Introduction to Operating Systems

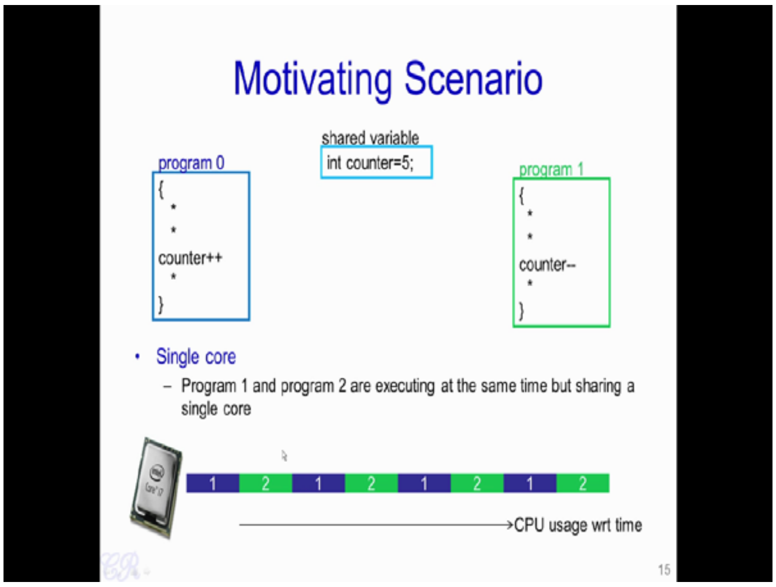
Prof. Chester Rebeiro
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Week – 06
Lecture – 24
Synchronization

Hello. In the last video we had looked at IPC’s or Inter Process Communication. So we had seen that due to IPC’s we could built application comprising of several processes and we could achieve modularity. As a result of having modular processes with each process doing a specific job we are able to have a very efficient and easy to understand applications.

One consequence of having this modular approach and the use of IPC’s is the requirement of Synchronization. In this video we will look at what synchronization is and what are the issues corresponding to synchronization, and how it can be solved.

(Refer Slide Time: 01:03)



Motivating Scenario

shared variable
```
int counter=5;
```

program 0
```
{
  *
  *
counter++
  *
}
```

program 1
```
{
  *
  *
counter--
  *
}
```

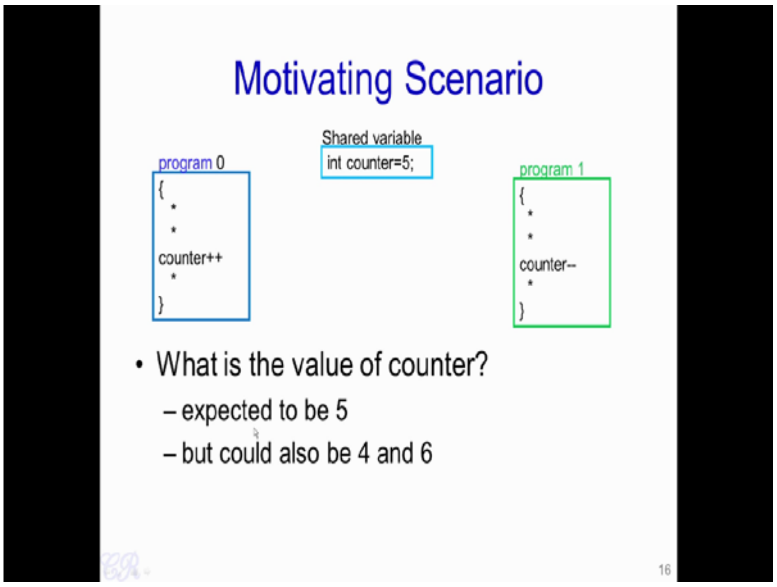
- Single core
  - Program 1 and program 2 are executing at the same time but sharing a single core

1
2
1
2
1
2
1
2
CPU usage wrt time
15

Let us explain what synchronization is with this motivating scenario. Let us say we have two programs; program 0 and program 1, and a shared variable which is defined as counter. It is a global shared variable int counter equal to 5. So, in program 0 we are incrementing the counter by 1, while in program 1 we are reducing the counter by 1, essentially decreasing the counter by 1. Now as we know that when we execute this program, let us say in a single code processor.

So, program 0 would execute for some time then there will be a context switch then program 2 will execute and then program 1 would execute and so on. Assuming that it is a round robin scheduling and assuming that no other process is present Now, the question is what would be the value of counter.

(Refer Slide Time: 02:03)



Motivating Scenario

Shared variable

```
int counter=5;
```

program 0

```
{
  *
  *
counter++
  *
}
```

program 1

```
{
  *
  *
counter--
  *
}
```

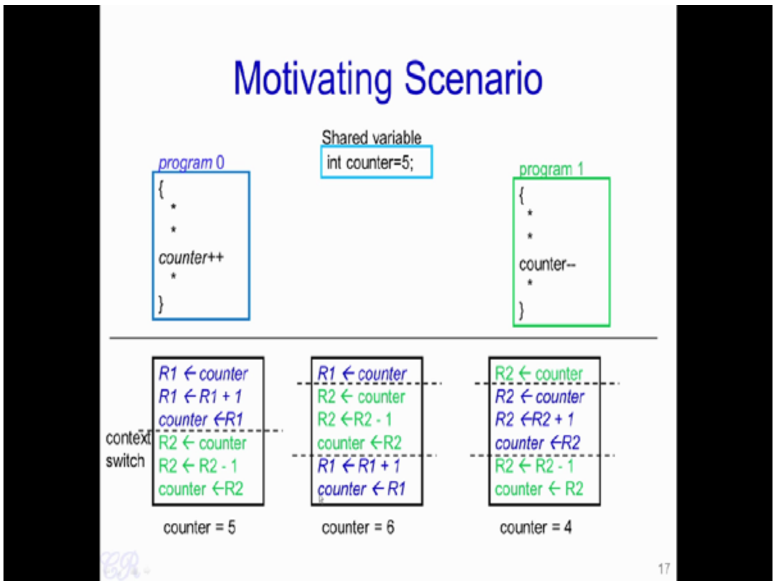
- What is the value of counter?
  - expected to be 5
  - but could also be 4 and 6

16

One would expect that the counter value is 5 because, let us say program 0 executes first so it is going to increment the value of counter, so counter becomes 6 over here. And then program 1 executes and it decrements the value of counter and since a counter is shared and has the value of 6 it gets decremented back to 5. On the other hand, let us say program 1 executes first and then program 0, so counter minus minus would cause the counter to reduce from 5 to 4, and then program 0 executes this particular line causing it to increment the counter from 4 back to 5.

Thus, one would expect that the result at the end of both these programs would be 5 for the value of the count. But now what we will show is that, we can also obtain the values of 4 and 6 for the value of counter. So let us see this more in detail.

(Refer Slide Time: 03:08)



Motivating Scenario

Shared variable

```
int counter=5;
```

program 0

```
{
  *
  *
counter++
  *
}
```

program 1

```
{
  *
  *
counter--
  *
}
```

context switch

```
R1 ← counter
R1 ← R1 + 1
counter ← R1
R2 ← counter
R2 ← R2 - 1
counter ← R2
```

counter = 5

```
R1 ← counter
R2 ← counter
R2 ← R2 - 1
counter ← R2
R1 ← R1 + 1
counter ← R1
```

counter = 6

```
R2 ← counter
R2 ← counter
R2 ← R2 + 1
counter ← R2
R2 ← R2 - 1
counter ← R2
```

counter = 4

17

So, let us look more deeply into how these instructions are executing. Let us see what happens when we are actually incrementing this counter. So, essentially what we are doing is we are seeing how what happens when we are counter plus plus with respect to the assembly instruction. First, the value of counter which is stored in memory is loaded into a register say R1 and then R1 is incremented by 1. So that is the second line R1 equal to R1 plus 1 and then R1 return back into memory. So, R1 is return back into the value of counter which is stored in memory.

In terms of the numbers we have 5 here which is the counter, so the contents of counter is 5 and that is loaded into R1 then 5 is incremented so now R1 contains 6 and the value R1 is return back into counter. So, this value of 6 is return back into the memory location specified by counter and therefore 6 is return back in the counter.

Now, suppose there is a context switch the same thing happens again. The value of counter is is loaded into R2, so the R2 has a value of 6 then we are decrementing R2 by 1, so R2 becomes 6 minus 1 that is 5. And then the register R2 is stored back into the memory location counter. Therefore, R2 is 5; therefore the value of 5 is stored back into the counter. At the end of these two programs executing the value of counter is 5.

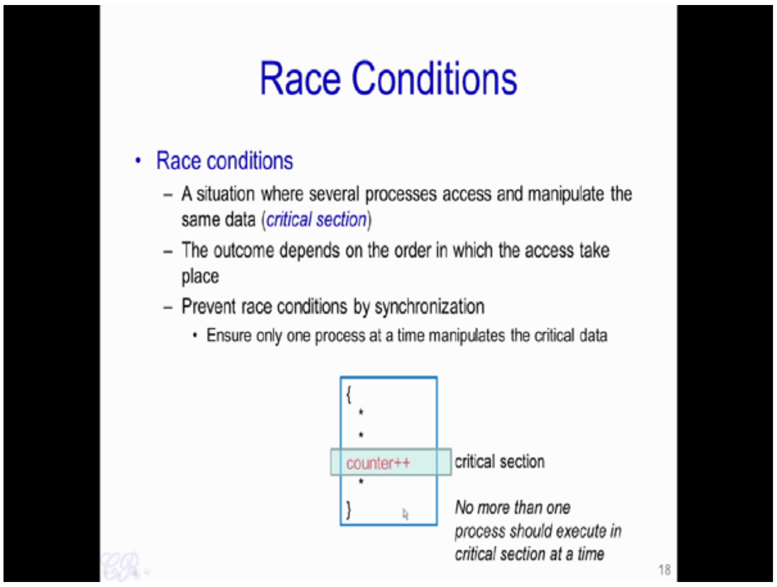
Now let us look at the second two scenarios, when we get the value of counter as 4 and we get the value of counter as 6. So, let us look at this scenario. Program 0 executes and lowers the value of counter into R1; therefore recollect that R1 has the value of 5. Now there is a context switch over here and process 1 executes. Now the counter over here is still having the value of 5 and it loaded into the register R2. Then R2 is decremented by 1, so R2 now has 4. And the value of 4 is stored back into counter. Now, counter in memory has the value of 4.

Now there is a context switch again and recollect. When there is a context switch program 0 continues from wherever it has stopped. Essentially, the context switch was stored in the kernel is restored back into process 0 allowing it continue from where it stopped. Therefore, we see that value of R1 at this particular point was 5 and after the context switch we have R1 back at 5 again over here, so R1 is incremented by 1 to get 6 and the value of 6 is stored into counter. Therefore, at the end of this execution we get the counter value equal to 6.

Now let us look at the third case that is when we get the counter value equal to 4. This is the exactly opposite to the second case in which the program 1 is executed first, so essentially the counter which has the value of 5 gets loaded into R2, therefore R2 has 5. Now there is a context switch causing program 0 to execute and the value of counters is loaded into R2 incremented by 1, so that is 6 and 6 is return back into the counter so, counter now will have the value of 6. There is a context switch again causing program 1 to execute from where it had stopped. So, we notice that R2 had the value of 5. Now, R2 is reduced by 1, so makes it 4. And the value of 4 is stored into the memory location corresponding to counter. Thus, at the end of the execution in this case the value of counter is 4

So, this was an example of the issues that would occur when we have a shared memory. Even though there was a very simple operation of incrementing a counter in one place while decrementing the counter in another place we had seen that the result could be different, depending on the how the instructions get executed and how the context switch is occurred. We would define this scenario more formally by what is known as Race Conditions.

(Refer Slide Time: 07:43)



Race Conditions

- Race conditions
  - A situation where several processes access and manipulate the same data (*critical section*)
  - The outcome depends on the order in which the access take place
  - Prevent race conditions by synchronization
    - Ensure only one process at a time manipulates the critical data

```
{
 .
 .
 counter++
 .
}
```

critical section

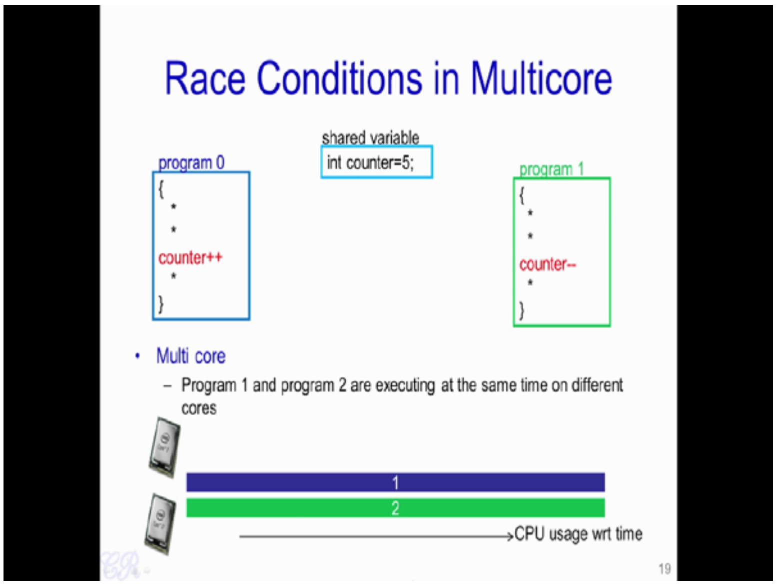
No more than one process should execute in critical section at a time

18

A race condition is a situation where several processes access and manipulate the same data. So this part of the process which accesses comment or the shared data is known as a Critical Section. The outcome of a race condition would depend on the order in which the accesses to that data take place. As we have seen in the previous example depending on which program executes first as well as depending on how the context switch is occur the result would vary. The race conditions could be prevented by what is known as synchronization. Essentially, with synchronization we would ensure that only one process at a time would manipulate the critical data.

So coming back to our example what is required, is that we would mark this area of instructions which access or which manipulate shared data as a critical section. Then we would have some additional techniques to ensure that no more than one process could execute in a critical section at a given time. We will see this in more detail in this particular videos that follow this.

(Refer Slide Time: 09:09)



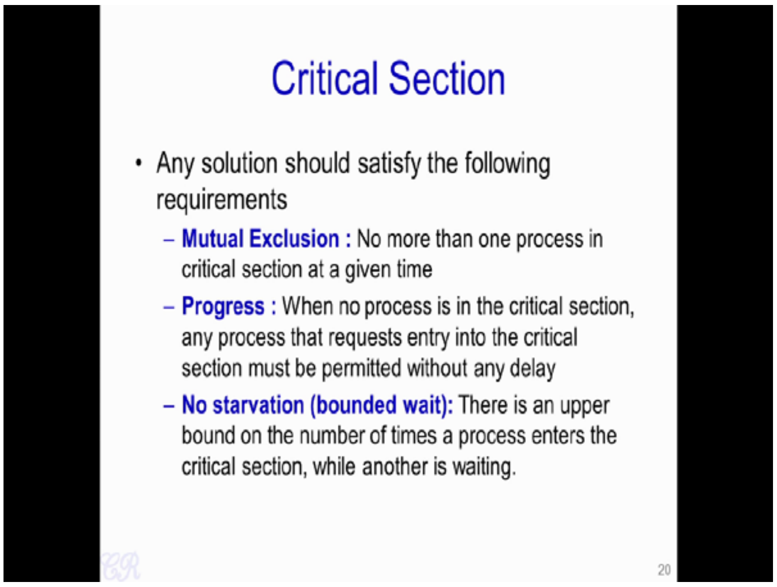
Race Conditions in Multicore

- Multi core
  - Program 1 and program 2 are executing at the same time on different cores

19

Race conditions not only occur in single core system, but also in multi core systems. So, essentially this is quite obvious to reduce that due to the fact that each processor in a multi core system is executing simultaneously it is likely that the shared variable could be accessed by both programs at exactly the same time. Therefore, a race condition in multi core systems is more pronounced compared to the single core systems.

(Refer Slide Time: 09:42)



Critical Section

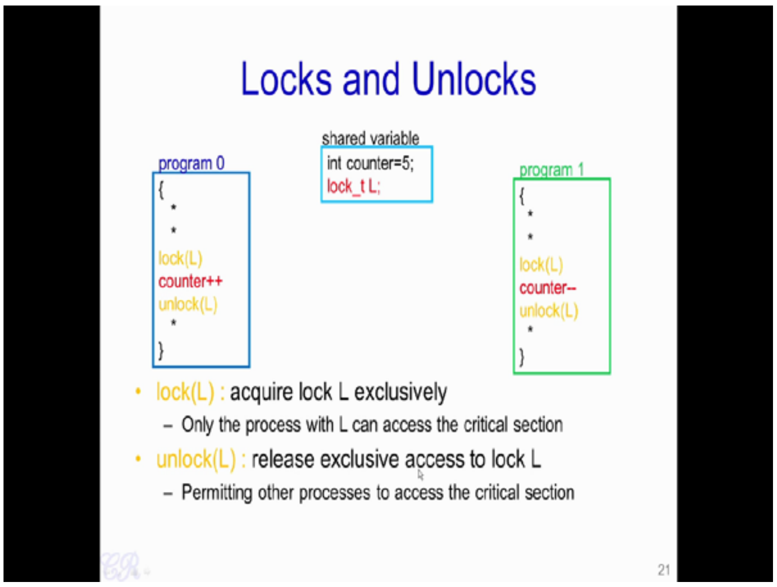
- Any solution should satisfy the following requirements
  - **Mutual Exclusion :** No more than one process in critical section at a given time
  - **Progress :** When no process is in the critical section, any process that requests entry into the critical section must be permitted without any delay
  - **No starvation (bounded wait):** There is an upper bound on the number of times a process enters the critical section, while another is waiting.

20

Now, let us look at how solutions for the critical section problem can be obtained. Any solution for the critical section problem should satisfy the following requirements these are; one mutual exclusion, two progress, and three no starvation or bounded wait. In mutual exclusion the critical section solution should ensure that are not more than one process in the critical section at a given time.

Progress should ensure that when no process is in the critical section any process that requests entry into the critical section must be permitted without any delay. Bounded wait or no starvation means that there is an upper bound on the number of times a process enters the critical section while another is waiting. Essentially, it means that a process should not wait infinitely long in order to gain access into the critical section.

(Refer Slide Time: 10:44)



Locks and Unlocks

shared variable
```
int counter=5;
lock_t L;
```

program 0
```
{
  *
  *
lock(L)
counter++
unlock(L)
  *
}
```

program 1
```
{
  *
  *
lock(L)
counter--
unlock(L)
  *
}
```

- lock(L) : acquire lock L exclusively
  - Only the process with L can access the critical section
- unlock(L) : release exclusive access to lock L
  - Permitting other processes to access the critical section

21

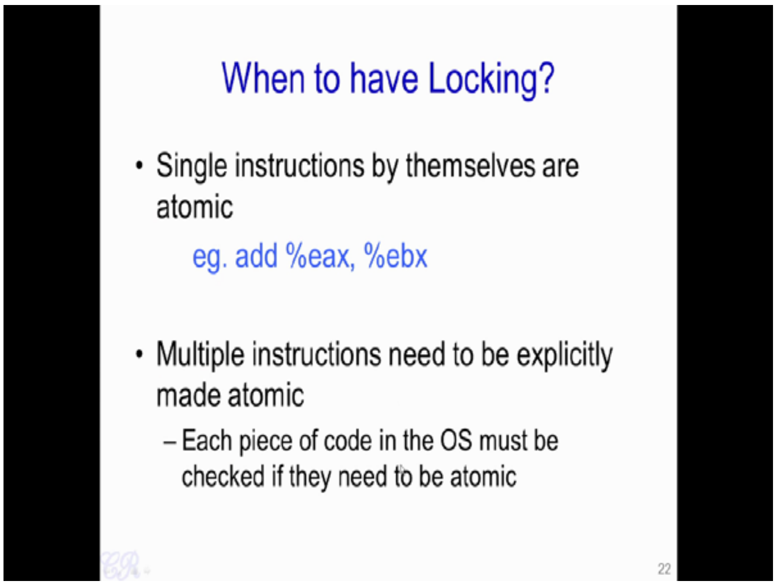
All critical section problems use techniques known as locking and unlocking in order to solve the critical section problem. Essentially, in the solution we would have something like this which is also a shared variable. We define something known as a lock and define it as an L. So before entering into the critical section the program should invoke lock L, and while exiting from the critical section the unlock L should be invoked. Similarly every program which uses the same critical section should lock and unlock before entering and exiting the critical section respectively.

Now what lock does is that it acquires the lock L exclusively. So, after lock completes its execution that is after the function lock L completes the execution it is ensured that exactly one process, in this case only this process enters into the critical section and is present in the critical section. When unlock is invoked the exclusive access to the lock is released, this permits other processes to access the critical section. Now the locking and unlocking should be designed in such a way that the three requirements of the critical sections solution should be satisfied, that is mutual exclusion progress and bounded wait

We had already seen mutual exclusion and let us see progress. So, progress means that let us say program 0 is not in the critical section and let us say program 1 has come here and it has requested the lock. So progress means that, since no other program is in the critical section so it should be given the lock immediately. So, program 1 should get exclusive access to the lock.

Bounded wait means this particular scenario. Let us say program 0 is present in the critical section while program 1 has requested the lock. There is an limit on the amount of time that program 1 has to wait before it gets access into the critical section, that is the solution should ensure that program 0 unlocks L and only then will program 1 enter into the lock. So, there is a bound on amount of time that program 1 waits if it requests the lock while program 0 is already in the critical section.

(Refer Slide Time: 13:26)


When to have Locking?

- Single instructions by themselves are atomic

  eg. add %eax, %ebx

- Multiple instructions need to be explicitly made atomic
  - Each piece of code in the OS must be checked if they need to be atomic

22

The use of lock and unlock constructs in a program will ensure that the critical section is atomic. So when do we need to use this locking mechanisms? Essentially single instructions by themselves are atomic. Instruction such as add, eax to ebx is an atomic instruction and does not require any explicit locking. However, multiple instructions where you have a sequence of instructions and if you need to make them atomic then explicit locking and unlocking is required, so each piece of code in the operating system must be checked if they need to be made atomic or not. Essentially things involving the interrupt handlers need to be checked to be made atomic

In this particular video, we had seen the requirement for locking and unlocking of instructions. So in the next video, we are going to see how such locking and unlocking mechanisms are implemented in systems.

Thank you.