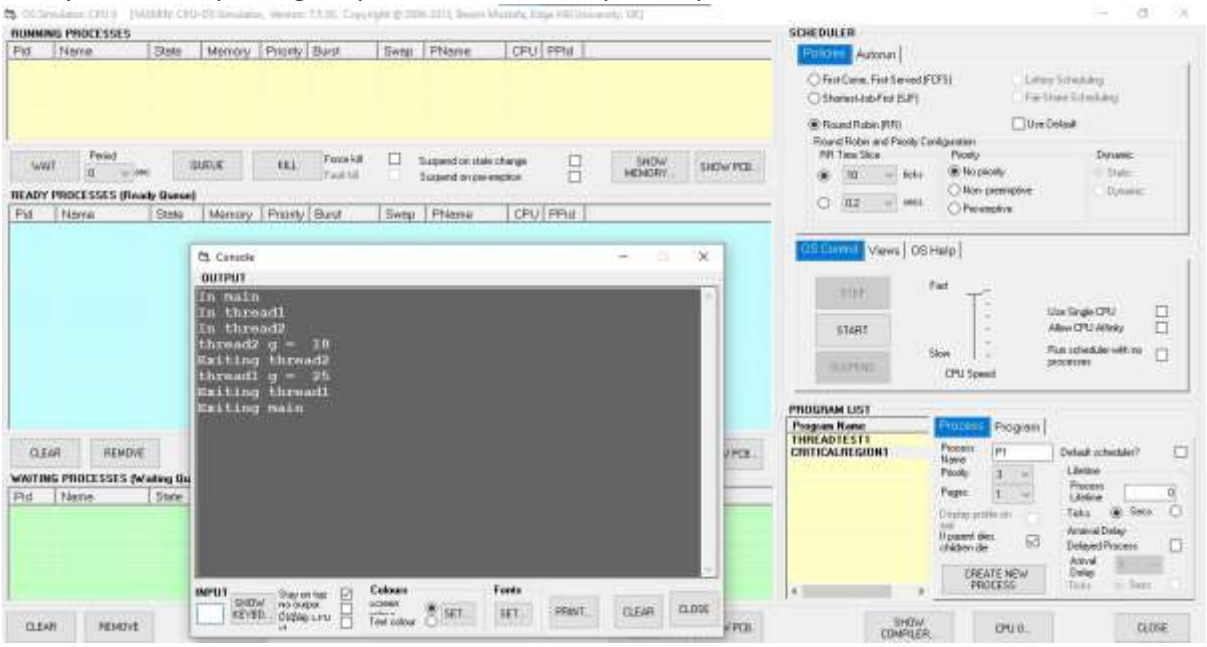
Running the above code

- Enter the OS simulator using the **OS 0…** button in CPU simulator.
- You should 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 (see above).
- Move the **Speed** slider to the fastest position.
- Start the scheduler using the **START** button.

Now, follow the instructions below without any deviations:

When the program stops running, make a note of the two displayed values of **g**. Are these values what you were expecting? Explain if there are any discrepancies.



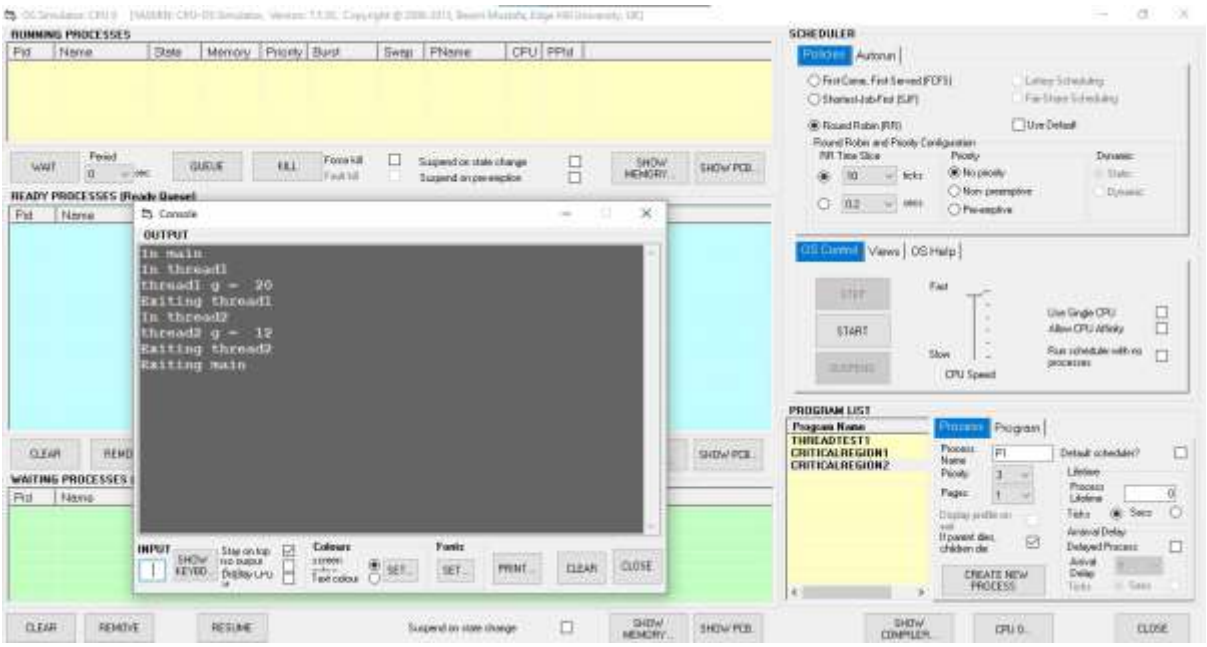
Modify this program as shown below. The changes are in bold and underlined. Rename the program *CriticalRegion2*.

```
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
```

Compile the above program and load in memory as before. Next, run it and carefully observe how the threads behave. Make a note of the two values of variable **g**.



Modify this program for the second time. The new additions are in bold and underlined. Remove the two **synchronise** keywords. Rename it *CriticalRegion3*.

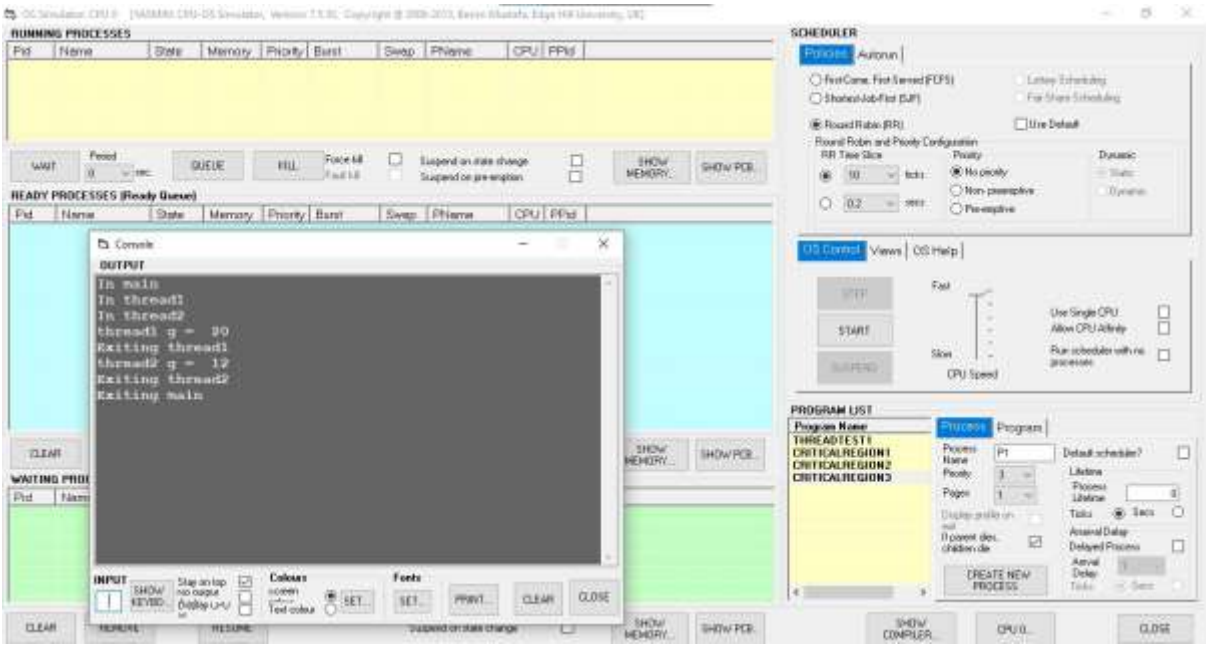
```
program CriticalRegion2  var g
     integer  sub thread1 as
     thread  writeln("In
     thread1")
          enter
          g = 0  for n =
          1 to 20  g = g
          + 1
          next
          writeln("thread1 g = ", g)
          leave
          writeln("Exiting thread1")
```

```
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.

Locate the CPU assembly instructions generated for the **enter** and **leave** keywords in the compiler's **PROGRAM CODE** view. You can do this by clicking in the source editor on any of the above keywords. Corresponding CPU instruction will be highlighted.:
Compile the above program and load in memory as before. Next, run it. Make a note of the two values of variable **g**.



Deadlock prevention and avoidance

Four processes are running. They are called **P1** to **P4**.There are also four resources available (only one instance of each). They are named **R0** to **R3**. At some point of their existence each process allocates a different resource for use and holds it for itself forever. Later each of the processes request another one of the four resources.

Use the Scenario P1 holding R0 and waiting for R1. P2 Holding R1 and waiting for R2. P3 holding R2 and waiting for R3. P4 holding R3 and waiting for R0.
Draw the resource allocation graph for a four process deadlock condition.

In the compiler window, enter the following source code in the compiler source editor area (under **PROGRAM SOURCE** frame title).

```
program DeadlockP N
resource(X, allocate)
wait(3)  resource(Y,
allocate)  for n = 1 to
20  next  end
```

2)

a. Copy the above code and paste it in three more edit windows so that you have a total of four pieces of source code. (Click on New under compiler to create new edit windows)
b. In each case change **N** in the program name to 1 to 4, e.g. DeadlockP**1,** DeadlockP**2,** etc.
c. Look at your graph you constructed in (1) above and using that information fill in the values for each of the **X**s and **Y**s in the four pieces of source code. (X is resource the process is holding and Y is the resource process is waiting for. Eg. For P1, X=0 and Y=1)
d. Compile each one of the four source code.
e. Load in memory the four pieces of code generated.
f. Now switch to the OS simulator.
g. Create a single instance of each of the programs. You can do this by double-clicking on each of the program names in the **PROGRAM LIST** frame under the **Program Name** column.
h. In the **SCHEDULER** frame select **Round Robin (RR)** scheduling policy in the **Policies** tab.
i. In OS Control tab, push the speed slider up to the fastest speed.
j. Select the **Views** tab and click on the **VIEW RESOURCES...** button.

k. Select **Stay on top** check box in the displayed window.
l. Back in the **OS Control** tab use the **START** button to start the OS scheduler and observe the changing process states for few seconds.
m. Have you got a deadlock condition same as you constructed in (1) above? If you haven't then check and if necessary re-do above. Do not proceed to (n) or (3) below until you get a deadlock condition.
n. If you have a deadlock condition then click on the **SHOW DEADLOCKED PROCESSES…** button in the **System Resources** window. Does the highlighted resource allocation graph look like yours?

Now that you created a deadlock condition let us try two methods of getting out of this condition:

a. In the **System Resources** window, there should be four resource shapes that are in red colour indicating they are both allocated to one process and requested by another.
b. Select one of these resources and click on the **Release** button next to it.
c. Observe what is happening to the processes in the OS Simulator window.
d. Is the deadlock situation resolved? Explain briefly why this helped resolve the deadlock.

e. Re-create the same deadlock condition (steps in 2 above should help).

f. Once the deadlock condition is obtained again do the following: In the OS Simulator window, select a process in the waiting queue in the **WAITING PROCESSES** frame.

g. Click on the REMOVE button and observe the processes.

h. Has this managed to resolve the deadlock? Explain briefly why this helped resolve the deadlock.

This part of the exercises was about two methods of **recovering** from a deadlock condition **after** it happens.

We now look at two methods of **preventing** a deadlock condition **before** it happens.

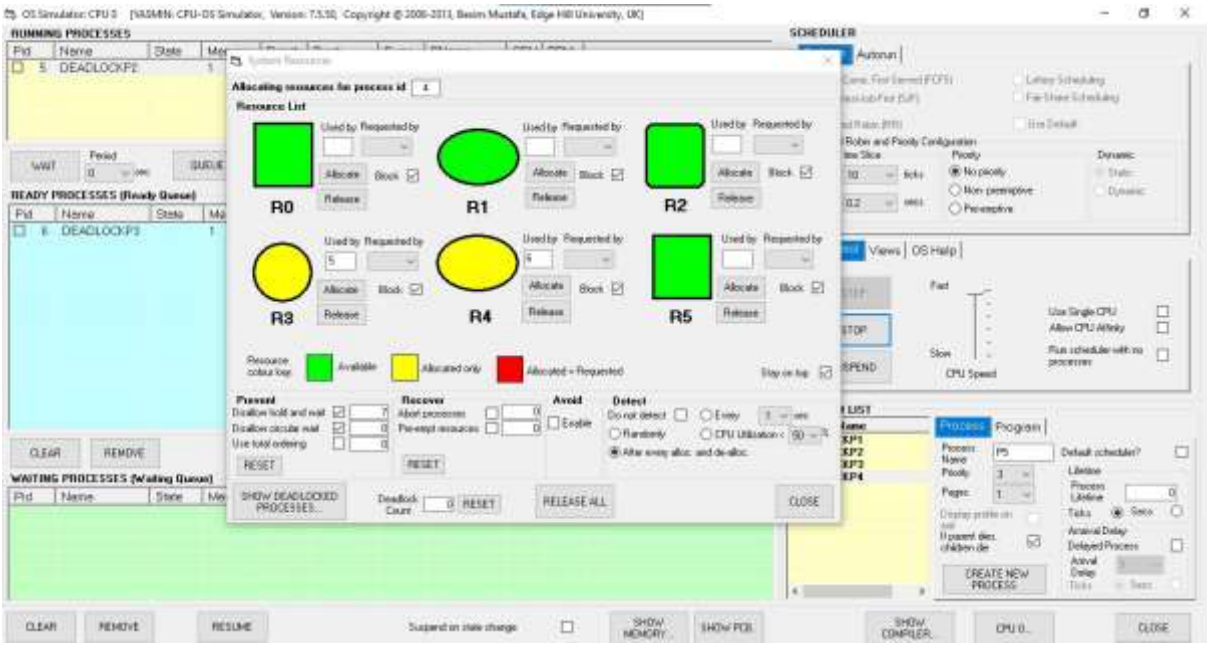
a. In the **System Resources** window select the **Disallow hold and wait** check box in the **Prevent** frame.

b. Try to re-create the same deadlock condition as before. Have you been successful? What happened? Click on the **SHOW DEADLOCKED PROCESSES...** button and observe the displayed information in the text window for potential clues.

c. Next, uncheck the **Disallow hold and wait** check box and check the **Disallow circular wait** check box.

d. Try to re-create the same deadlock condition as before. Have you been successful? What happened? Click on the **SHOW DEADLOCKED PROCESSES...** button and observe the displayed information in the text window for potential clues.

We are now going to try a third method of preventing deadlocking before it happens. It is called "total ordering" method. Here the resources are allocated in increasing resource id numbers only. So, for example, resource R3 must be allocated after resources R0 to R2 and resource R1 cannot be allocated after resource R2 is allocated. Looking at your resource allocation graph can you see how this ordering can prevent a deadlock? Comment.



a. In the **System Resources** window select the **Use total ordering** check box in the **Prevent** frame. The other options should be unchecked.

b. Try to re-create the same deadlock condition as before. Have you been successful? What happened? Click on the **SHOW DEADLOCKED PROCESSES...** button and observe the displayed information in the text window for potential clues. What happened? Comment.


RUNNING PROCESSES
READY PROCESSES (Ready Queue)
WAITING PROCESSES (Waiting Queue)
SCHEDULER
R0
R1
R2
R3
R4
R5
SHOW DEADLOCKED PROCESSES...
RELEASE ALL
CLOSE