Investigating Synchronisation

Introduction

At the end of this lab you should be able to:

1. Show that writing to unprotected shared global memory region can have undesirable side effects when accessed by threads at the same time.
2. Understand shared global memory protection using synchronised threads.
3. Explain how critical regions of code can protect shared global memory areas.
4. Show that memory areas local to threads are unaffected by other threads.

Processor and OS Simulators

The computer architecture tutorials are supported by simulators, which are created to underpin theoretical concepts normally covered during the lectures. The simulators provide visual and animated representation of mechanisms involved and enable the students to observe the hidden inner workings of systems, which would be difficult or impossible to do otherwise. The added advantage of using simulators is that they allow the students to experiment and explore different technological aspects of systems without having to install and configure the real systems.

Basic Theory

Concurrent processes accessing global shared resources at the same time can produce unpredictable side-effects if the resources are unprotected. Computer hardware and operating system can provide support for implementing critical regions of code when globally accessible resources are shared by several concurrently executing threads.

Lab Exercises - Investigate and Explore

Start the CPU simulator. You now 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, you 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.

1. 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.

Work out what two values of **g** you would expect to be displayed on the console when the two threads finish?

thread1 g= 20 thread2 g = 12

Compiling and loading the above code:

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

Running the above code:

i) Enter the OS simulator using the **OS 0…** button in CPU simulator.
ii) You should see an entry, titled *CriticalRegion1*, in the **PROGRAM LIST** view.
iii) Create an instance of this program using the **NEW PROCESS** button.
iv) Select **Round Robin** option in the **SCHEDULER/Policies** view.
v) Select **10 ticks** from the drop-down list in **RR Time Slice** frame.
vi) Make sure the console window is displaying (see above).
vii) Move the **Speed** slider to the fastest position.
viii) Start the scheduler using the **START** button.

Now, follow the instructions below without any deviations:

2. 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.

The values I expected were not the outputs that displayed. Thread2 g = 18 and thread1 g = 25. This happened because we gave the CPU control for 10 ticks and the value was being passed between both threads.

3. Change **RR Time Slice** in the OS simulator window to **5 ticks** and repeat the above run. Again, make note of the two values of the variable **g**. Are these different than the values in (2) above? If so, explain why.

With 5 ticks, we got Thread2 g = 16 and Thread1 g = 23. This was different than the values in 2 above because the CPU is switching between the two at less ticks than above. That means that g is being passed between both threads at less ticks it's going to do it faster.

4. 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
```

NOTE: The **synchronise** keyword makes sure the thread1 and thread2 code are executed mutually exclusively (i.e. not at the same time).

5. 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**. Are the results different than those in (2) and (3) above? If so, why?

The two values of variable g are different than the results from 2 and 3. We got thread1 g=20 and thread2 g=12. This happened because we added synchronise in which made the CPU start & finish one thread before going to the other one. Therefore we're not switching between threads. Synchronise made them mutually exclusive which means when one thing happens the other cant.

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

7. 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. Make a note of this instruction here:

Enter/Leave will make them exclusive and not share anything between two threads.

Instruction = SWI 18(enter) SWI 19(leave)

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

Thread1 g= 20 and thread2 g = 12

9. Modify this program for the third time. The new additions are in bold and underlined. Remove the global variable **g, enter** and **leave** keywords. Rename 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
```

10. Compile the above program and load in memory as before. Next, run it. Make a note of the two values of variable **g**. How do the new **g** variables differ than the ones in (1), (4) and (6) above?

We get, thread2 g= 12 and thread1 g= 20. The new g variables differ from the ones in 1 4 and 6 above because the variable is no longer a global variable, the threads no longer have to share the variable.

11. So what have we done so far? To help understand theory better try to answer the following questions. You need to include this in your portfolio, so it is important that you attempt all the questions below. However, you don't need to complete this part during the tutorial session.

    a) Briefly explain the main purpose of this tutorial as you understand it.
    b) Why have we chosen to display the same global variable **g** in both threads?
    c) What popular high-level language uses the keyword **synchronise** (or similar) for the same purpose as the code in (4)?
    d) Critical regions are often implemented using **semaphores** and **mutexes**. Find out what these are and how they differ. Describe on a separate sheet.
    e) Some computer architectures have a "test-and-set" CPU instruction for implementing critical regions. Find out how this works and briefly describe on a separate sheet.
    f) In the absence of any help from hardware and operating system, how would you protect a critical region in your code? Suggest a way of doing it and state how it would differ from the above methods (hint: "busy wait").

From my understanding the purpose of this tutorial is to show us to be mindful of how we use global memory as the results can vary from it's intended purpose. The reason for why we have chosen to display the same global variable g in both threads to represent memory and show just how global memory is affected if not used properly. A popular popular high-level language that uses the keyword synchronise is Java. While mutexes allows only one task at a time, semaphore is sort of the same, except another task can signal by another thread. The test-and-set CPU instruction allows you to check that the value in memory location is the value you are trying to overwrite. In the absence of any help from the hardware of operating system, you could protect critical region in your code by continuously checking if a condition is true.