High Performance Computing

Prof. Matthew Jacob

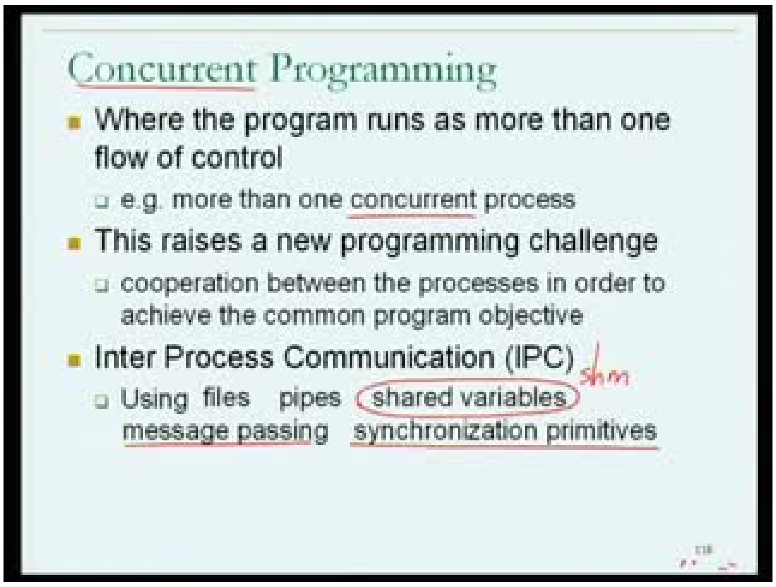
Department of Computer Science and Automation

Indian Institute of Science, Bangalore

Lecture No. # 20

This is lecture 20 of our course on high performance computing. In the previous lectures, we had been looking at a new form of programming which I called concurrent programming. This became of interest to us, once we learned how the operating system which runs on all the computers that we typically use, thus support the idea of multiple processes, even coming out of a single program. Suggesting that the operating systems provide support for the required communication between those processes, in order to achieve their common goal, that it might be in our interests to learn a little bit more about this, since it might be a way to improve the performance of our programs on such computer systems.

(Refer Slide Time: 01:06)



Concurrent Programming

- Where the program runs as more than one flow of control
  - e.g. more than one concurrent process
- This raises a new programming challenge
  - cooperation between the processes in order to achieve the common program objective
- Inter Process Communication (IPC) shm
  - Using files pipes shared variables message passing synchronization primitives

So, in this be for side on concurrent programming which we started in the previous lecture, I mention that concurrent programming describes the situation where the program that you are dealing with runs as more than one flow of control; one way this could be achieved is, by having more than one process, by the program that you execute starting by may be doing a fork or more than one fork to create multiple child processes.

So, the program runs as more than one concurrent process. I had used the word concurrent before; we now understand that it is the same use of the word in concurrent programming. The general idea being that there is activity of across multiple flows of control which, as we now understand do not happen at the same time on the one CPU, but from the perspective of the programmer, it may be necessary to think of them as potentially happening at the same time, for reasons that we began to see in the previous lecture.

So, concurrent programming, unlike the kind of programming we have been looking at before, has an interesting new programming challenge. And that is how to setup the cooperation between the many processes, which are cooperating to achieve the common program objective; and we understood that this has to be provided by mechanisms, some of which may be available only through the operating system.

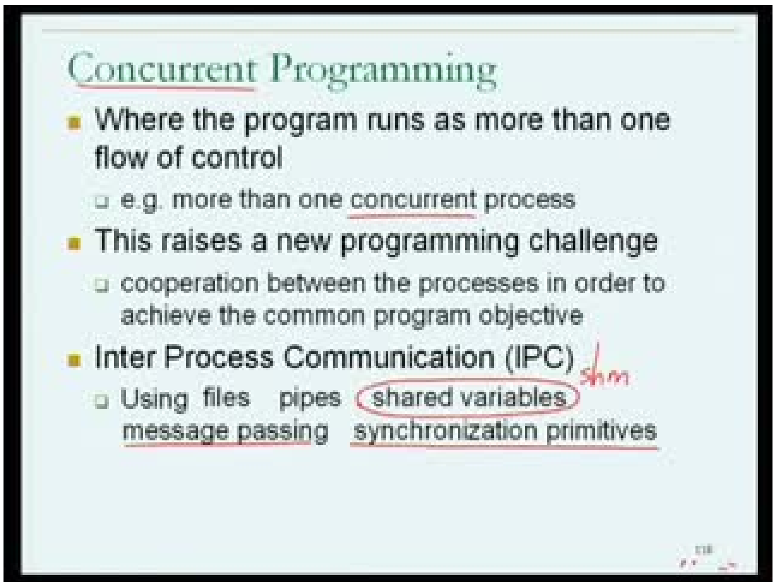
In general, these class of mechanisms would be known as the inter process communication or IPC mechanisms and I gave you some examples of how processes could cooperate. For example, they could cooperate by writing and reading pieces of data in files, which are commonly accessible between the processes. On the other hand, they could cooperate using an operating system mechanism known as a pipe, which superficially looks very much like, but the communication between processes through files would look like; in that, the communicating processes would be reading and writing from file descriptors, in the case of the pipe.

In addition to this, there was the interesting idea that processes could communicate with each other, passing values or sharing values - actually having variables that are shared - commonly accessible by the processes. And this was the deviation from our understanding of the need for processes to be protected from each other, which is why we talked about the operating systems, support for address translation, virtual memory paging, etcetera.

But in the light of concurrent programming, the potential of using shared variables for inter process communication is clearly what looking into, which is why we suspect that operating systems provide support for this as well, for defining regions of the virtual address spaces, which are actually common between processes; and this might be available through some form of a shared memory, control facility through an operating system.

In addition to this, address the possibility of processes being able to communicate through operating system provided mechanisms called message passing and receiving. So, one process could explicitly send a piece of data to another process, which could explicitly receive it, and that would be known as a message passing inter process communication.

(Refer Slide Time: 01:06)



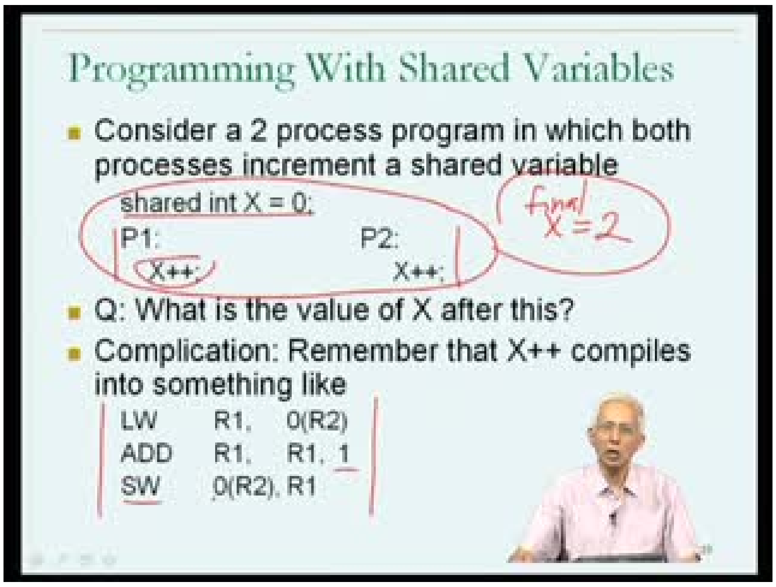
Concurrent Programming

- Where the program runs as more than one flow of control
  - e.g. more than one concurrent process
- This raises a new programming challenge
  - cooperation between the processes in order to achieve the common program objective
- Inter Process Communication (IPC)
  - Using files pipes shared variables message passing synchronization primitives

And we also looked at the idea of, they are being situations where may not be necessary for the cooperating processes to communicate values to each other, but rather be sufficient if they can synchronize their activities and that may be supported through the availability of certain synchronization primitives.

Now, we will be seeing more about message passing and synchronization primitives towards the end of this course. But in this part of the course, I spent some time beginning in the previous lecture, on the complications that could arise if one is setting up concurrent programs and making that communication happen through the facility of shared variables, just to make sure we understand the nature of the problem that could arise in dealing with shared variables.

(Refer Slide Time: 04:52)



Programming With Shared Variables

- Consider a 2 process program in which both processes increment a shared variable

```
shared int X = 0;
P1:                 P2:
   X++;                X++;
```

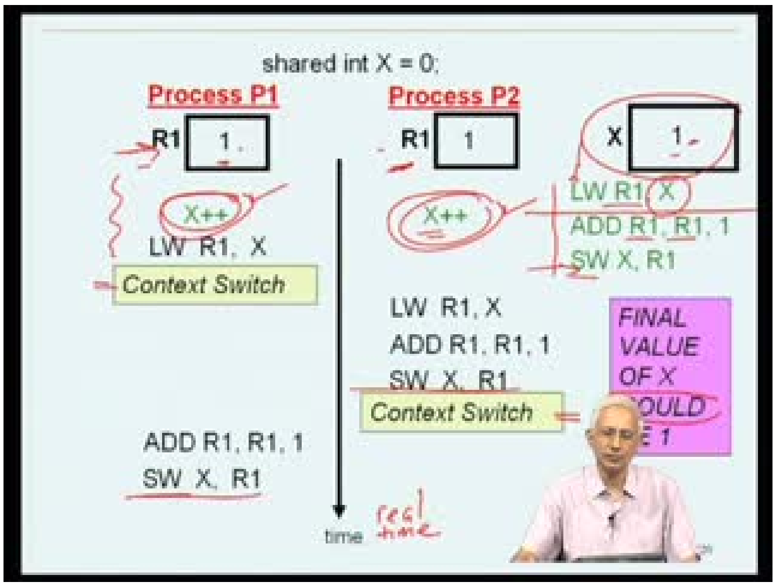
- Q: What is the value of X after this?
- Complication: Remember that X++ compiles into something like

```
LW      R1,    0(R2)
ADD     R1,    R1, 1
SW      0(R2), R1
```

We looked at the very simple example of a two process program, in which there is one shared variable which is initialize to 0 and all that each of the processes is doing is incrementing the variable. And as I pointed out last time for person writing such a program, it would appear logical to assume, that since the shared variable X is incremented twice, the end result should be that X should end up with the value 2. But it turns out, that this may not be what happens if one writes this particular code, without taking into account the complications of concurrent programming.

So to fully understand what was happening here, we viewed this piece of code not as X plus plus, but rather as what it would compile into. Let us say, the mips 1 assembly or machine language; this is what we came up with. The variable X would have been loaded into a register; from the register the addition of one could be done, and from the register, the variable X could be updated in memory using a store word instruction.

(Refer Slide Time: 06:12)



shared int X = 0;

Process P1

R1 1

X++

LW R1, X

Context Switch

ADD R1, R1, 1

SW X, R1

Process P2

R1 1

X++

LW R1, X

ADD R1, R1, 1

SW X, R1

Context Switch

X 1

LW R1, X

ADD R1, R1, 1

SW X, R1

FINAL VALUE OF X OULD E 1

time

real time

So, the x plus plus is actually three machine instructions. So, to understand how it could be possible for the value of X not to end up as 2, I will step through this to make sure this is very clearly understood. Initially, the shared variable X has a value of 0. It is initialized to have a value of 0, so let me draw a box which has representing the memory location X, in terms of the shared variable; and initially, it contains the value 0. Subsequently, I have two processes: process P1 and process P2, which are both executing machine instructions corresponding to the C construct X plus plus.

Now, let us assume that this, as we saw compiles into the sequence of instructions, load word add and store word as shown on the right hand side. In the right hand side, I am showing a slide deviation from the mips 1 assembly language notation and that I am not showing the address of X in base-displacement addressing mode, but this is slightly more readable in a current context, since it clearly indicates that we are talking about loading the variable X; and we understand that this could be easily corrected using the base-displacement address of the variable X.

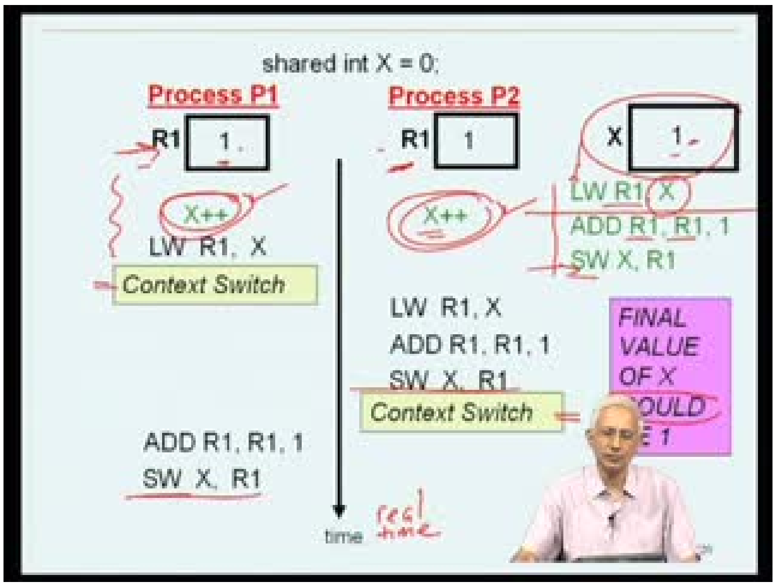
So, let us assume that, when this concurrent program made up of these two processes, after the initialization of X to 0. Let us suppose a process P1 is the first process to execute on the CPU; there is only 1 CPU, therefore only one of them can be running at a time and we are going to assume the process P1 is the one which is running.

Now, we do need to note that, as far as this particular code segment is concerned, in addition to the contents of memory location X or the variable X, we do have to keep track of the contents of the register R1. And note that each of the processes will have its own perspective on what the contents of register R1 is; as we have learnt, there is only one processor, it is shared among all these processes, on the one of which can be running at a time.

Therefore, the contents of R1 have to be viewed as being specific to a process. So, I showed two possible situations, as far as R1 is concerned; the one on the left corresponds to what R1 should contain, when process P1 is running; and the one on the right corresponds to what R1 should contain, when process P2 is running.

So, we starting by assuming the process P1 is running; and therefore, the first thing that we will do is, we will consider events along a time line. Last time, we learned about different possible time lines. In this particular case, we are trying to show on the time line what activity happens on the CPU; and the activity could happen on the left side, if process P1 is running, or it could happen on the right side, if process P2 is running. And further, since we now know the name to give for that kind of time, I will more correctly label this accesses not just as time, but as real time as supposed to virtual time. And note, I could just as well have labeled it, wall clock time or a laps time, all of which as synonymous terms with real time. So, process P1 is on the CPU and running.

(Refer Slide Time: 06:12)


shared int X = 0;

Process P1 | Process P2 | X: 1

R1: 1 | R1: 1

```
X++                      X++    LW R1, X
LW R1, X                        ADD R1, R1, 1
Context Switch                  SW X, R1
                         LW R1, X
                         ADD R1, R1, 1
                         SW X, R1
                         Context Switch
ADD R1, R1, 1
SW X, R1
```

FINAL VALUE OF X COULD BE 1

Let suppose, that it starts by executing the load word instruction. We know that the effect of this instruction is to copy the contents of the variable X into register R1 for the perspective of process P1. Therefore, the effect is to get the value 0 copied from memory into register R1.

Now, let us suppose that process P1 had been running for some amount of time prior to execution of the load word instruction; and it is at this point in time that the operating system in its pre-emptive process scheduling strategy chooses to context switch from process P1 to process P2, so this is where context switch occurs.

The effect of the context switch is going to be, that one of the other processes in this case process P2 is identified as one which should run. Subsequently, the hardware state associated with process P1 will be stored somewhere in memory and the process state associated with process P2 will be inserted into the hardware. In other words, the hardware register R1 will now contain whatever value R1 should contain from the perspective of process P2.

So, the picture on the right suddenly becomes the current picture, as far as the processor is concerned. So, process P2 now starts its execution on the CPU. Again, it may have been running before, but let suppose that, when process P2 resumes execution, a 2 is trying to execute this piece of code. So, it starts by executing load word R1 X, as a result of which the value in memory location X, which is the value 0 gets copied into R1. It then executes add R1, R1, 1, which adds the value 1 to the current contents of R1 which is 0, resulting in a new value of R1 which is 1, so R1 gets updated to 1; it then moves to the store word instruction, where it updates main memory with this value of the register R1. Therefore, the main memory variable ends up with the value 1.

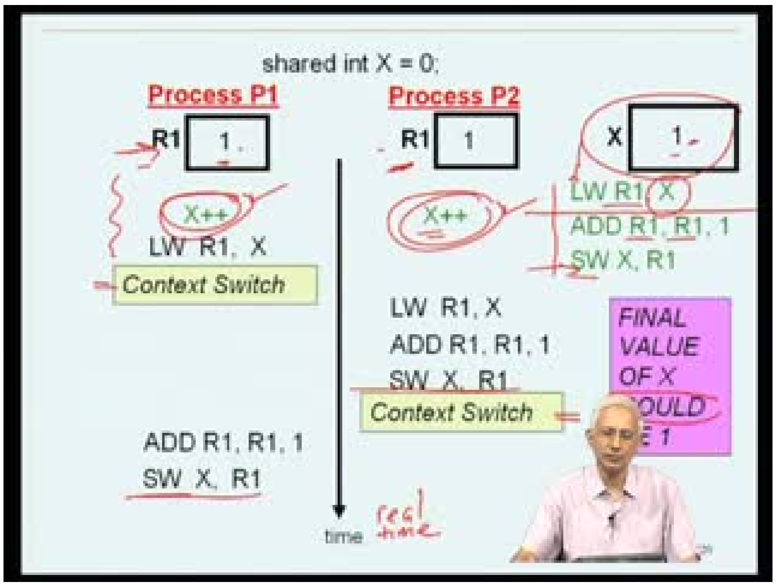
Now, let us suppose that once again, it is about this point in time that the process gets context switched. What is going to happen now? Hardware state of process P2 will be saved in memory; and let suppose the process P1 is the ready process, that is now selected for running. Then, it is hardware state will be loaded, and therefore, the current content of register R1 will once again be 0, because that was the correct value of register R1, as far as process P1 is concerned.

Now, when process P1 resumes execution, it finds or starts running on the processor again; the value in register R1 will be 0. Process P1 has already finished executing the load word instruction, if therefore will proceed with the add instruction; as a result of which the current contents of R1, which is from the perspective of process P1, 0 will be added to 1 yielding a value of 1, which will be the new value of register R1.

And then, process P1 will execute the store word instruction, where it will update the main memory with this value 1. Value in main memory is 1 right now, the new value 1 will be over written over that 1 and the net result is going to be, that after both processes have executed the store word instruction. In other words, after both of them have finished executing X plus plus… Note, there at this point in time, process P1 has finished with X plus plus and process P1 has already finish with X plus plus.

Therefore, the net result after both processes have incremented the variable X is, what we see on the right hand side. The value of the memory with location is 1, not 2 as we have suspected. Therefore, this is happened as we can now understand, because of this interleaving of the activities of process P1 and P2 and the context switch is that might occur in between. The net result being that the impact on the shade variables may not be what we had thought they were from our understanding of the program at the C level. The simple understanding that X plus plus executed twice, will cause the value of X to go up by 2.

(Refer Slide Time: 06:12)


shared int X = 0;

Process P1

R1 1

X++

LW R1, X

Context Switch

ADD R1, R1, 1

SW X, R1

Process P2

R1 1

X++

LW R1, X

ADD R1, R1, 1

SW X, R1

Context Switch

X 1

LW R1, X

ADD R1, R1, 1

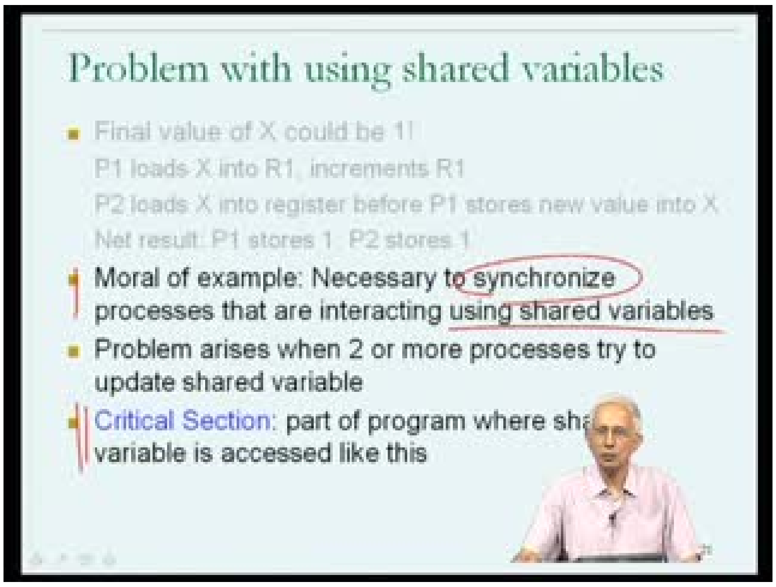
SW X, R1

FINAL VALUE OF X OULD E 1

So the final value of X could be 1. Again, you should note that, for this particular concurrent program, it is also possible that the final value of X could be 2. If the context switch is that I talked about here had not occurred at these carefully managed the points in time, then each of these processes would independently have incremented X; and therefore, the final value of X could have been 2. But it is possible that the context switch is could occur as I have outlined; and that is why, I point out that the final value of X could in fact be 1.

So this is an example of non-determinism. If I were to write this program in this way - this concurrent program - I would have no control, whether the final value of X would be 1 or 2 and it is not often the case that programs which produce non-deterministic results or views to us. Typically, we write the program which will produce results in a predictable way from the perspective of the data, which we have providing as input to the program. So, this is not always the desirable feature of the non-determinism of a program

(Refer Slide Time: 14:02)

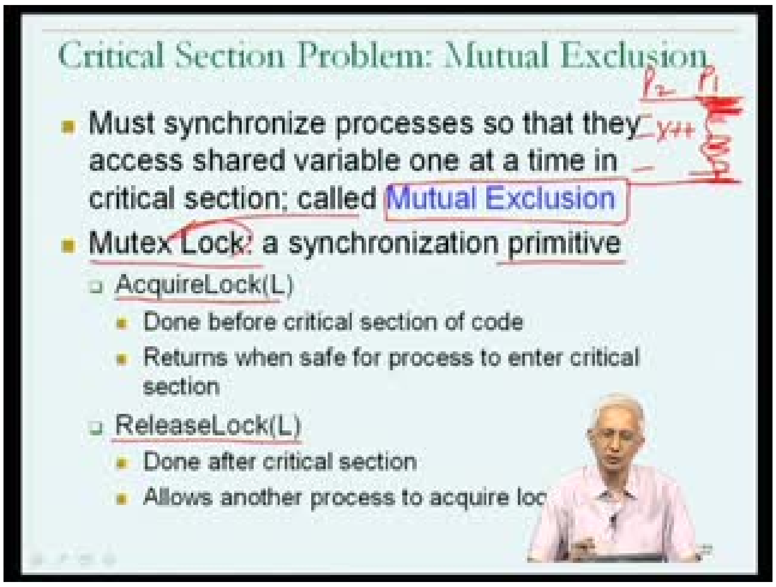


Problem with using shared variables

- Final value of X could be 1!
  P1 loads X into R1, increments R1
  P2 loads X into register before P1 stores new value into X
  Net result: P1 stores 1, P2 stores 1
- Moral of example: Necessary to synchronize processes that are interacting using shared variables
- Problem arises when 2 or more processes try to update shared variable
- Critical Section: part of program where sha variable is accessed like this

So, this was the problem, arising out of the use of shared variables, and I do the moral that it is in fact necessary to synchronize the activity of processes, when they are modifying shared variables. And that potentially, one has to view the regions of a concurrent program, where shared variables are modified or accessed in this way. As the critical parts of the concurrent program, around which some form of synchronization of the two processes in this example P1 and P2 may have to be done, in order to produce the result, which I considered to be desirable or the one for which I had written the program.

(Refer Slide Time: 14:40)



Critical Section Problem: Mutual Exclusion

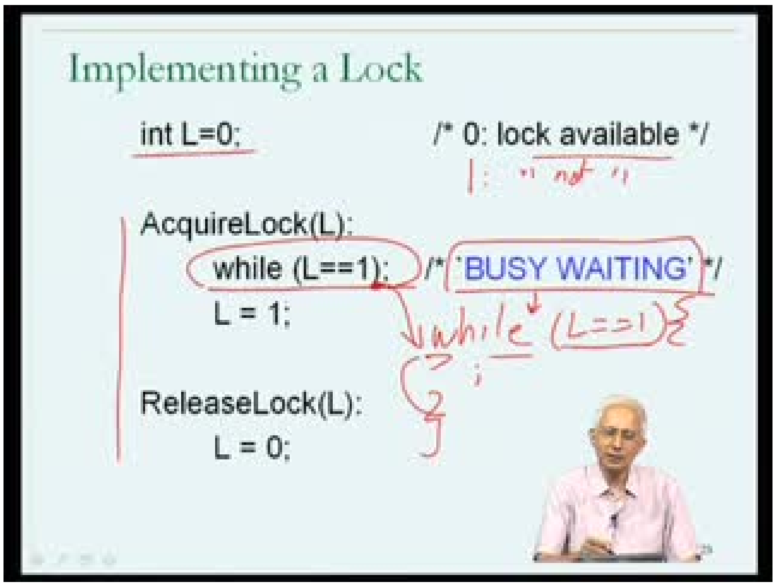
- Must synchronize processes so that they access shared variable one at a time in critical section; called Mutual Exclusion
- Mutex Lock: a synchronization primitive
  - AcquireLock(L)
    - Done before critical section of code
    - Returns when safe for process to enter critical section
  - ReleaseLock(L)
    - Done after critical section
    - Allows another process to acquire loc

So, the particular synchronization requirement here, which were actually remove the problem that we had seen is, if I had in marked each of these two critical sections, in other words, the region of code where the incrementing of X was being done by process P1 or by process P 2, as a region in which I wanted a guaranty that no more than one process should be executing concurrently, which is where the term mutual exclusion comes from. In other words, I want to guarantee that, within that critical section, whatever X plus plus compiles into, I will say is the critical section, the region of code where a shared variable X is being accessed. We saw that this compiled into three statements, therefore the critical section would contains those three statements.

So, what we are saying here is, the mutual exclusion requirement is, I require that either process P1 could be executing those three statements over process P2 could be executing those three statements, but I do not want process P1 to be half way through when process P2 enters that region of its program. Hence, the term mutual exclusion, we want the processes to execute in critical sections mutually exclusively of each other; in other words, only one at a time.

Now, it is conceivable to have operating system primitives or basic functionalities provided for this purpose and one such is typically known as a mutual exclusion lock. Mutex stands for mutual exclusion and lock is a word suggesting that it is a mechanism through which one can safely achieve this, essentially, by locking the entry into that critical section from one process, if it is not safe. So, the word lock is appealing in that sense. And I describe the lock is having two function calls associated with it; one to acquire a lock; in other words, to be use before one enters the critical section and the other to release the lock; in other words, to indicate that one is living the critical section and another process if it so desired could enter the critical section.

(Refer Slide Time: 16:37)



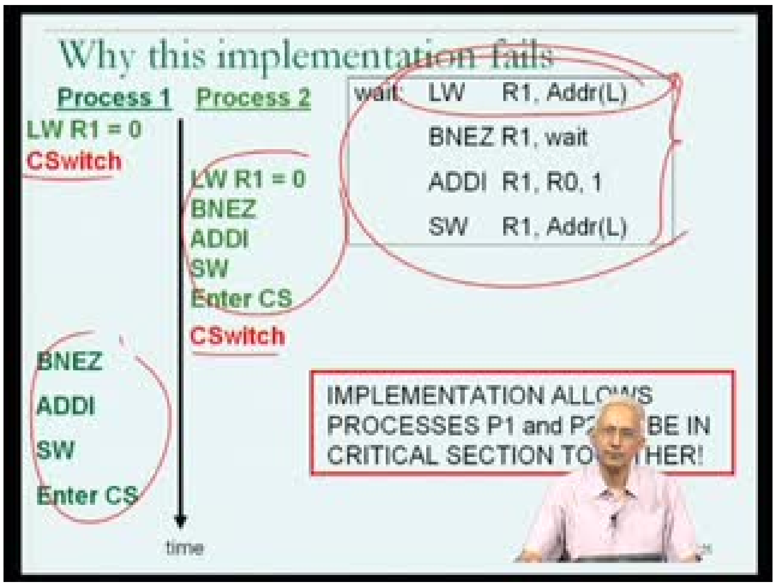
Implementing a Lock

```
int L=0;                /* 0: lock available */
                           1: " not "
AcquireLock(L):
    while (L==1);       /* 'BUSY WAITING' */
    L = 1;                 while (L==1){
                              ;
ReleaseLock(L):            }
    L = 0;
```

We came up with this tentative mechanism for implementing a lock using an ordinary variable. And the idea was that, until if we viewed the implementation of the lock as working by ensuring that, if the value of the lock is 0, that means, that the lock is available; in other words, it can be a acquired by a process which request; where is it value of the lock is 1, that means, that is lock is not available, meaning that it is not available and a process trying to acquire the lock at that point in time should not be allow to enter it is critical section and suggested this implementation.

But unfortunately, we are going through this implementation along the lines of what we did with that simple piece of X plus plus code; we learn that this is not a correct implementation of a lock. But we did learn an interesting idea over here, the notion that, in order to keep a process waiting, until it became safe to enter the critical section, one could use this concept of busy waiting, where process was essentially executing this piece of code, which likely to look at again is essentially a loop. Since, it is executing no instructions or no statements within the loop, I show it as a one line loop, but just know that this is what it actually means; it is a loop in which there are no statements in the body. But merely the checking of the condition, and as long as the condition is true, it keeps on executing this loop; in other words, its keeps on being busy executing 1 or 2 instructions related to this, whatever this complies into, but it is really waiting for the value of L to change to 0. Hence, this kind of a loop would be called a busy wait loop.

(Refer Slide Time: 18:24)



Why this implementation fails

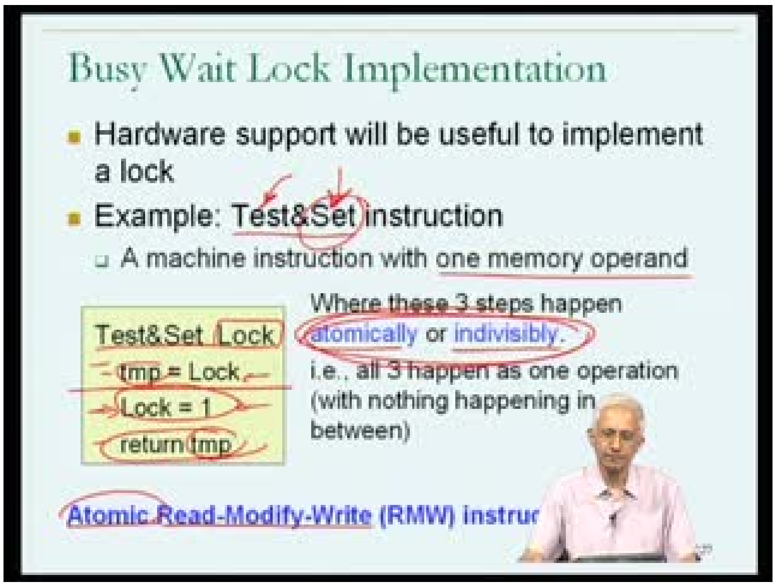
Process 1	Process 2
LW R1 = 0	
CSwitch	
	LW R1 = 0
	BNEZ
	ADDI
	SW
	Enter CS
	CSwitch
BNEZ	
ADDI	
SW	
Enter CS	

```
wait: LW     R1, Addr(L)
      BNEZ   R1, wait
      ADDI   R1, R0, 1
      SW     R1, Addr(L)
```

IMPLEMENTATION ALLOWS PROCESSES P1 and P2 TO BE IN CRITICAL SECTION TOGETHER!

And unfortunately we learned that this does not work, when we step through this in terms of its machine language instructions; this is one possible compilation of that sequence of 2 C statements into mips machine language. We saw that the situation very similar to that with the X plus plus code could happen. In other words, process P1 could start with this sequence of code and then get context switched out; and then process P2 could execute all of the codes successfully, and then get context switched out. And then, if control is transfer back to process P1, then process P1 could in fact execute the rest of the code successfully, and the net result could be the both the processes seem to have acquire the lock and enter the critical section.

(Refer Slide Time: 19:20)



Busy Wait Lock Implementation

- Hardware support will be useful to implement a lock
- Example: Test&Set instruction
  - A machine instruction with one memory operand

```
Test&Set Lock
  tmp = Lock
  Lock = 1
  return tmp
```

Where these 3 steps happen atomically or indivisibly.
i.e., all 3 happen as one operation (with nothing happening in between)

Atomic Read-Modify-Write (RMW) instruc

What does this mean? It does not mean that locks are not a safe mechanism to make shared variables a good idea; it just means that this implementation of the lock is wrong. Now, in order to implement the lock correctly, it might be useful to have some hardware support and one form of hardware support which is often provided in instruction sets is an instruction which may look like this, I am going to describe one particular instruction which is sometimes provided. It typically goes by the name test and set. Now, as a name suggest it is in instruction, which will actually do two things: it will test or read the value of a variable and it will also as part of the same instruction; set are change the value of that variable.

So test and set is the name of what, we are going to assume that it is the name of a machine instruction. For example, it might be mips 1 machine instruction and it has only one memory operand; it does not take all its operands out of registers, but rather has one machine; this machine instruction has one memory operand, which is the memory location which is going to be read and updated. What does this instruction do? Now I am going to describe what this instruction does, using this yellow code segment over here. I am showing it you and see, but just note that this is the functionality of the instruction test and set.

So, you look at this there are three steps. In the first step, over here I am referring to the memory operand of the instruction by the word lock, so this is the memory operand. So, initially, the first thing that it does is, it reads the value of lock and remembers it. In C, I would describe this by saying that it stores the value of lock into a temporary variable, but this is just you know that it is reading value of lock.

In the second step, it updates that memory location to the value 1, that is the set that we are talking about. And as a whole what the instruction has as a return value is, the value that was stored in this temporary variable; in other words, the old value of lock. So, this instruction is updating lock to 1 and returning whatever the value of lock was before; that is what this is meant to achieve.

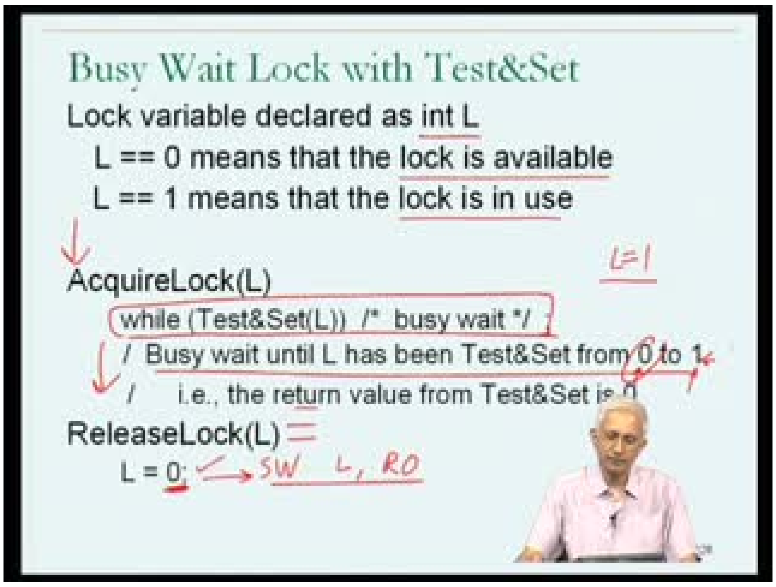
Now, the important thing about this instruction is that, though it does this, what look like three operations, it actually does these three operations as one; and the word which is used to describe this, is to say that these three steps of the test and set instruction are done atomically or indivisibly. Essentially, what it means is, all three of these happen as one operation, with nothing conceivably happening in between.

Therefore, either all three of them happen or none of them happens; that is, the meaning which we should infer. It is not possible for a process to execute test and set instruction and get context switched, after it had done the first step of the instruction, that is not possible and the guarantee is given by the hardware. So, you can now see that this kind of hardware support is going to make the implementation of a mutual exclusion lock easier, because it gives guarantee, that context switch is cannot occur between these three operations.

Now one general point, I mention the test and set instruction as an example of the kind of instruction that could be included in an instruction set, in order to support the implementation of busy wait locks or synchronization primitives in general; it is only an example, they could be others. In general, instructions of this kind which serve this kind of a purpose would be described as atomic; for this atomic indivisible property, read modify write, because if you think about it, this instruction is reading the value of lock and subsequently modifying and writing - updating the memory location. Therefore, these are the important characteristics of the instruction, that the three steps that we saw are atomic - indivisible - happening as if they were one.

Just as a closing note on this particular instruction, you may be wondering, you said that this instruction has only one memory operand. How then does it return a value? And the answer to that is, that if we were talking about a test and set instruction in, let say the mips 1 instruction set and they had to be a return value, the return value could be returned in a register.

(Refer Slide Time: 23:46)



Busy Wait Lock with Test&Set

Lock variable declared as int L

L == 0 means that the lock is available

L == 1 means that the lock is in use

L=1

```
AcquireLock(L)
    while (Test&Set(L))  /*  busy wait */ ;
    / Busy wait until L has been Test&Set from 0 to 1
    /     i.e., the return value from Test&Set is 0
ReleaseLock(L) =
    L = 0;  → SW    L, R0
```

So, the form of the instruction could be test and set R1 lock, and the return value destination register with then contain the previous value of memory location, which was the memory operand. Now, given such an instruction, how would we implement the busy wait lock?

Now, as before as we try to do with our simpler implementation a while back, we will assume that there is a variable, an integer variable let say, associated with a lock, and that this is the memory location which is going to be test and setted or tested and set. The way that we interpret the value of the variable L will be the same as we were doing up to now; in other words, it will be initialize to 0. If it has a value of 0, which means the lock is available; and if it is has a value of 1, that means that the lock is not available or in use.

So, we can now go ahead and try to figure out, what has to be present in the acquire lock function and the release lock function. These are the two functions, that concurrent programmers who want to provide support for their critical sections will have to use and we were just talking about how these things could be implemented.

So, this is not something that we have to worry about is programmers, these are general awareness of what is happening in the computer system. So, the releasing of the lock is very clearly easy. The release lock function will be executed by a process, when it is getting out of its critical section. In other words, changing the situations from one way the lock was in use, to one way the lock is available, which means that, when the release lock function is executed the value of L was 1 and now merely has to be set to 0, which can be done with a simple assignment; so release lock is easy. The lock will be set to 0. Basically, this L equal to 0 will amount to a single store word instruction, where into the variable L the contents of register R0 will be return.

Let us note that R0 is the mips 1 - I am using mips 1 like notation here; R0 contains the value 0. This is another example, the fact that R0 contains the value 0 is useful; the alternative would have been, I would have to generate the value 0 in a register, which may have required many instructions. And once again I may have had a problem with a critical section, series of instructions which are accessing the value L. So, it turns out to be nice set, we have a 0 available to us in the register R0. So, this compiles into a single statement.

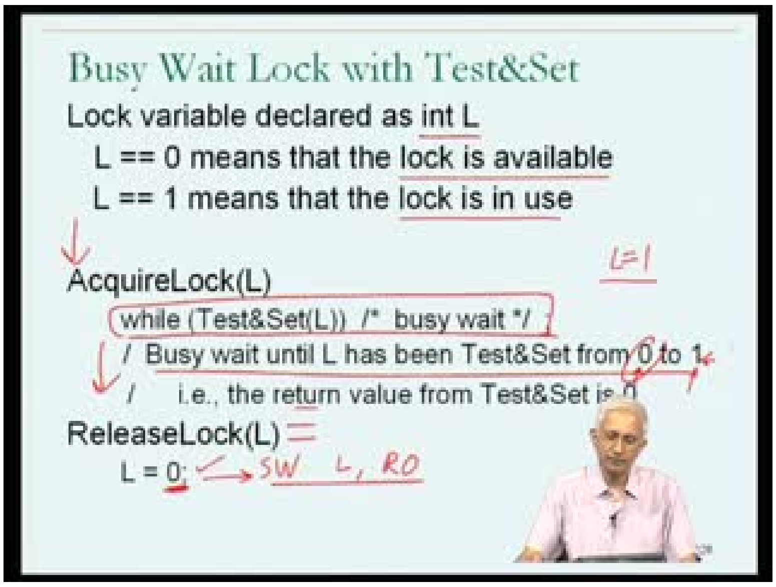
Now, what about acquire lock? What do we want to do in acquire lock? Now, we know that acquire lock we want to busy wait. We want of any process which executes acquire lock should be stuck in a acquire lock - busy waiting - until the value of L has been successfully changed from 0, which means from a condition where it was available to a condition where it was in use.

We are going to change the value of the lock L from 0 to 1 using test and set, but we want to have a busy wait loop, because it is possible, that when a process executes acquire lock, the value of L is already 1. In which case, it can test and set L, which would not the value of L. But there will be a return value of 1, if the value of the lock was already 1; and therefore, that process should continue to busy wait, until this is the case, that it has successfully test and set L from a value of 0 to a value of 1.

In other words, whether we need to do this busy waiting is test and set L which will change the value of L to 1. But since it may have been 1 or 0 before, we want to keep on testing and setting L, until the return value of test and set of L is equal to 0, which means that the old value was 0.

So, this is what the implementation would look like; we have once again a busy wait loop. So, as long as test and set of L returns 1, we want to continue executing that loop, this is the busy wait loop. But as soon as the return value from test and set of L is 0, I want to get out of loop, I have successfully satisfied this requirement. The process which gets a return value of 0, is the first which has found that the lock available and changed its status to not available.

(Refer Slide Time: 23:46)

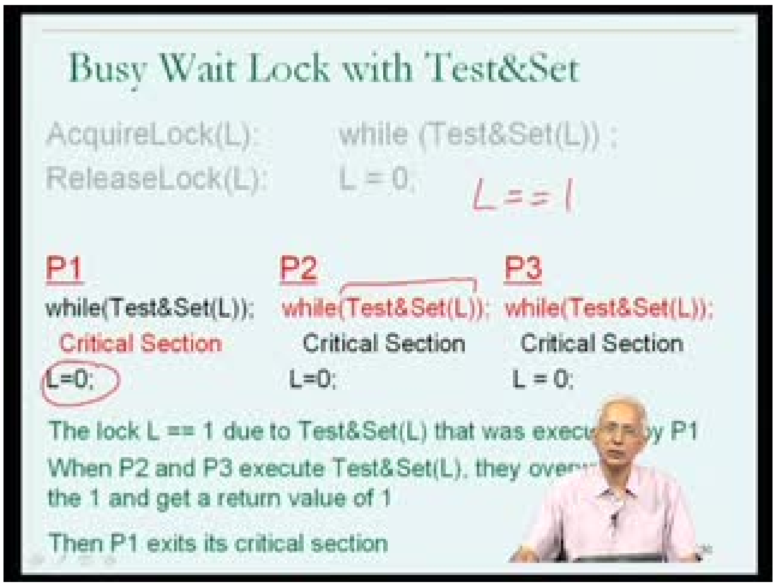


Busy Wait Lock with Test&Set

Lock variable declared as int L
 L == 0 means that the lock is available
 L == 1 means that the lock is in use

```
AcquireLock(L)
    while (Test&Set(L))  /*  busy wait */
    /  Busy wait until L has been Test&Set from 0 to 1
    /     i.e., the return value from Test&Set is 0
ReleaseLock(L)
    L = 0;
```

(Refer Slide Time: 27:58)



Busy Wait Lock with Test&Set

AcquireLock(L): while (Test&Set(L)) ;
ReleaseLock(L): L = 0;

L == 1

P1	P2	P3
while(Test&Set(L));	while(Test&Set(L));	while(Test&Set(L));
Critical Section	Critical Section	Critical Section
L=0;	L=0;	L = 0;

The lock L == 1 due to Test&Set(L) that was executed by P1
When P2 and P3 execute Test&Set(L), they over[illegible]
the 1 and get a return value of 1

Then P1 exits its critical section

So, we can actually just make sure we understand what is happening by running through the scenario, where assuming that our implementation of acquire lock is, what I have just described while test and set of L and busy wait on this; and the implementation of these lock is what I have just described, set L equal to 0. What will happen, if they are a lot of processes trying to acquire the lock? This example I am showing three processes.

Now, we need to start off with some assumption. So, in each case the process is going through the protocol of acquiring the lock, before it enters its critical section; and then releasing the lock, when it is done with this critical section. So, this was meant to be the protocol for the use of the lock primitives. Now, let us suppose, for the movement that process P1 is inside its critical section, it was the first process which tries to acquire the lock, it successfully acquired the lock and is therefore executing in its critical section.

And I will also assume that the processes P2 and P3 are trying to acquire the lock, in order to enter their critical sections. Now, if this is the case, then what I will notice is that the process P1 is inside its critical section and both process P2 and P3 are executing the acquire lock function, which means that they are executing while test and set of L; they are the loop the busy wait loop while test and set of L.

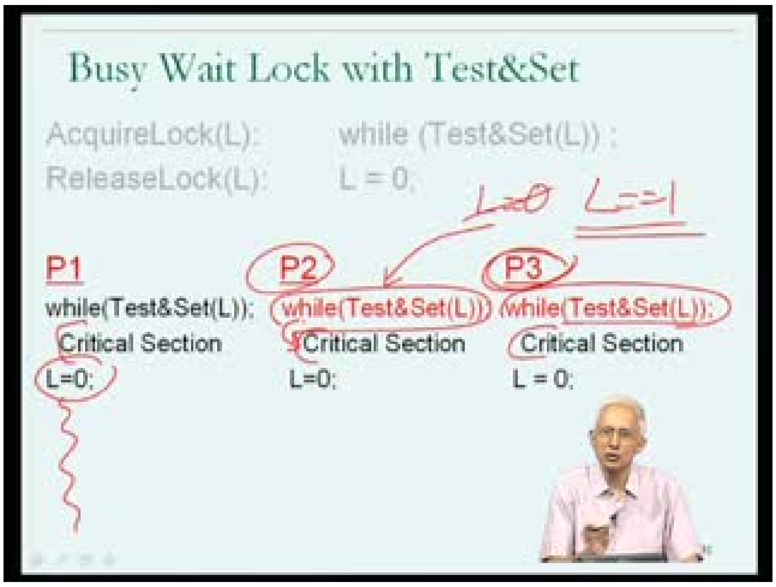
Now, what happens the process P2 or process P3? Now since process P1 had successfully set the lock, the value of the lock is equal to 1; and every time process P2 or

P3 run on the processor, they attempt test and set L. You look all that the test and set instruction will read the old value of the lock variable, which is 1, and that will be the return value of the call. Subsequently, it will modify the value of L to 1, which is not really a modification, because old value of L was 1. Hence, as long as process P1 is in the critical section, both process P2 and P3 will busy wait on their loops, repeatedly changing modifying the value of L to 1, but always finding that the old value of L was 1 and therefore unable to escape from their busy wait loops.

So, we know this lock is equal to 1, because it was set to 1, when process P1 enter it is critical section. And further that when processes P2 and P3 repeatedly execute test and set on L, they merely overwrite that 1 with 1 again.

Now, this situation will continue, as long as process P1 is in its critical section. So, note that both process P2 and P3 and any other processes which may be trying to use the same lock, will all be blocked or rather busy waiting and unable to enter the critical sections, because this implementation at this point seems to be doing the right job.

(Refer Slide Time: 31:01)



Busy Wait Lock with Test&Set

```
AcquireLock(L):    while (Test&Set(L)) ;
ReleaseLock(L):    L = 0;

P1                        P2                        P3
while(Test&Set(L));       while(Test&Set(L));       while(Test&Set(L));
  Critical Section          Critical Section          Critical Section
L=0;                      L=0;                      L = 0;
```

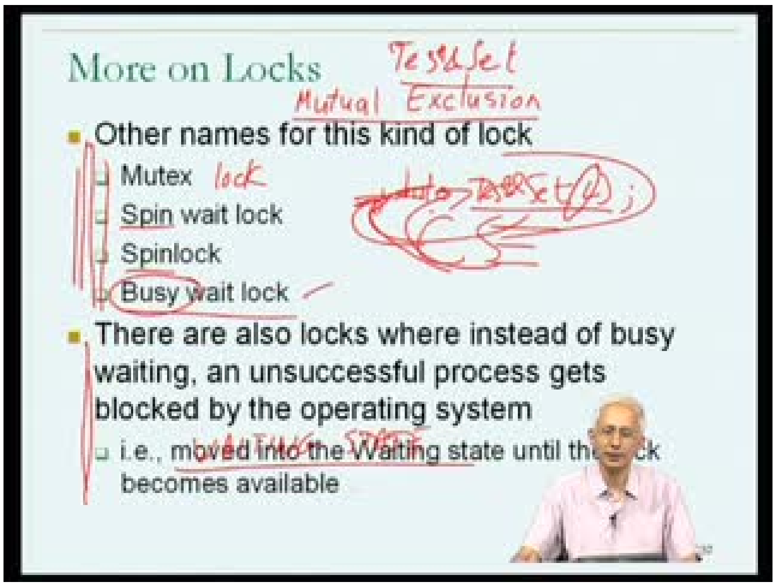
When will this situation change? The situation can only change, when process P1 exits its critical section. So, after it finishes its critical section, it will come to the release lock function. The effect of it executing the release lock function is, that its sets the value of L to 0 and it proceeds with its other activity, but the value of L is now 0.

Now, in the main while, the next time either process P2 or P3 runs, it is still in a situation where it is running while test and set of L. And depending on which one these happens to be running first, it will notice that the value of L which is return by test and set of L is 0, and indivisibly, because it is doing so with the test and set instruction will change that value to 1.

And hence, we will get out of the busy wait loop, enter its critical section and so on. So, depending on which of P2 and P3 actually runs next, after process P1 has reset L to 0. One or the other of the processes will enter its critical section and the other unfortunately will not find that the value of L has change from 1.

And the same will actually be the situation regard this of how many processes I have. I could have 100 processes, all trying to acquire this lock and the same guarantees will hold. Therefore, this the actually seems to be a correct implementation, in terms of the requirement that, only one process should be allow to enter it is critical section, if we use the protocol described. In order to do this safely, we required a little bit of help; we need that individual instruction, in the middle of which, no context which was possible.

(Refer Slide Time: 32:32)


More on Locks

- Other names for this kind of lock
  - Mutex
  - Spin wait lock
  - Spinlock
  - Busy wait lock
- There are also locks where instead of busy waiting, an unsuccessful process gets blocked by the operating system
  - i.e., moved into the Waiting state until the lock becomes available

Now, just continuing on the subjects of locks a little bit, the kind of lock which we have just seen actually goes by a few other names as well.

Some people just refer to it as a Mutex; you will remember that the word Mutex was itself just an abbreviation of mutual exclusion. This is essentially the requirement that we had, in order to make the regions of our program where we access shared variables safe. The requirement was we wanted only one process to conceivably be in such a critical section at a time, in order to avoid the kinds of problems we saw with the X plus plus scenario.

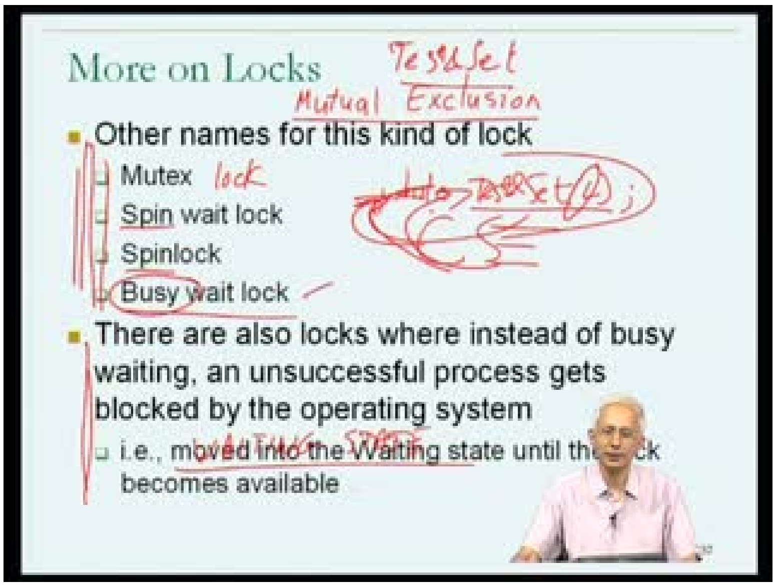
So, some people may refer to a lock of the kind that we just saw as a Mutex; otherwise, may refer to it as a Mutex lock. Still others may refer to it as a spin wait lock or a spin lock, because the kind of busy waiting that we saw could be as well, described as this spinning on a lock, in terms of there was a very tight while loop. And conceptually, view the process which is executing this while loop as repeatedly executing a very small loop which could be described by the word spin. And as I myself have used it, sometimes refer to as a busy wait lock, so many terms for what is basically the same idea.

Now, there are other kinds of locks, which could be used for the same purpose. And some of them might be different in the way that they keep a process from entering its critical section. In the case of the busy wait lock that we saw, the mechanism that was used, was the keeping a process running, executing on this loop, until it became safe for it to enter it is critical section. And alternative could have been, that if it was found that it was not safe for this process to enter the critical section, then the operating system could be invoked to block that process; in other words, to put it into a waiting state. So, what I mean by blocking is, for the operating system to actually moving that process to the waiting state, until the lock becomes available.

Now, conceptually one can see that there is a distinction between these two. In the one case, the process is actually keeping the processor busy, because the process which is busy waiting is executing instructions. There is going to be a test and set instruction, and after that, there is going to be an instruction to decide, whether or not to branch back; so, at least two instructions.

Depending on the instruction set, it could even be a few more. And in some sense, one could argue that when a process is busy waiting, it is not really progressing towards its objective. It is keeping the CPU busy; it is keeping the CPU utilization high, but it is waiting for the value of the lock L to change. And as long as it is busy waiting, in other words, as long as it is running on the processor, we know that the value of the variable L cannot change, because the value of the variable L can only be changed by the process which currently holds the lock, and therefore in some sense, busy waiting is not a constructive activity. The alternative that is suggested in the lower half of the slide is that, an implementation of the lock in which the process which could have been busy waiting, in other words, it is trying to acquire the lock, rather than spending CPU cycles, executing test and set instruction, you could actually be moved by the operating system the process into a waiting state. And subsequently, when the lock becomes available, one of the waiting processes on the lock could be woken up.

(Refer Slide Time: 32:32)



More on Locks

Test&Set

Mutual Exclusion

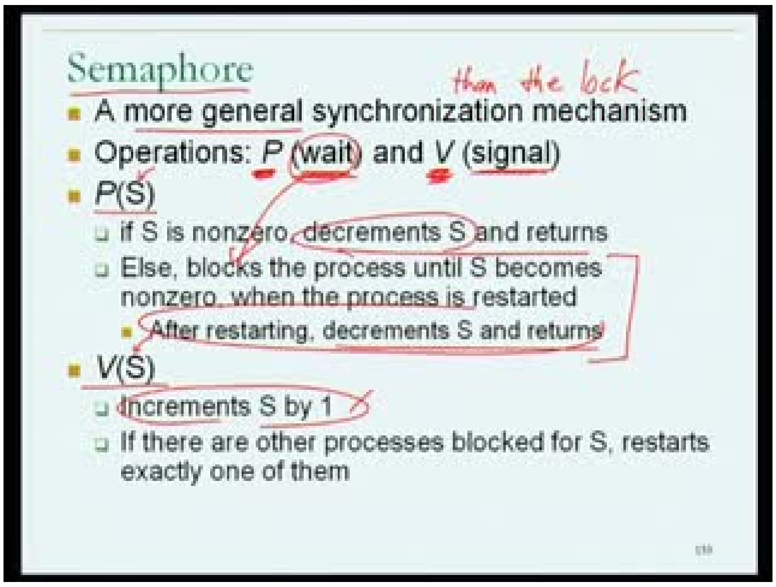
- Other names for this kind of lock
  - Mutex lock
  - Spin wait lock
  - Spinlock
  - Busy wait lock
- There are also locks where instead of busy waiting, an unsuccessful process gets blocked by the operating system
  - i.e., moved into the Waiting state until th[illegible]ck becomes available

The difference between these two should be clearly understood; both can be used to achieve mutual exclusion for critical sections, one requires operating system support, the other does not necessary require operating system support.

If you think about it little bit, in order to safely implement the busy wait lock, what we needed was, an instruction like the test and set instruction. In other words, in atomic read, modify, write instruction. And as long as the instruction set contains an atomic read modify write instruction like test and set, which is not a privilege instruction, then any user can write his or her own lock primitives or the lock primitives can be provided as library functions rather than a system calls.

Therefore, the upper option does not require operating system support, whereas lower option does, since it involves changing the state of a process from the perspective of the operating system; and in that sense, they are different perspective on how a lock could be implemented.

(Refer Slide Time: 37:04)



Semaphore

- A more general synchronization mechanism than the lock
- Operations: *P* (wait) and *V* (signal)
- *P*(S)
  - if S is nonzero, decrements S and returns
  - Else, blocks the process until S becomes nonzero, when the process is restarted
    - After restarting, decrements S and returns
- *V*(S)
  - Increments S by 1
  - If there are other processes blocked for S, restarts exactly one of them

But moving forward, there are other primitives, which might be provided to provide support for the various situations that may arise in writing concurrent programs. And one of the more widely known primitives is something known as the semaphore.

Some of you may of heard the term semaphore before outside of a computer science context, because semaphore is the name of some kind of a communication mechanism, that is used both between ships as well as I understand on airtime max, some kind of a flag based mechanism to signal - for individual to signal information to each other over distances, semaphore flags are used. So, the term semaphore comes from that origin. Once again it is referring to some kind of a communication mechanism or synchronization mechanism in the real world.

Now, I would like to talk about the semaphore a little bit, because it is an example of something that is more general than the simple lock that we have just seen. As I indicated, we will be looking at synchronization towards the end of this course, but I thought will be nice to give a few leaden ideas right now.

So, the semaphore is more general than the lock; and I will explain what I mean by more general shortly, when we talked about the lock, we found that they were two operations: there was acquire lock and there was release lock and we understand the implementation of acquire lock. In the case of the semaphore, there are two operations which are referring to as the P operation and the V operation. People sometimes refer to the P operation as the wait operation and the V operation as the signal operation.

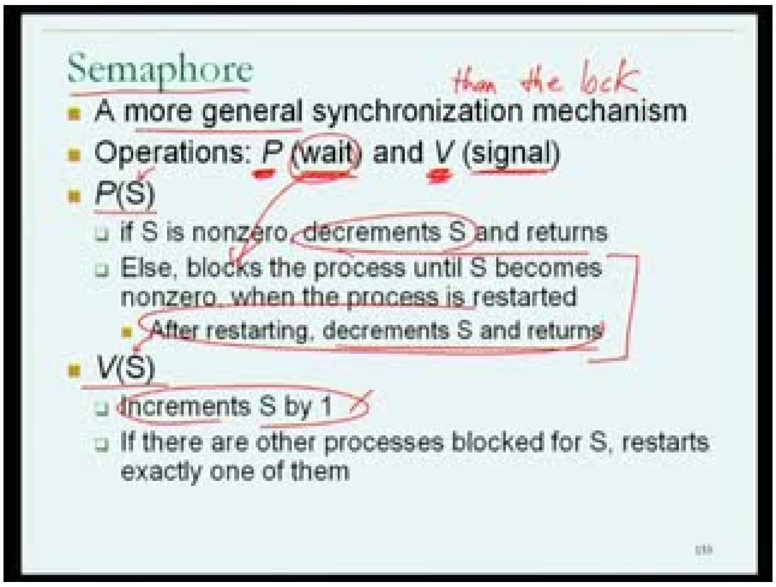
You may be wondering, why they call P and V. The reason is that the inventor of the idea of the semaphores in the context of inter process communication made up these ideas, he was not a English speaker, but a Dutch speaker; and Dutch word for wait starts with P, the Dutch word for signal starts with V. But in any event, the operations are fairly well known as P and V today, I will use that notation rather than writing them down as wait and signal.

Since the names are P and V, I do have to outline what their functionality is, the names do not give away, unlike release lock and acquire lock; the names do not give away any information about what these operations do.

So I am going to describe the functionality in terms of, what happens when the P operation is applied to a semaphore S, what happens when the V operation is apply to a semaphore S. So, S refers to a semaphore; at this point, we view a semaphore V being some kind of data structure.

Now, in the case of the P operation, we find the first thing that is done is, it is checked to see whether S has a nonzero value associated with it. And if that is the case, then the value of S is decremented by 1 and control is transferred; in other words, return from the P operation from the happens.

(Refer Slide Time: 37:04)


Semaphore

than the lock

- A more general synchronization mechanism
- Operations: P (wait) and V (signal)
- P(S)
  - if S is nonzero, decrements S and returns
  - Else, blocks the process until S becomes nonzero, when the process is restarted
    - After restarting, decrements S and returns
- V(S)
  - Increments S by 1
  - If there are other processes blocked for S, restarts exactly one of them

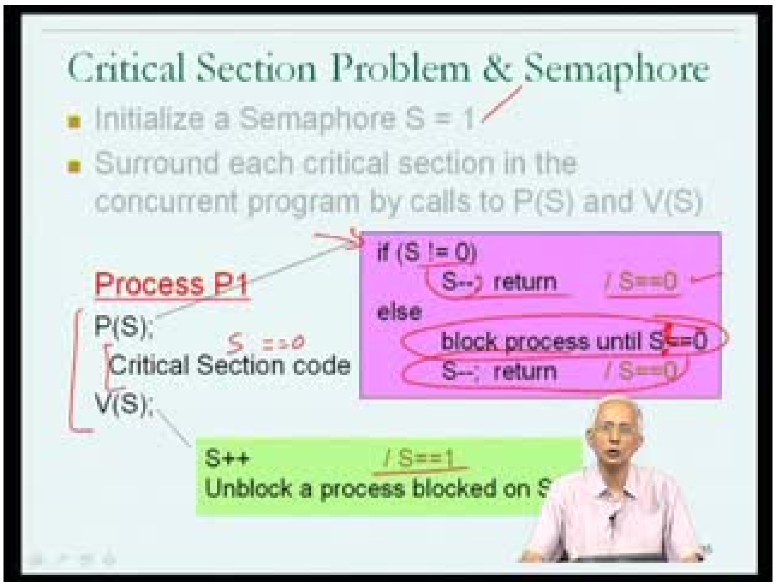
So, if S has a nonzero integer value associated with it, then S is decremented by 1 and return from P of S. What if S is not nonzero? Then what happens is the process which executed as P of S gets blocked. In fact, it gets blocked until S becomes nonzero; and when that happens, when the process is restarted, what it does is, decrements S and written.

So essentially, now understand wait operation does is, when a process calls P of S, then if the value associated with the semaphore is nonzero, then the value gets decremented and the process returns; the process continues with successful return from the P of S call. On the other hand, if the value associated with S is 0, then the process gets blocked, until the value of S becomes nonzero; at which point in time, the process resumes execution and starts by decrementing as and returning. The same thing that you would have done, if you had found that S was nonzero.

What about the V operation? So, we now we understand why this is called wait, because there is a blocking, until some condition is satisfied; the desired condition is B being S being equal to 0.

Now, what happens when the V or signal an operation is executed on S? Basically, the value associated with semaphore is incremented by 1; the opposite of what was happening in the case of P, and subsequently, if there are any processes which were blocked waiting for S, such as we saw could happen, then one of them is restarted. And when it is restarted, it proceeds as we see. So, in some sense, we understand why it is called signal, because it is an operation which can be used to signal that a process has in some sense, finish with the use of the semaphore and that incrementing the semaphore is a flag of having finished with the use of the semaphore; I used that description in the light of our discussion about the lock. In the more general sense, just thinking of them as wait and signal may make sense let us see.

(Refer Slide Time: 42:25)



Critical Section Problem & Semaphore

- Initialize a Semaphore S = 1
- Surround each critical section in the concurrent program by calls to P(S) and V(S)

Process P1

```
P(S);
  Critical Section code
V(S);
```

```
if (S != 0)
    S--; return        / S==0
else
    block process until S!=0
    S--; return        / S==0
```

```
S++                    / S==1
Unblock a process blocked on S
```

Now, let us just try to understand the semaphore a little clearer by using the semaphore to implement in a sense of solution to the critical section problem. Until now, we had used the lock to solve the critical section problem. The critical section problem was we want to ensure that, processes executing in their critical sections mutually exclusively; in other words, only one at a time.

Now, the where that I am going to suggest the setting up is, I am going to assume that the semaphore is initialized to 1. Note that what happens in P is that the semaphore gets decremented, what happens in V is that semaphore getting is incremented, therefore, I am going to initialize the semaphore to 1, in other words, I am initializing the semaphore to a nonzero value.

Now, what I will do now is, I understand that a process can bring the value of the semaphore to 0 by doing a P of S and that it can bring the value of the semaphore back to 1 by doing a V of S. Therefore, the suspension is that I can achieve mutual exclusion by surrounding each critical section by a call to P of S, before the critical section, and to V of S, after the critical section, if I assumed that I had initialized the semaphore to 1. So, that is what I will assume, that whenever I have a critical section in my concurrent program, it is preceded by a call to P of S and it is succeeded by a call to V of S.

Now, let us see how this works. Now, let suppose that I have a process P1 which is executing this code. What happens when it executes P of S? This was the essentially what happens in, this is an expansion of my description of P of S. Remember, if S is nonzero, it decrements as and returns, in other words, it makes S equal to 0; why do I add this comment? Just note that, if S was initially 1, then by decrementing S, I making S equal to 0.

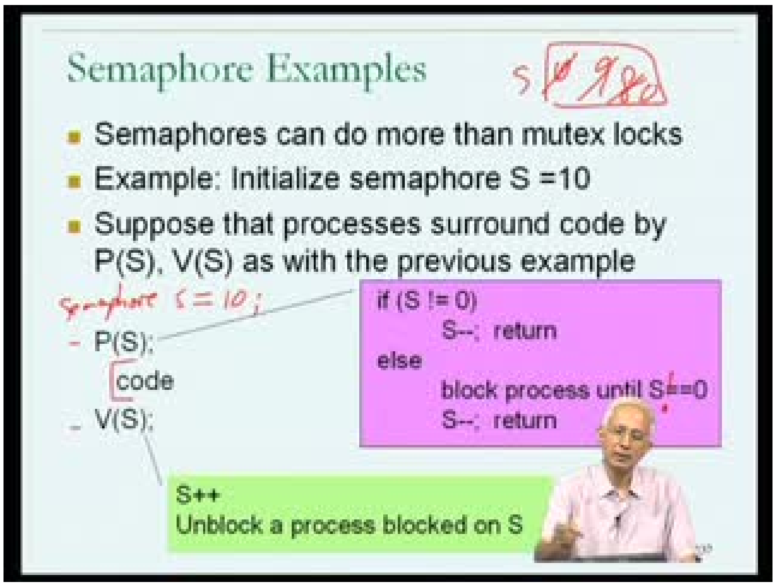
What if S was equal to 0? Then, the process gets blocked, until S becomes equal to 0; and at that point, it decrement S from 1 and returns; at the block process until S is not equal to 0. Just looking back at the previous slide, just note that the process is blocked, until S becomes not 0. So, these thus have to be corrected.

Now, we understand that this, what happens when P of S is executed. What happens V of S is executed? Semaphore gets incremented, so it was equal to 0; will now get incremented, which means that it becomes equal to 1 and one of the process which was blocked waiting on the semaphore gets unblocked.

So, this is what happens as far as the process is concerned. Therefore, while the process is executing in the critical section, we understand the value of S is going to be equal to 0. And subsequently, any other process which tries to acquire the semaphore by using the same series of events, for example, if a process P2 tries to execute the same piece of code, it will find that it gets blocked, until the value of S becomes nonzero; in other words, until process P1 increments the semaphore. Therefore, very clearly the semaphore can be used to do what the Mutex lock did; kindly note the correction to the slide, this should be not equal to 0.

Now I had mentioned in my introduction to semaphores, that the semaphore is more general than the Mutex lock. Suggesting that the semaphore can not only do everything that the Mutex lock or the busy wait lock can do, we can do other things as well.

(Refer Slide Time: 46:14)



Semaphore Examples

S ~~1~~ ~~9~~ ~~8~~ 0

- Semaphores can do more than mutex locks
- Example: Initialize semaphore S =10
- Suppose that processes surround code by P(S), V(S) as with the previous example

semaphore S = 10;

```
- P(S);
    code
- V(S);
```

```
if (S != 0)
        S--;  return
else
        block process until S!=0
        S--;  return
```

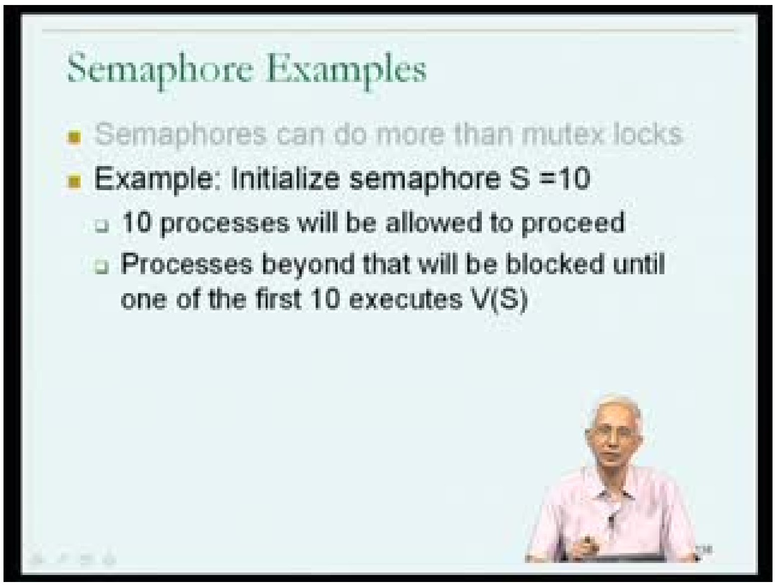
```
S++
Unblock a process blocked on S
```

So, let me just try to show you, some examples of additional things that the semaphore can be used for. Now, let us just think of a simple possibility. In the example, which we have seen using the semaphore, I talked about initializing the semaphore to 1, because I wanted to ensure that at most one process could be in its critical section at a time.

What if I initialize the semaphore to 10? What would happen then? Now, once again I will just assume that I am going to use this semaphore, in such a way that I have concurrent program with many processes.

And that each of the processes will have some code; I would not prefer to it as a critical section code, I will just refer to it has some code, which it might have P of call to P of S, before that, and call to V of S, after that. So, this particular semaphore may have been declared as semaphore S and may have been initialized to 10; in the previous example, the semaphore had been initialized to 1.

(Refer Slide Time: 47:57)

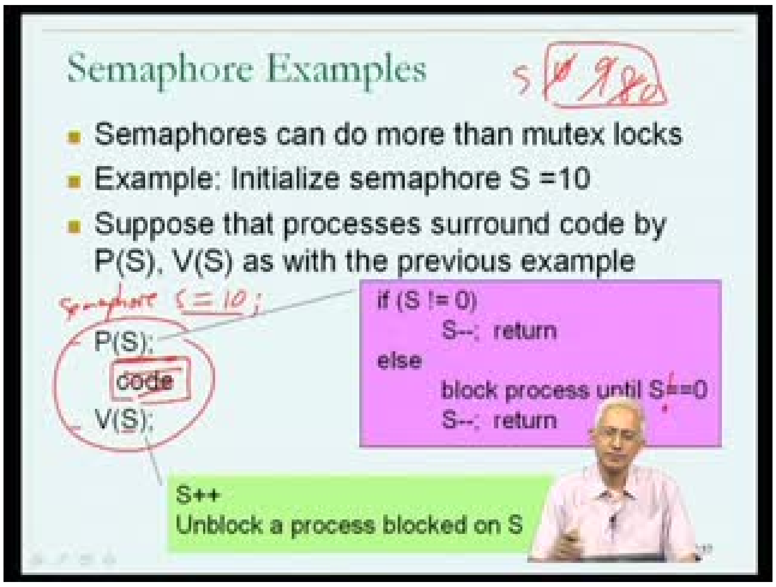


Semaphore Examples

- Semaphores can do more than mutex locks
- Example: Initialize semaphore S =10
  - 10 processes will be allowed to proceed
  - Processes beyond that will be blocked until one of the first 10 executes V(S)

Now, what will happen if I have a semaphore that is initialize to 10? Once again, the same piece of code gets executed. Remember, the definition of P of S and V of S is fixed – it is well defined. But, now, when the first process executes P of S, it finds that S is not 0 and therefore decrements S. In other words, the value of S goes down from 10 to 9 and the process returns. What happens when the second process executes P of S? It also finds that S is not 0, therefore it decrements S goes on to 8 and enters the piece of code, and this keeps on happening, until 10 processes have successfully decremented as at which point value S will be 0. And when the eleventh process enters a picture and tries to decrement S, before it tries to decrement S, it finds that the value of S has become 0.

(Refer Slide Time: 48:12)



Semaphore Examples

- Semaphores can do more than mutex locks
- Example: Initialize semaphore S =10
- Suppose that processes surround code by P(S), V(S) as with the previous example

```
P(S);
    code
V(S);
```

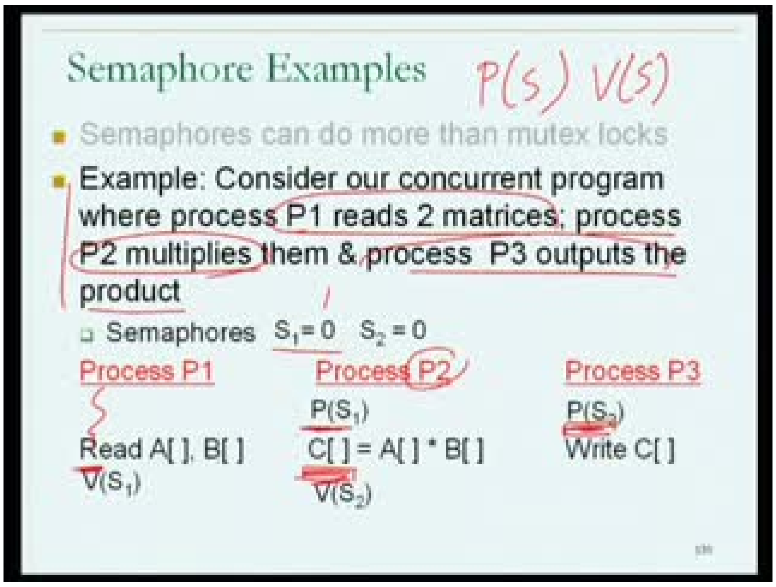
```
if (S != 0)
        S--; return
else
        block process until S!=0
        S--; return
```

```
S++
Unblock a process blocked on S
```

S ~~10~~ 9 8 0
semaphore S = 10;

And therefore, this use of the semaphore; note that I have used the semaphore an exactly the same way as it was used for the critical section. Only the difference in the program is that, the semaphore is initialized to 10. Now, the net effect was, this is a some kind of facility by which I can allow up to 10 or processes to execute in a particular piece of code or up to 10 processes to proceed beyond the call to P of S, do whatever they are doing; and eleventh process and beyond will be blocked, until at least 1 of the 10 had finished and exited from that region of the code.

(Refer Slide Time: 48:45)



Semaphore Examples

- Semaphores can do more than mutex locks
- Example: Consider our concurrent program where process P1 reads 2 matrices; process P2 multiplies them & process P3 outputs the product
  - Semaphores $S_1 = 0$ $S_2 = 0$

Process P1	Process P2	Process P3
	$P(S_1)$	$P(S_2)$
Read A[ ], B[ ]	C[ ] = A[ ] * B[ ]	Write C[ ]
$V(S_1)$	$V(S_2)$	

So this is definitely something that is different from one the lock could be made to do. But that is not a very satisfying example, because it is not clear why I would want to do, why I have an actually given your satisfying example of situation where I would want 10 processes is to be doing one thing or the other.

So, I will actually go back to an example which I have talked about, in order to show that the semaphore can do something more than what a lock can do. This example is an example I had, when I first talked about concurrent programs, I talked about this scenario. I could have a situation where I want to multiply two matrices - large matrices let us say - and I choose to do it as a three process program: one process reads the matrices and another process multiplies the matrices and the third process,prints out or outputs the product matrix.

And we saw that there was a synchronization requirement here. I did not want process P2 to start doing the multiplication, until process P1 had finished doing the reading; further, I did not want process P3 to start doing the output, until process P2 had finished doing the multiplication.

So, there was the need to synchronize or hold back processes, until another process had achieved some of its work. And we can do this using semaphore quite easily; so, let me just illustrate how.

So, here we have three processes and they are assigned work is listed over here. Process P1 is supposed to read in the matrix; the matrices that are to be multiplied, I refer to them as a matrix A and the matrix B. Process P2 is supposed to multiply the two matrices and produce a product matrix, which is supposed to be what is written out by process P3. And I will just remind you that in this example, I am assuming that the matrices themselves are possibly available in shared memory. So, we are not concerned about the fact that they are shared variables, where trying to understand something regarding the synchronization of the processes.

Now, the requirement that I have is process P2 should not preceed beyond this point in its program, until process P1 has reached this point in its program; that is, the first synchronization requirement. In other words, it is only after process P1 has finished reading, the process P2 should start multiplying. So, there is one synchronization that we have to do over here.

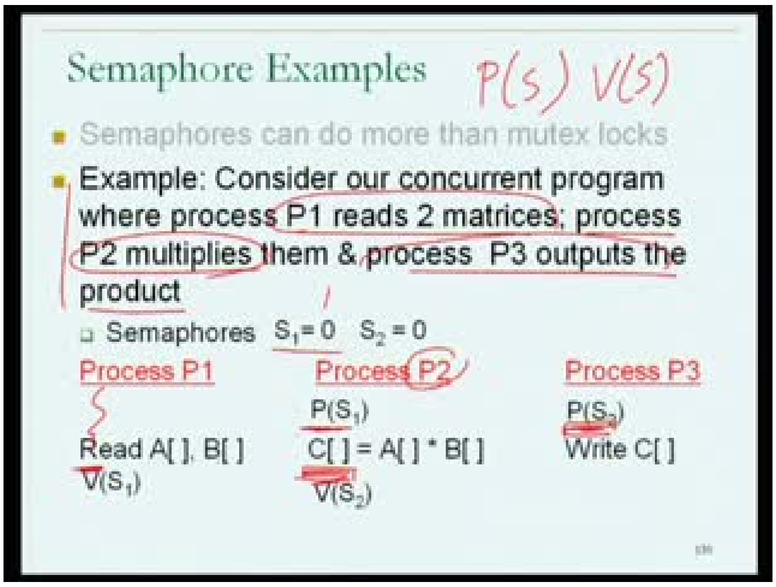
Now, how can we achieve these using semaphores? Now, the way to look at this is the following; now, the process which I want to cause to wait is the process P2. Specifically, I want processes P2 to wait at the red line, until process P1 has reached its red line. In other words, process P1 has passed the read A B, part of its program. So, I want process P1 to wait and using semaphores the operation which I have, that can be used to make a process wait is the P operation. The V operation cannot be used for waiting, the V operation merely updates the semaphore.

Therefore, I know that I need to include a P operation on a semaphore, therefore I understand that I need to declare 1 semaphore, which I will call S1 and that prior to doing the multiplication, process P2 will have to weight on that semaphore.

So, I include a call to P on S of 1 over here. Now, if process P does a wait on the semaphore, unless the value of the semaphore is 0, process P1 will successfully finished with the call to P. Therefore, it is necessary for me to initialize the semaphore to the value 0, if I am using this particular semaphore for process P1 to wait. Now, that if I initialize it to 1, then process P1, when I call P of S of 1 would decrement the semaphore and proceed to multiply the matrices prematurely, therefore it is important that I initialize that semaphore to 0.

So, what will happen? What will happen is the process P1 will be blocked inside the semaphore, until the value of the semaphore changes. How can the value of the semaphore change? The value of the semaphore should be causes to change, as soon as process P1 has finished reading the matrices. And I can do that using the V call to the semaphore, which will cause the value associated with the semaphore to be incremented.

(Refer Slide Time: 48:45)



Semaphore Examples

P(S) V(S)

- Semaphores can do more than mutex locks
- Example: Consider our concurrent program where process P1 reads 2 matrices; process P2 multiplies them & process P3 outputs the product
  - Semaphores $S_1 = 0$ $S_2 = 0$

```
Process P1        Process P2          Process P3
                  P(S1)               P(S2)
Read A[ ], B[ ]   C[ ] = A[ ] * B[ ]  Write C[ ]
V(S1)             V(S2)
```

Therefore, I include a call to V on S of 1 at the appropriate point in process P1 and by doing this, I get the required synchronization. Process P1 we will finish the reading and we will then increment to semaphore. The semaphore had been initializing to 0, therefore the incrementing of the semaphore will causes it go to 1. And any process which had been blocked on the semaphore, and such as possibly process P1, would then get unblocked and proceed to the next step. I can use exactly the same kind of use of semaphore for the other synchronization requirement that I have, which is the process P3 should not proceed beyond this point in its program, until process P2 has finish multiplying the matrices.

Now, can I use this semaphore S1 for this purpose? It is possible that I could, but to be absolutely sure that I am writing this program correctly, I will use another semaphore, synchronization between process P2 and process P3. The use will be exactly the same, but I will refer to it as the semaphore S2.

So, with this proves that the semaphore is more general than the lock. We know that, there are different mechanisms that an operating system provides to support concurrent programming. And I will end today's lecture with just a brief summary on where we are at this point in time. We understand that, in order to do concurrent programming in situations, where there are sheared variables, they may be a need to protect the regions of the programs, which modified the shared variables with mutual exclusion like requirement or to synchronize the processes in situations, where one may have to wait until one of the others has done a certain amount of the work of the common objective.

And typically, synchronization a mutual exclusion primitive, such as locks or semaphores will be provided. Either in the form of library functions, which could be the case for certain kinds of locks, I could busy wait lock or in the form of operating system functionality, which may be the case we necessary in cases, where the blocking of a process has to take place such as in a blocking lock or in a semaphore. By the definition of semaphores is as in our lecture today, the P operation and the V operation on semaphores may have to block or unblock processes, which require operating system intervention. And we will look at a little bit more about concurrent programming in the next lecture.

Thank you.