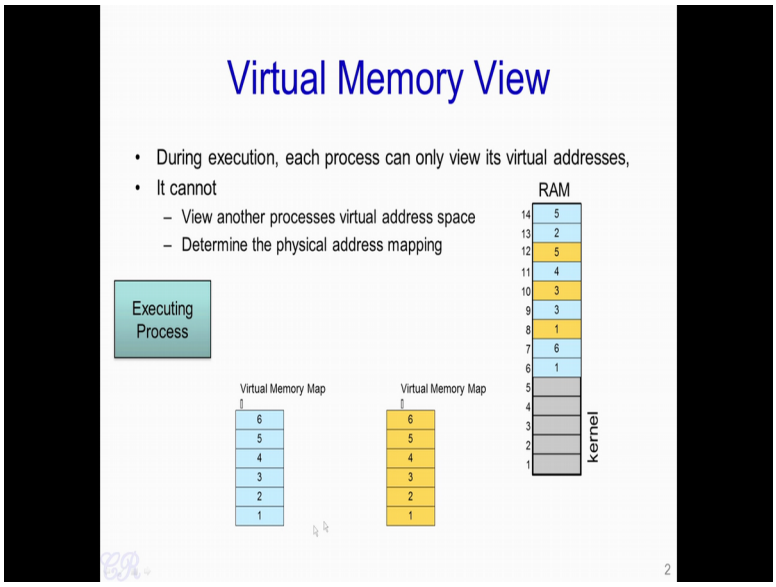
Introduction to Operating Systems
Prof. Chester Rebeiro
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Week – 06
Lecture – 23
Inter Process Communication

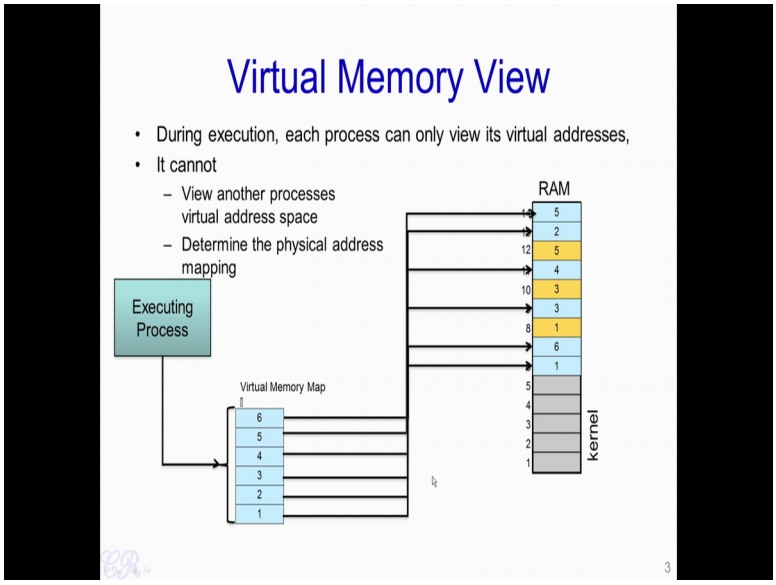
In this video we will look at Inter Process Communication. Essentially, when we write large applications it is often quite useful to write them as separate processes. So in order to have an efficient communication between these processes, in order to make things happen efficiently we use inter process communication. In this particular video, we will look at a brief introduction to IPC's that is Inter Process Communication and we will see a few examples of the same.

(Refer Slide Time: 00:50)



So we had seen this Virtual Memory View of a process. We said when a process is executing it sends out virtual addresses which get mapped into this virtual memory map. And we had seen that there is an MMU which uses page table stored in the memory and converts these virtual addresses into physical addresses. Now we have also seen that each process that executes in the CPU will have it is own virtual memory map.

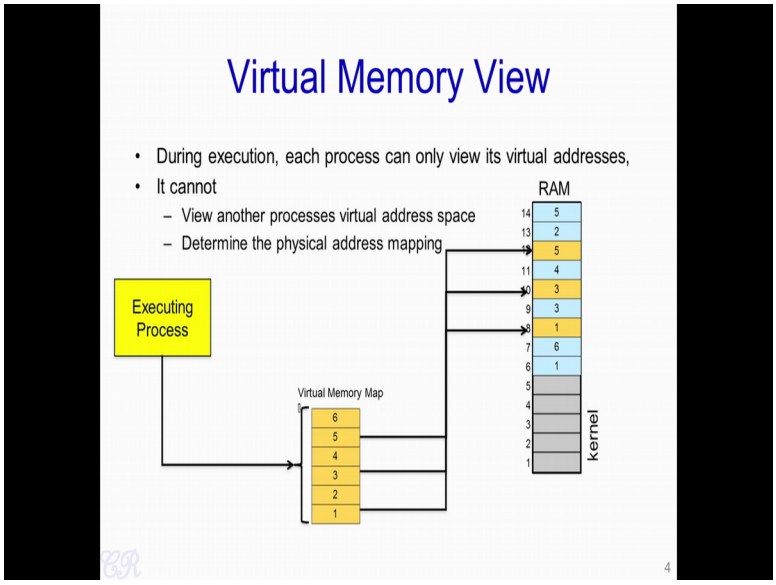
(Refer Slide Time: 01:23)



The use of virtual memory map provides some level of abstraction. Essentially, the executing process would only know its virtual addresses and with that virtual addresses, it can only access the user space of the virtual memory map. We had seen that the MMU then transfers this mapping into corresponding physical addresses. So, the executing process will have no way to determine what the corresponding physical address for its memory accesses are. So for instance, if you declare an integer 'i' in your program you can print out the virtual address corresponding to 'i' for example, with a printf &i. However, it is not possible for you in the user space to determine what the corresponding physical address is for that particular variable 'i'.

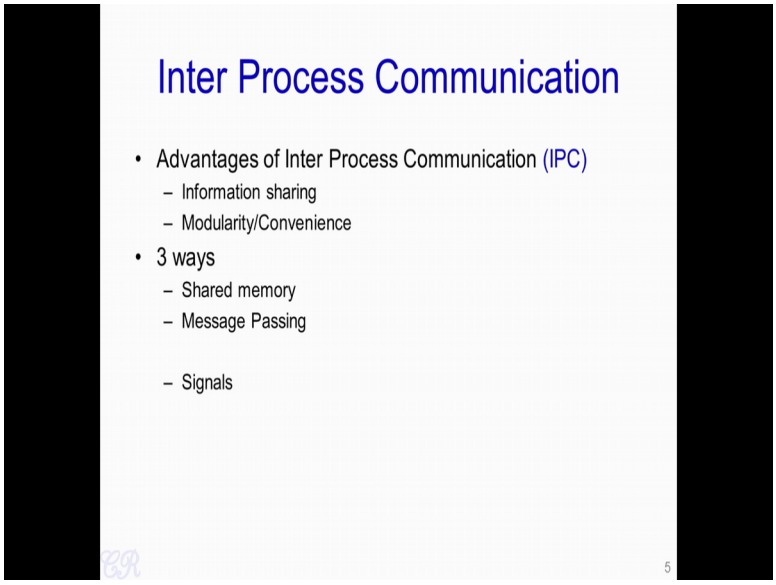
The second thing what the virtual memory map provides is that the executing process could only have access to its own virtual memory map. If there is another process it has with its own table, now in user space there is no way for the executing process to determine anything of the other process.

(Refer Slide Time: 02:42)



So, when there is a context switch and a new process executes it is the second processes virtual address map or virtual memory map which is then used. So this process has no way to determine any other processes virtual address map. So the thing to conclude from these 2 slides (Virtual memory view mentioned above) is that, a process has no way to determine any information or any data about another process. So given this, how does one process communicate with another process?

(Refer Slide Time: 03:15)

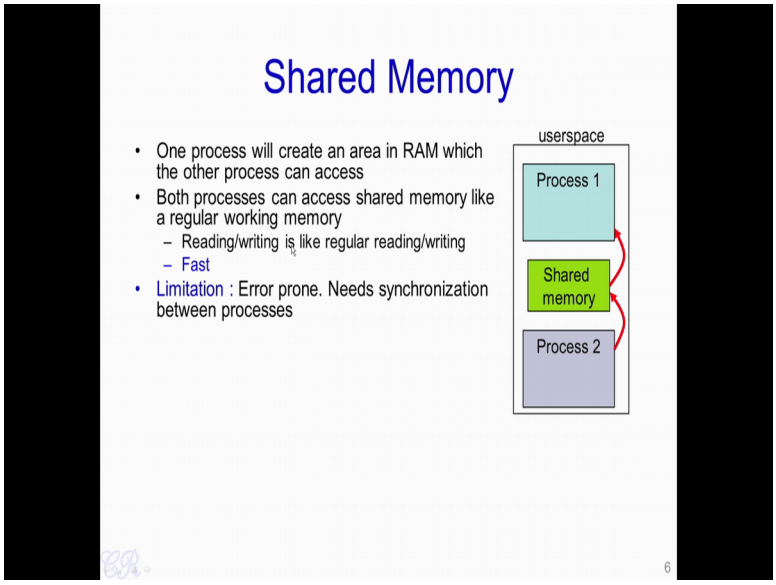


So in order to do this, there is a mechanism known as Inter Process Communication. Essentially with IPC's or inter process communication, two processes will be able to send and receive data between themselves. The advantage of IPC is that the processes could be return to be modular. So essentially each process is meant to do a single job and processes could then communicate with each other through IPC's.

For an example, let us say we have a data acquisition system and a control system. So essentially we could write one process which acquires data from the external world, such as the temperature, pressure, or the speed, and then it could then send this information or the data collected to a second process which then analyzes the data and determine some particular parameters. So these parameters could be sent to a third process which then actuates some external data, for instance it could open a valve or close a valve or adjust the temperature of the room and so on. Thus, we see we are able to achieve a modular structure in our application.

Each processes job is to only focus on a single aspect. The communication between the processes is achieved by inter process communication. In typical operating systems there are 3 common ways in which IPC's are implemented. These are through Shared memory, Message passing and Signals. So let us look at each of these things.

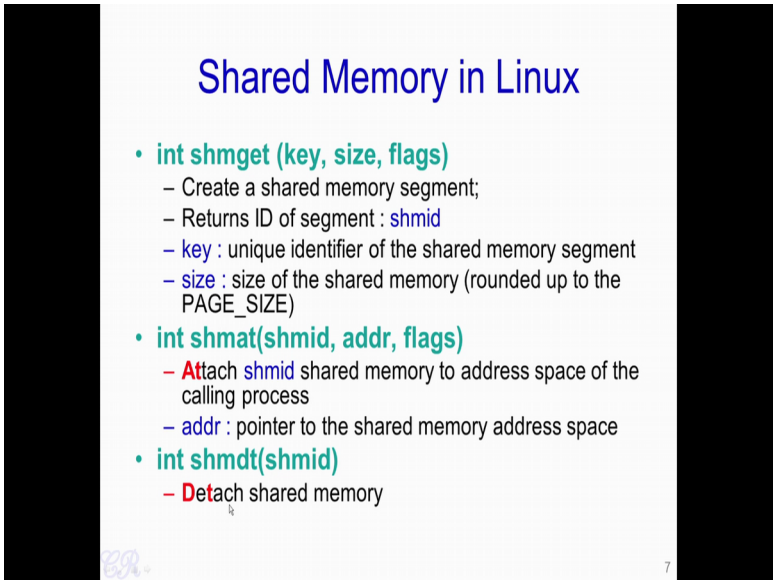
(Refer Slide Time: 05:00)



So, with a shared memory we have one process which creates an area in RAM which is then used by another process. So essentially, the communication between process 1 and process 2 is happening through this particular shared memory (green colored box mentioned in above slide). Both processes can access the shared memory like a regular working memory, so they could either read or write to this particular shared memory independent of each other. The advantage with this is that the communication is extremely fast, there are no system calls which are involved. And, the only requirement is that you could define an array over here (in shared memory) and then fill the array in the shared memory which can then be read by the other process.

The limitation of the shared memory approach is that it is highly prone to error; it requires the two processes to be synchronized. So, we will take a small example of how shared memory is implemented in Linux.

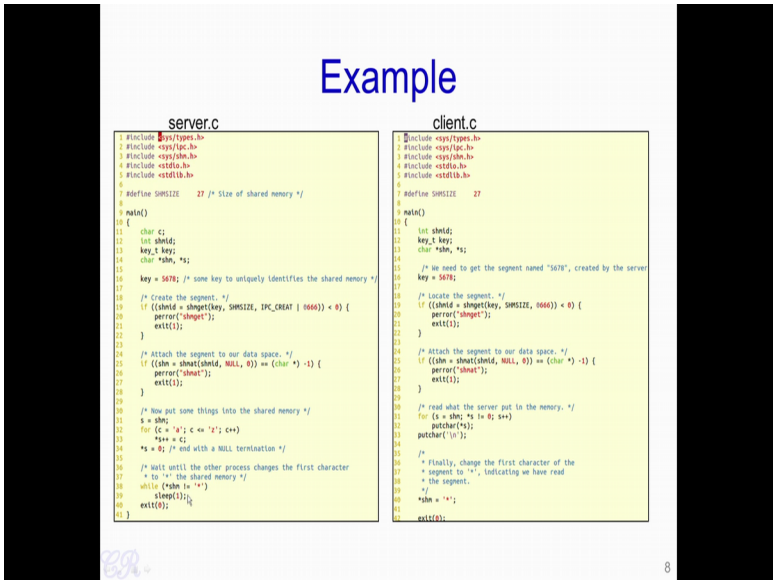
(Refer Slide Time: 06:05)



So essentially, in a shared memory in the Linux operating system, we have three system calls which are used. First is the 'shmget' or the shared memory get, which takes three parameters; a key, size, and flags. So this system call creates a shared memory segment. It returns an ID of the segment that is 'shmid' - the shared memory ID of the segment. So the parameters key is a unique identifier for that shared memory segment, while size is the size of the shared memory. So this is typically rounded up to the page size that is 4 kilobytes. So, this is how a shared memory gets created in a particular process (mentioned in above slide).

Now an other process could attach to this shared memory by this (mentioned in above slide) particular system call that is shared memory attach - 'shmat'. So, this particular call requires the shared memory ID (shmid), and address (addr), and flags. So, essentially system call would attach the shared memory to the address space of the calling process. The address is a pointer to the shared memory address space. So we will understand more of this through an example. The opposite of the shared memory attach is the shared memory detach where a process could detach the shared memory from it is user space. So, the detach system call takes the shared memory ID (shmid).

(Refer Slide Time: 07:40)



Let us see the, an example of shared memory with this example (refer above image). So, let us say we have written this particular program called server.c and we create a shared memory in this. So we define a key called 5678 this is some arbitrary value for key (defined in above program), but the requirement is that this key should uniquely identify the shared memory. So we could use this key to invoke this function shmget (mentioned in if condition in above program), we pass it the key, we pass it SHMSIZE that is is defined here (before main() in above program) as the size of the shared memory which we want to create. Note that; this size although we specified as only 27 bytes, we will get extended to a page that is we will create a page of 4 kilobytes corresponding to this.

The third parameter is the permissions, that is we are given it as IPC_CREAT that is we are creating this particular shared memory and these are the permissions (0666): read, write, execute permissions. And of course, if this function (shmget) fails then it enters over here (inside if condition) and exits. Otherwise, if it executes successfully we get a valid shared memory ID.

The next part is to invoke the shared memory attach - shmat (second if condition in above program) providing this particular ID (shmid) and we will get a pointer to this particular shared memory. So shm, so we have like char *shm this is a pointer to the

shared memory. Now the shared memory attach would return a pointer to this particular shared memory.

Now in case it (second if condition) returns -1 then it is due to an error and we are exiting. Now at this particular point we have obtained a pointer to the shared memory and we can use that particular pointer just as a standard pointer as we use in a C program. So for instance, over here (s = shm) we have moved the pointer to this variable s which is defined as char *s and we have put alphabets from - a to z in this thing (inside for condition). And finally, we have ended it with a null termination and then we are going in a loop continuously sleeping (while condition in above program), so we will come back to these two statements (mentioned inside while loop condition) later on.

Now, let us look at the client side of this code (client.c program mentioned in above slide). So we invoke this shmget (function in first if condition) and give it a key, that is key is the same 5678, the shared memory size (SHMSIZE) is as before 27 and we are providing the permissions (0666). Note that we do not require to give CREAT over here (IPC_CREATE given in server.c program but not in client side) because the shared memory region is already created. Then we attach to this particular shared memory segment (shmat function in second if condition) as before (as in server.c program), we invoke the function shared memory attach - 'shmat' and we pass it the shared memory ID (shmid). Then we get a pointer 'shm' which is a pointer (declared as char *shm) to this shared memory location.

So, essentially both the server as well as the client are pointing to the same physical memory page. Now we could actually read data from this particular shared memory region (for condition mentioned in client.c program). So remember that the server has put values from a to z that is a b c d to z (inside server,c program), while in this case (client side program) we could read the values a to z into this particular pointer s and print it out in the screen. So once we have completed reading all data from the shared memory we put this star into the shared memory location (*shm = '*';). This is the first location in the shared memory.

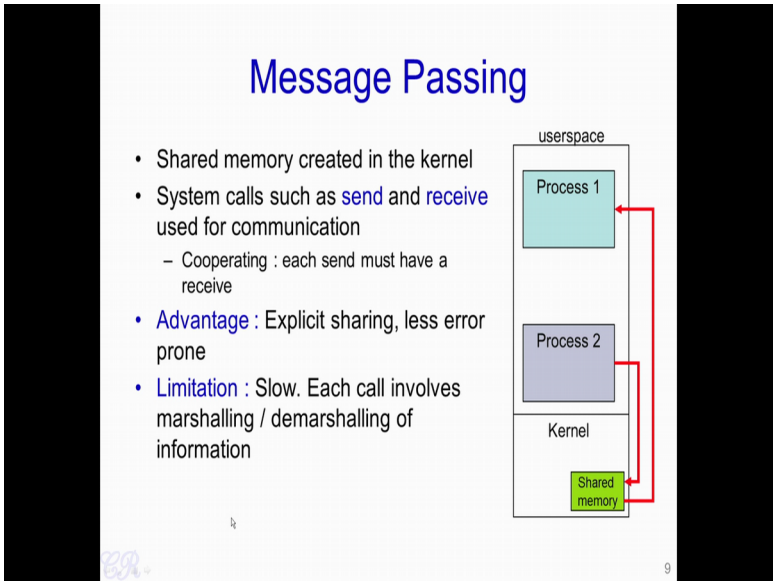
Now recall that over here (while loop mentioned in server.c program) in the server we

have put a while loop which continuously loops until the first parameter pointed to in the shared memory is a *. So when we obtain a star (from the client side program) it means that the client has completed reading all data and therefore the server can exit. In a similar way the client can exit as well. So, you see that we have these 2 processes; one is server, one a client.

And these two processes are having their own virtual address space, but by the use of this shared memory (shm) which is created by the shared memory get (shmget) and shared memory attach (shmat), we have able to create a shared memory region which is common between the server and the client. Then we have obtained a pointer to that particular shared memory region that is s on shm (s = shm) and the server could put data into the shared memory region while the client could read data, the vice verse is also possible.

And at the end of it we required to note that there is a synchronization required between the server and client. So this synchronization requirement should be explicitly program into this particular server client modules.

(Refer Slide Time: 12:54)

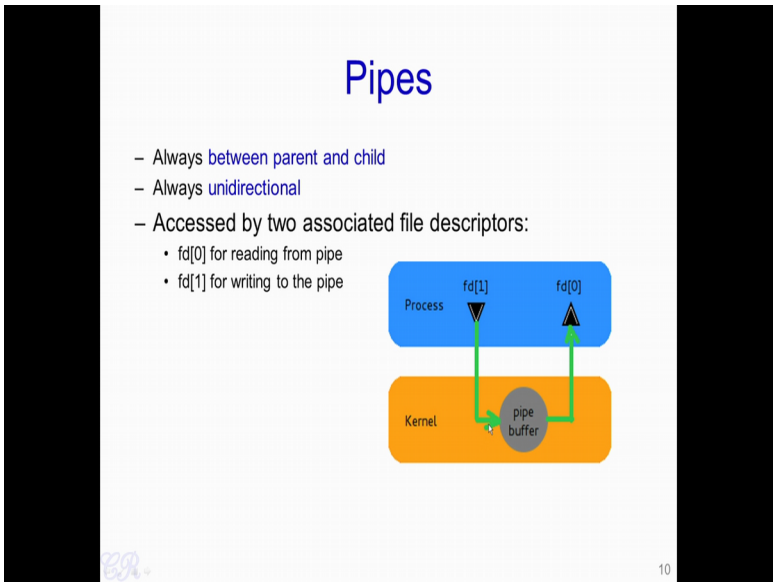


Next, we will look at the Message Passing. Unlike the shared memory that we just seen

where the shared memory is created as part of the user space, in message passing the shared memory is created in the kernel (refer above slide). Essentially we would then require system call such as send and receive in order to communicate between the two processes. If a process 2 wants to send data to process 1, it will invoke the send system call. So this would then cause the kernel to execute and it would result in the data return into the shared memory. While, when process 1 invokes receive, data from the shared memory would be read by process 1.

The advantage of this particular message passing is that the sharing is explicit. Essentially both process 1 and process 2 would require support for the kernel to transfer data between each other. The limitations is that it is slow, each call for the send or receive involves the marshalling or demarshalling of information. And as we know in general a system call has significant overheads. Therefore, message passing is quite slow compared to shared memory. However, it is less error prone then shared memory because the kernel manages the sharing, and therefore would be able to do the synchronization between the process 1 and process 2.

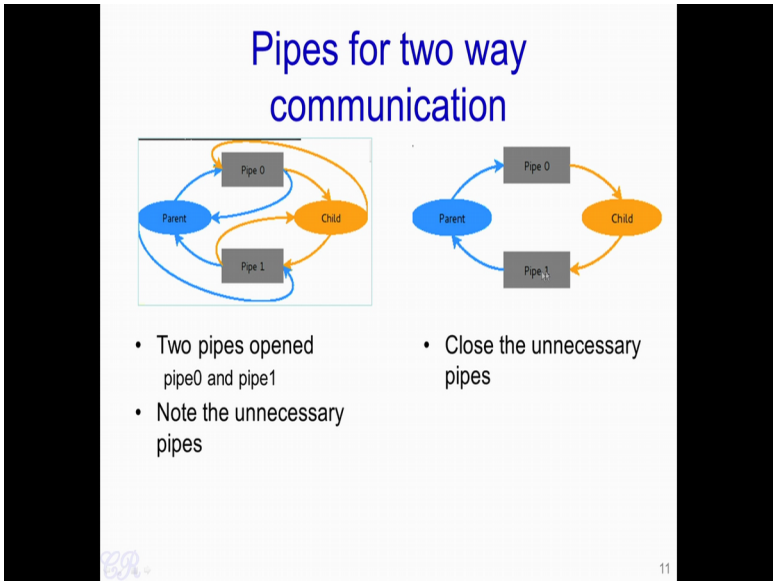
(Refer Slide Time: 14:27)



Another very common application or message passing is the use of Pipes. Now pipes are used between parent and child processes only. Essentially, you can only communicate

data from a parent process to a child process and vice verse. Another aspect of pipes is that it is unidirectional. Now when a pipe is created generally there are two file descriptors which are associated with it that is fd [0] which is used to read from the pipe, while fd [1] is used to write to the pipe. So we know the unidirectional nature of this particular IPC (refer above slide). So, fd [1] is exclusively used to write to the pipe buffer, while fd [0] is used to read from the pipe buffer.

(Refer Slide Time: 15:14)

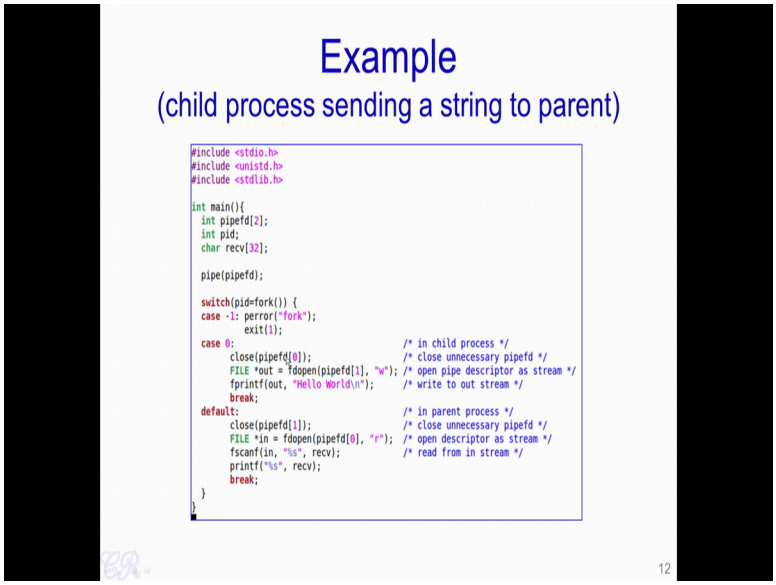


So in order to obtain two way communication between the parent and the child, we would require two pipes to be obtained; pipe 0 and pipe 1 (refer above slide image). So, when pipe 0 is opened it would create two file descriptors; one to write into pipe 0 and the other one to read from pipe 0. Similarly, there are two file descriptors to write to pipe 1 and read from pipe 1. Similarly there are two pair of file descriptors for pipes in the child process.

Now you note that these additional descriptors are not required, therefore we could actually close the extra file descriptors to obtain something like this (second figure in above slide image). Now in order that the parent sends some information to the child, the parent will write to pipe 0 while the child will read from pipe 0. In order to that the child sends some information to the parent, the child will write to pipe 1 while the parent will

read from the pipe 1.

(Refer Slide Time: 16:23)



So let us see an example of the user pipes (refer above slide). So this again is a standard Linux example and as we have said, so we create a parent and child process so that there is a uni directional communication from the parent to the child. So let us look at this particular thing (program mentioned in above slide). First we invoke this particular system call called pipe and give it this pipefd (i.e pipe(pipefd);). So pipefd is defined over here (variable below main()) and taking two elements and then we invoke the system called fork (mentioned inside switch condition). So as we have studied the fork system call would create a child process which is the exact duplicate of the parent process. So in the parent process the value of a pid that is the value what fork returns is the child's pid value, and it is a value which is greater than 0. On the other hand in the child process the value return by fork is 0.

So, let us see what happens in the child process first. So in the child process because the pid value is 0 so it would come over here (case 0) and the first thing what we do is we close pipefd [0]. So, closing pipefd [0] would mean that we are closing the read pipe to the child, that is so we are we having like two pipes over here (first figure in slide time 15:14) corresponding to pipe 0 and we have just closing the read pipe while the pipe 1

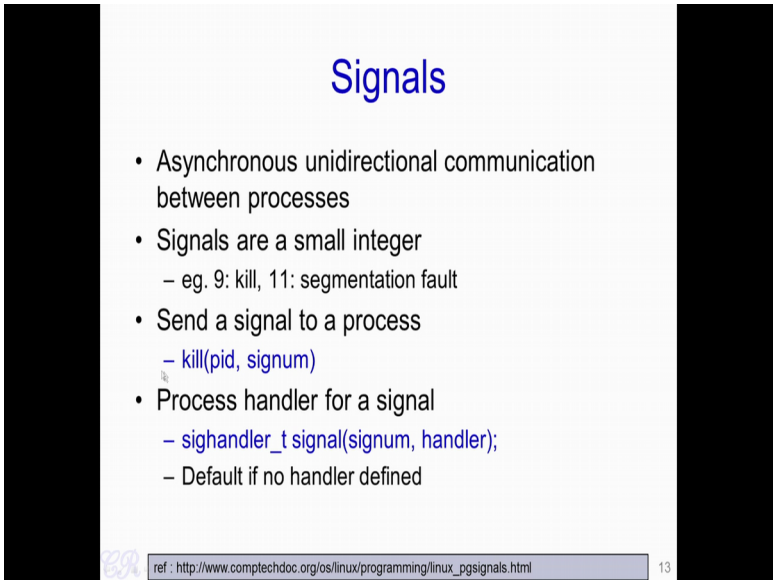
that is the right file descriptor is still open. Now let us see what happens in the child process, in the child process the value of pid returned by fork is 0, therefore it will result in this particular code being executed (case 0 in above program).

So in this code (case 0 set of statements) first, there is we are closing the pipefd [0]. So this means that the file descriptor corresponding to the read pipe is closed. Second, we are then opening a second file called fdopen corresponding to file descriptor 1 and it is opened in the write mode ("w") and we could then use fprintf(out, "Hello World\n");. So essentially what we are doing is to this pipe with descriptor 1 we are writing "hello world". If you go back to this particular thing (refer slide 15:14 second figure), so what we are seeing is that we are writing "hello world" into this pipe 1.

Now in the parent process which would execute over here (default set of statements) because fork would return with the value which is greater than 0, so we are closing the write pipe and opening using fdopen, the read pipe (refer default section in program). So we are then using fscanf to read whatever has been pushed into the pipe, in this case it is "hello world" and printing it to the screen.

So, essentially what we have implemented in the program is this lower part. So the child opens this pipe in the write mode and puts "hello world" into this pipe while the parent then opens the pipe 1 in the read mode and reads the string "hello world" from the pipe. Thus, the string "hello world" has been transferred from the child to the parent. So you can actually try to implement this particular program and execute in a Linux system.

(Refer Slide Time: 19:58)



Besides, the message passing, shared memory and pipes that we have studied so far. A Third way of inter process communication is by what is known as Signals. So, signals are asynchronous unidirectional communication between processes. The operating system defines some predefined signals and these signals could be sent from the operating system to a process, or between one process to another. Signals are essentially small integers, for instance an each of these (refer above slide second point) integers has a predefined meaning. For instance, 9 would mean to kill a process, while 11 would mean a segmentation fault and so on.

So in order to send a signal to the process we could for instance have kill send it to this pid and we could send this particular signal number which is an integer (i.e kill(pid, signum)). In order that the process handles that signal it is to define this particular function called signal, which takes the signal number for instance 9 or 11 and then the handler (i.e signal(signum, handler)). So, handler is a pointer to the signal handler for that particular number. So as a result we could send signals from one process to an other and depending on the type of the signal the corresponding handler would be executed.

In this video we had seen a brief introduction to IPC's. Essentially, we had seen different types of IPC's like, message passing, shared memory, pipes and asynchronous transfers

using signals. These IPC's are used extensively in systems to communicate between processes. And therefore, we are able to achieve or built applications which are extremely modular with small programs, with each program focusing on only a certain aspect in the entire system.

Thank you.

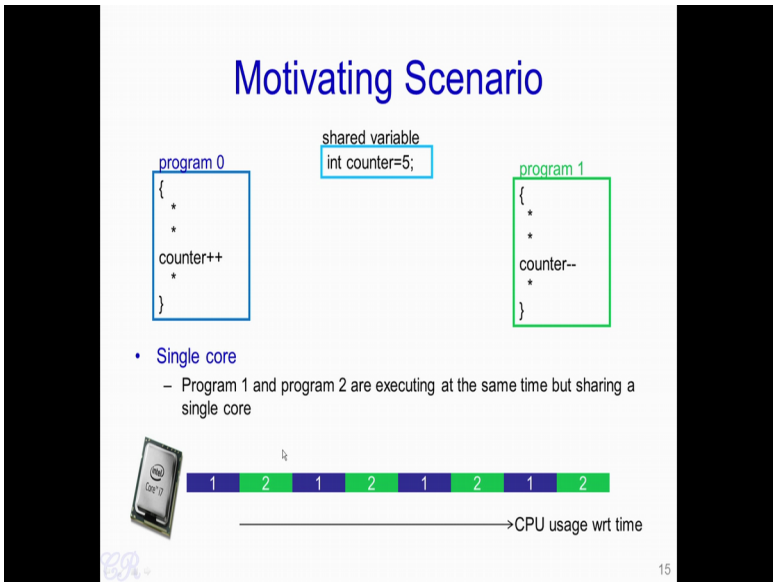
Introduction to Operating Systems
Prof. Chester Rebeiro
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

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. And 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 for 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)

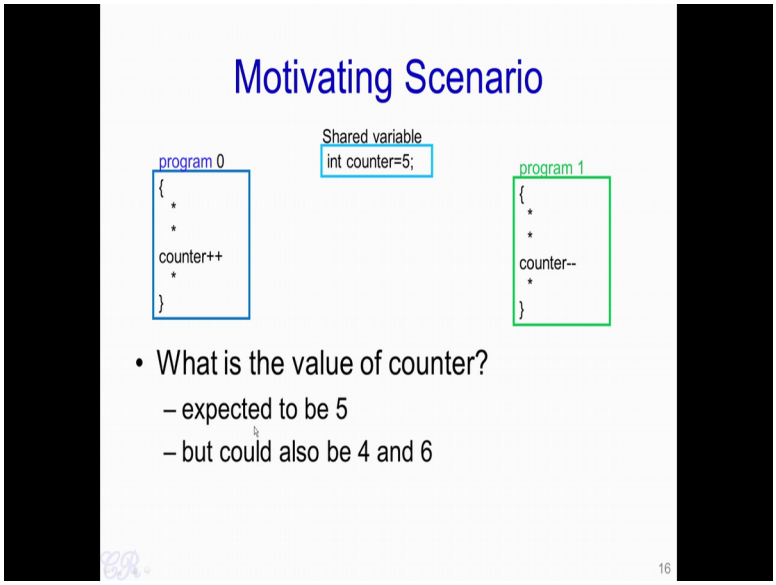


Let us explain what synchronization is with this motivating scenario (refer above slide). Let us say we have two programs; program 0 and program 1, and a shared variable which is defined as counter (defined in above slide image). It is a global shared variable int

counter = 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 core processor.

So, program 0 (program 1 in blue color mentioned above) 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)

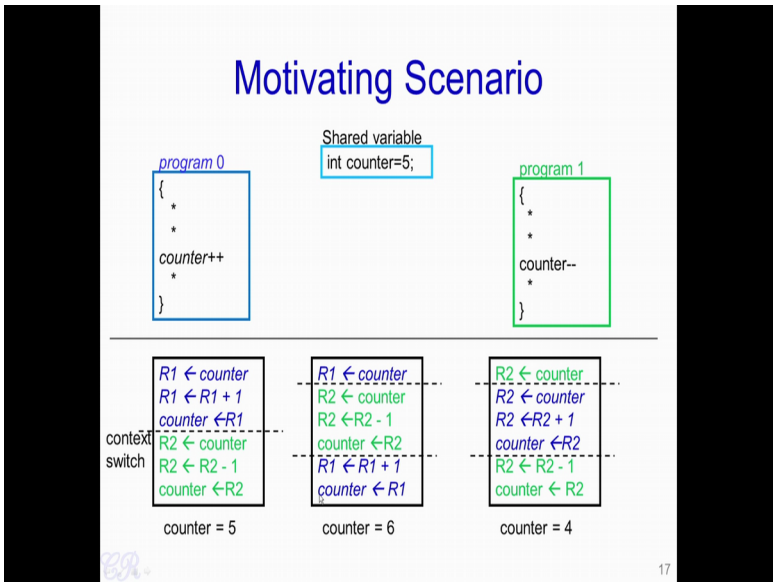


One would expect that the counter value is 5 because, let us say program 0 executes first so it is going to increment the value of counter, so counter becomes 6 over here (in program 0). And then program 1 executes and it decrements the value of the counter and since a counter is shared and has the value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



So, let us look more deeply into how these instructions (mentioned in above two programs) 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 doing counter++ with respect to the assembly instructions (mentioned below program 0 in above slide). So 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 = R1 + 1 and then R1 is return back into memory. So, R1 is return back into the value of counter which is stored in memory.

So in terms of the numbers, so we have 5 here which is the counter, so the contents of counter is 5 and that is loaded into R1 (i.e. R1 <- counter), then 5 is incremented so now R1 contains 6 (R1<- R1 + 1) and the value of R1 is return back into counter (i.e. counter <- R1). 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. So value of

counter is loaded into R2 (R2 <- counter), so the R2 has a value of 6 then we are decrementing R2 by 1 (R2 <- R2 – 1), so R2 becomes 6 - 1 that is 5. And then the register R2 is stored back into the memory location counter (counter <- R2). Therefore, R2 is 5; therefore the value of 5 is stored back into the counter. So at the end of these two programs executing the value of counter is 5.

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

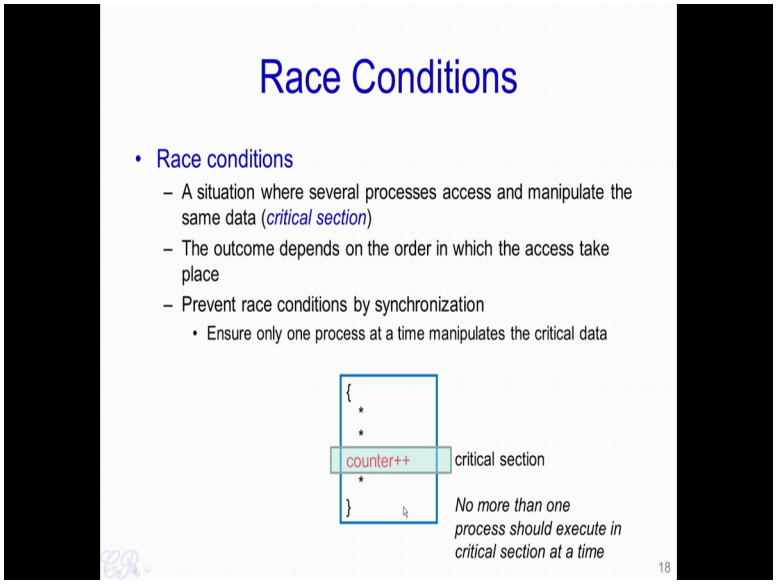
Now there is a context switch again and recollect, when there is a context switch the program 0 continues from where ever it has stopped. Essentially, the context switch was stored in the kernel is restored back into process 0 allowing it to continue from where it stopped. Therefore, we see that the value of R1 at this particular point was 5 (when R <- counter) and after the context switch we have R1 back at 5 again over here, so R1 is incremented by 1 (R1 <- R1 + 1) to get 6 and the value of 6 is stored into counter (counter <- R1). 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. So this is exactly the opposite to the second case in which the program 1 is executed first. So essentially the counter which has a value of 5 gets loaded into R2 (R2 <- counter), therefore R2 has 5. Now there is a context switch causing program 0 to execute and the value of counter is loaded into R2 incremented by 1 (R2 <- counter, R2 <- R2 + 1), so that is 6 and 6 is return back into the counter, so counter now will have the value of 6. So 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 that makes it 4 (R2

<- R2 – 1). And the value of 4 is stored into the memory location corresponding to counter (counter <- R2). 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. So 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 how the instructions get executed and how the context switch is occurred. So we would define this scenario more formally by what is known as Race Conditions.

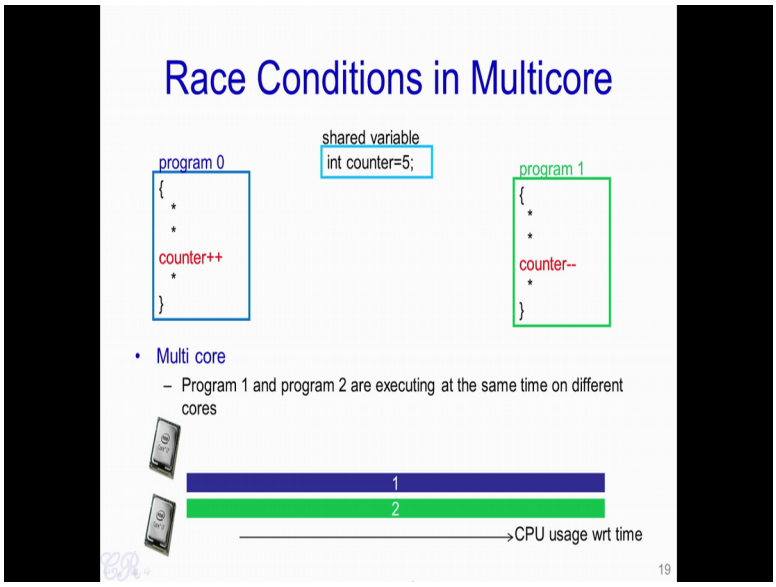
(Refer Slide Time: 07:43)



A Race Condition is a situation where several processes access and manipulate the same data. So this part of the process which accesses common 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.

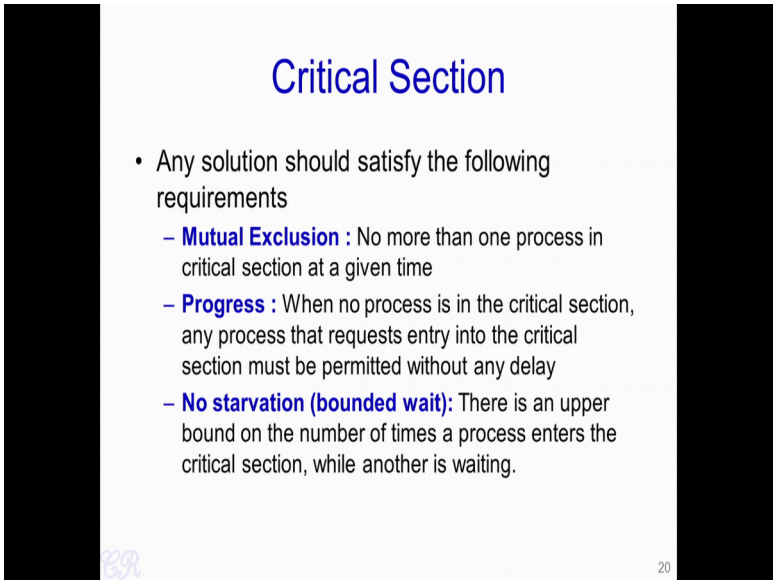
So coming back to our example what is required, is that we would mark this area of instructions which access or which manipulate the shared data as a critical section (refer above slide). Then we would have some additional techniques to ensure that no more than one process could execute in a critical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

(Refer Slide Time: 09:09)



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

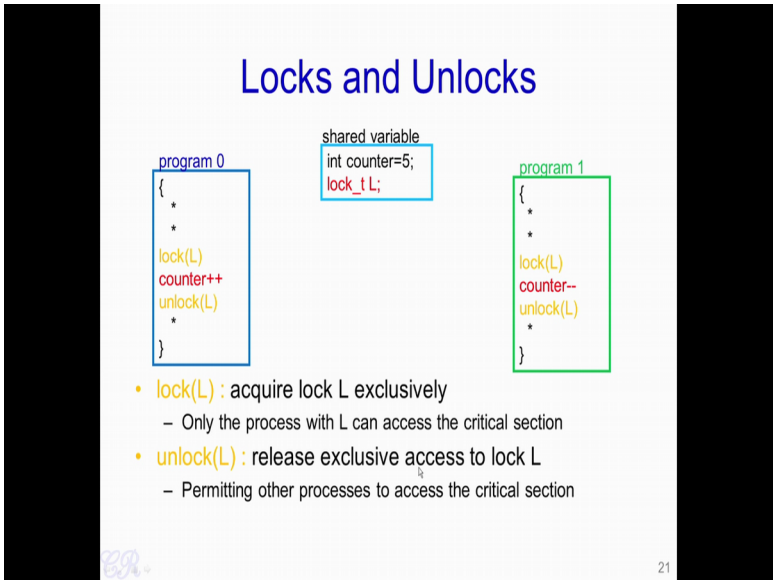
(Refer Slide Time: 09:42)



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; 1 Mutual Exclusion, 2 Progress, and 3 No starvation or bounded wait. In Mutual Exclusion, the critical section solution should ensure that not more than one process isin 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 that 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)



All critical section problems use techniques known as locking and unlocking in order to solve the critical section problem (refer above slide image). Essentially, in the solution we would have something like this (lock variable) which is also a shared variable. So we define something known as a lock and define it as an L (i.e. lock_t 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.

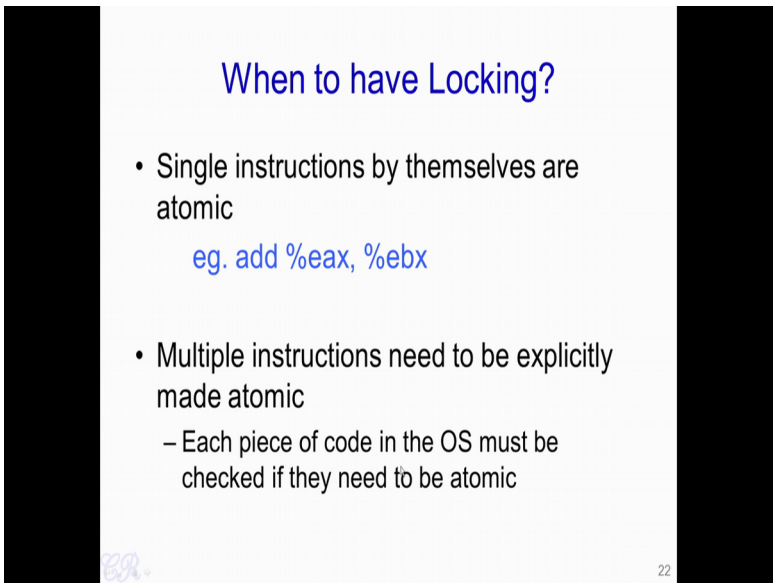
Now what lock does is that it acquires the lock (L) exclusively. So, after lock completes its execution that is after the function lock (L) completes execution, it is ensured that exactly one process, in this case (refer above two programs) only this process enters into the critical section and is present 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.

So 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. So 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 the 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)



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 an 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.

So 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.

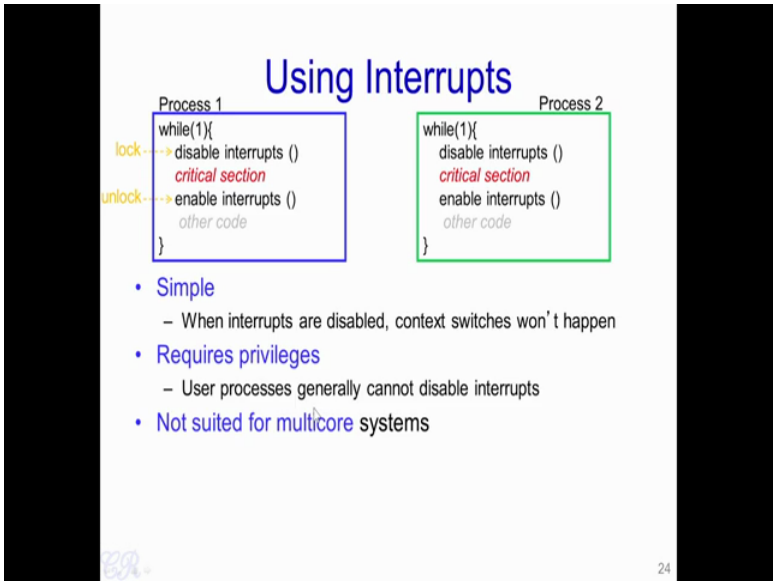
Thank you.

Introduction to Operating Systems
Prof. Chester Rebeiro
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Week - 06
Lecture – 25
How to Implement Locking (Software Solutions)

Hello. In the previous video, we had took, seen an Introduction to Critical Sections. We had seen that in order to solve the critical section problem, there were 3 requirements. 1st is mutual exclusion, 2nd progress and 3rd bounded weight. In this video, we will see some software solutions for the critical section problem.

(Refer Slide Time: 00:43)



So let us start with the more simple solution that is by disabling interrupts, so this solution is only applicable for single core systems. With multicore systems, essentially because of the advanced programmable interrupt control is present in the systems; interrupts are routed to several processors. Therefore this solution will not work on multicore systems. On the other hand, for single core processors or single core systems, disabling interrupts will work. Essentially we had seen that if interrupts are disabled then it would prevent context switching and preventing context switching would mean that the critical section would be not pre-empted during its execution.

Let us look at how interrupts can be used to solve the critical section problem (refer above slide). So, we have the two processes, process 1 and process 2 and they have this critical section which is shown in red. In this critical section, both processes access or modify the same shared data. In order to ensure that the critical section problem is solved and that both processes do not end up in the critical section at the same time, we disable interrupts before entering into the critical section.

While, on the other hand, while leaving the critical section the interrupts are enabled again. Disabling interrupts as we have seen will prevent the process from getting pre empted at the end of its time slice. Essentially, there would not be a timer interrupt that would occur which would prevent process 1 from pre-empting. Therefore, we see that this is a very simple solution and process 1 will continue to execute without any other interrupts at occurring as a result of the disabling of interrupts.

Similarly, when process 2 enters into this critical section, it disables interrupts; and on leaving the critical section, it enables interrupts again (mentioned in above slide). So, this application of disabling and enabling interrupts is similar to the locking mechanism before entering the critical section and the unlocking mechanism at the exit of the critical section. The locking will ensure that only one process enters or executes in the critical section at a given time. While the unlocking mechanism that is when interrupts are enabled would then allow other processes the chance to enter or execute in the critical section.

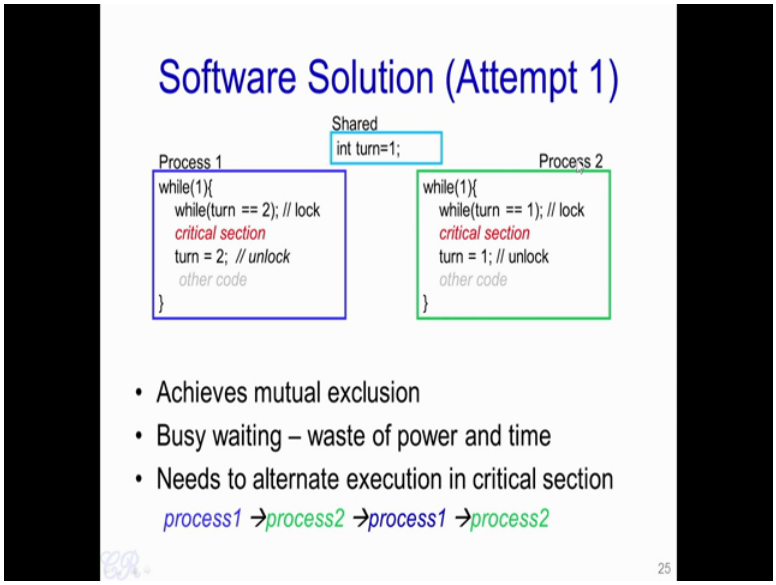
So, this method of solving the critical section problem is simple. It just requires a single instruction in order to disable or enable interrupts. Thus, preventing context switches from happening. However, the limitations of using interrupts is that it requires a higher privilege level. So, normal application programs which run in user space will not be able to disable and enable interrupts in such a way. Therefore, this solution is only applicable for code that runs in the kernel.

So it is only the kernel or the operating system that is allowed to disable interrupts and enable interrupts again. Therefore, this solution is only applicable for operating system code. And as we have seen before this disabling and enter enabling of interrupts is not suited for multicore systems; essentially because when interrupts are disabled on a multicore system, it would mean that interrupts corresponding to that code alone is

disabled. On the other hand, interrupts are still allowed on the other codes which are running. Thus on a multicore system when disable interrupts is invoked, all interrupts to only that code or that code in the processor is disabled. Other processors such as a processor which is running this process 2 (mentioned in above slide) will not be affected by the disabling of interrupts.

Therefore, this solution is not suited for multicore systems. So, what we will do next is we will try to actually build our own solution for the critical section problem. So, we will start with a very simple solution, we will see its drawbacks, and then we will gradually modify that particular solution until we reach a point where we have solved the critical section problem. So let us look at this.

(Refer Slide Time: 05:47).



So let us start with our first attempt. So let us say we have this critical section in process 1 and this critical section in process 2 (refer above slide) and both these critical sections are manipulating the same data. In addition to this shared data, we also define an other shared data known as turn, and set this value of turn to 1 (i.e turn = 1;). So, what happens we look into more detail? So, when process 1 begins to execute it sees that turn = 1, therefore this while loop immediately exits (because while condition is false) and the critical section gets executed.

Now during the time when process 1 is in the critical section, when process 2 arrives at this particular point, it sees that turn = 1, and therefore it will continue to loop in this

particular while loop (because while condition is true). The only time that it is capable of exiting from this while loop is when the 1st process sets turn to 2, so when the value of turn is set to 2, and remember that turn is a shared variable, therefore, this while loop in process 2 will terminate. And process 2 will then enter into the critical section. At the exit of the critical section, process 2 will set the value of turn to 1 (i.e turn = 1;). So, setting the value of turn to 1 would mean that a process 1 can then enter into the critical section.

So, essentially when we look at the locking and unlocking mechanism to solve this critical section problem, we see that this while loop present over here (second while in process) is the locking mechanism, while the unlocking mechanism is the third line that is setting the value of turn to 2 (i.e turn = 2 in process 1). The second observation that we make is that this algorithm or this solution for the critical section problem requires that process 1 executes into the critical section and then because turn is set to 2 over here (last line in process 1), after process 1 completes its critical section execution, process 2 should execute. So, once process 2 completes its execution, process 1 will execute and so on.

So essentially there is an alternating behaviour for the critical section (refer last bullet point in above slide image). First, process 1 needs to execute the critical section because the value of turn is equal to 1 (turn = 1;). Then when it completes it sets turn to 2 (turn = 2;), so process 2 will execute. And process 2 will set the value of turn to 1 (turn = 1;) at the end of its critical section which will result in process 1 executing in the critical section and so on.

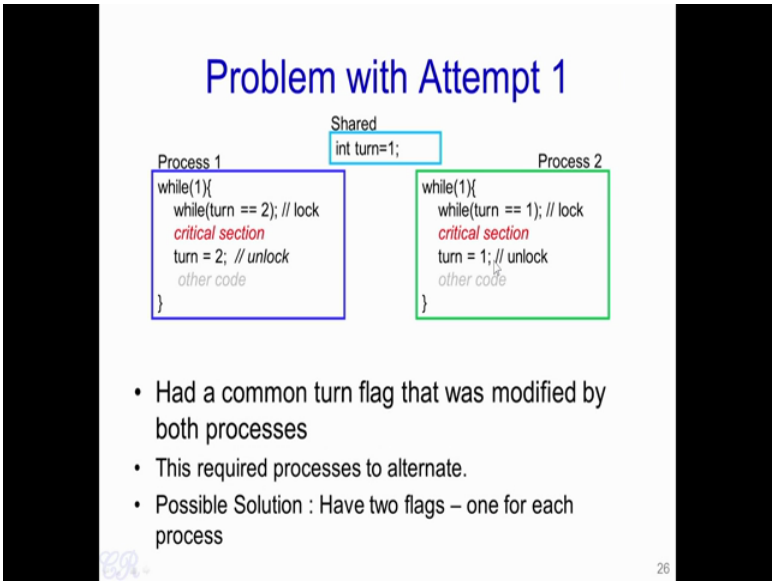
So, we see that there is an alternating behaviour between process 1 and process 2. Quite naturally because the value of turn is shared between both the processes and as we have seen only one process could satisfy this while condition, and break out from this while loop. Therefore, exactly one process would enter into the critical section at a given time. Therefore, this solution achieves mutual exclusion. However, there are two limitations of this solution; 1 is the busy waiting. So while process 1 is executing in this critical section, process 2 would endlessly execute in this particular while loop.

Note that process 2 is in the ready state or the running state it is not in the blocked state and therefore, it would consume power as well as time, because it needs to execute in the

CPU periodically in order that this value of turn is checked. Another limitation as we have seen before it is that this solution for the critical section problem requires that process 1 and process 2 alternates its access into the critical section. So what it means is that first process 1 should execute then process 2, process 1, process 2 and so on. So, this creates a problem especially with the progress requirement of the critical section solution.

So, the progress requirement condition stated that if no process is in the critical section, and a process request for the critical section, then it should be offered the lock, essentially it should be able to execute in the critical section. We see that this solution will not satisfy the progress requirement. For instance (refer above slide image) let us say that process 2 begins to execute before process 1. So process 2 comes here (check while condition) and it executes this while turn equal to 1, and we see that since process 1 has not yet started. So if the progress condition needs to be met, so process 2 should enter into the critical section. But this is not so, therefore this is not a good solution for the critical section problem.

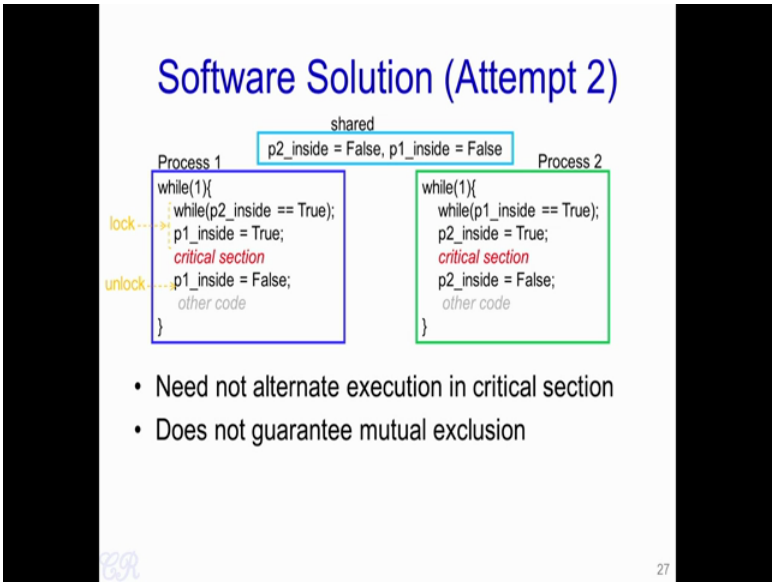
(Refer Slide Time: 11:16)



The main drawback or the main problem with our first attempt to solve the critical section problem was that it used a common turn flag that was modified by both processes. So, this turn flag was set to 2 in process 1 and set to 1 in process 2 (last line in both the process); and this force the two processes to alternate execution in the critical

section. So, one possible solution for this or one possible thing we could actually make better in our second attempt is to not have a single shared flag, instead we will have 2 flags one for each process.

(Refer Slide Time: 12:02)



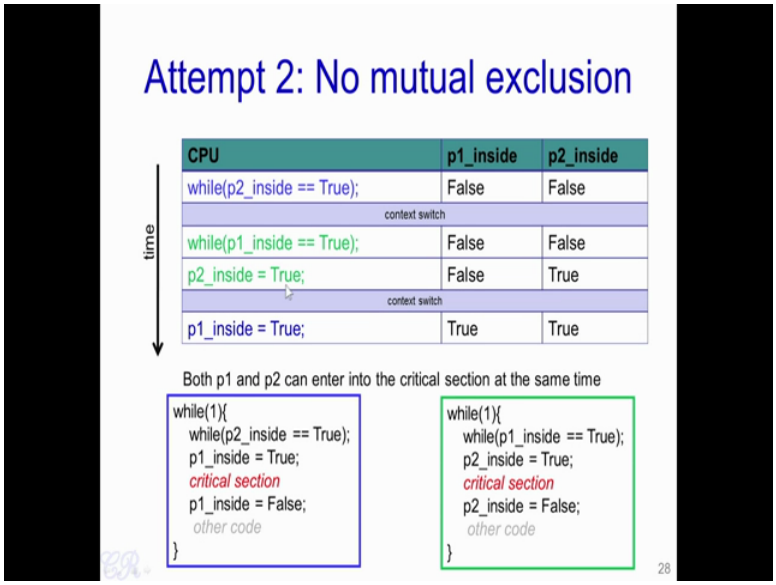
So let us see the second attempt to solve the critical section problem. So, this is the second attempt (refer above slide image). So, unlike the previous attempt we now have 2 flags present. These 2 flags are also shared among the two processes; that means that this flag p2_inside can be accessed from both process 1 as well as process 2. However, it is process 2 which actually changes or modifies this flag, so process 2 changes from false to true or true to false, while process 1 only reads the status of this flag; it only determines whether the flag is a true or a false.

Similarly, the second flag is p1_inside. Now these flags p2_inside and p1_inside are used during the locking and unlocking of the critical section. Essentially before process 1 enters into the critical section, it does two things. 1st, it needs to check that the second process that is p 2 is not inside the critical section and this happens when p2_inside = false. So, if p2_inside = true, it would mean that the second process is executing in the critical section. Then it sets it own flag that is p1_inside = true and executes in the critical section (process 1 critical section), while at the exit of the critical section it sets p1_inside = false.

Similarly, process 2 would first execute this while loop (check while condition). Until p1_inside = false (i.e already set in process 1 while exiting), so a value of p1_inside is false would mean that process p 1 has exited from the critical section and is no longer in the critical section. And after it obtain such a condition, the while loop is completed and execution comes over here where process 2 sets p2_inside = true indicating that it is inside the critical section and at the end of the critical section, just before exiting it sets that p2_inside = false.

Now this second solution does not require that the two processes alternate execution in the critical section. So, for instance, since the initial values of p1_inside and p2_inside are false, suppose process 2 executes first, so it sees p1_inside = false, so it breaks from this (while) loop and enters into the critical section. Similarly, if process 1 does not start executing, it will keep entering into the critical section without requiring process 1 to execute. However, the problem with this particular solution is that it does not guarantee mutual exclusion rather under some circumstances it is possible that process 1 as well as process 2 is present in the critical section at the same instant of time.

(Refer Slide Time: 15:45).



So let us see when this mutual exclusion does not hold for our second attempt. This particular table here (refer above slide image) shows the execution of various statements in the CPU with respect to time that means, that this particular while loop (while loop in first row of the table) executes and while since it is in blue colour it means that it

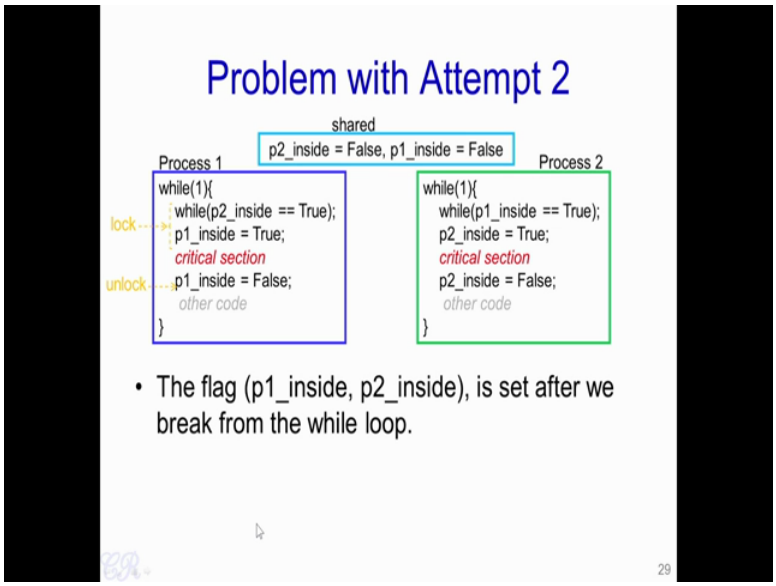
corresponds to the process 1. Then while (p1_inside == true) (third row in the table) and p2_inside = true (fourth row in the table), so this is in the green colour, so it corresponds to process 2. And p1_inside = true (sixth row in the table) again corresponds to process 1.

So, this is how the CPU would execute instructions corresponding to the two processes and we have like several context switches that occur during the execution and these are the status of the various flags as the execution progresses (second and third column in the table).

So let us start with process 1, we have initially set that p1_inside and p2_inside is both false, indicating that both processes are not in the critical section. Therefore, this particular while loop (while loop mentioned in first row) will complete, essentially because p2_inside is false; now suppose there is a context switch that occurs resulting in process p 2 which executes, so it will also execute these statements while p1_inside is false (mentioned in third row of the table), so the while loop will break. And then it sets p2_inside = true (fourth row of the table)(.

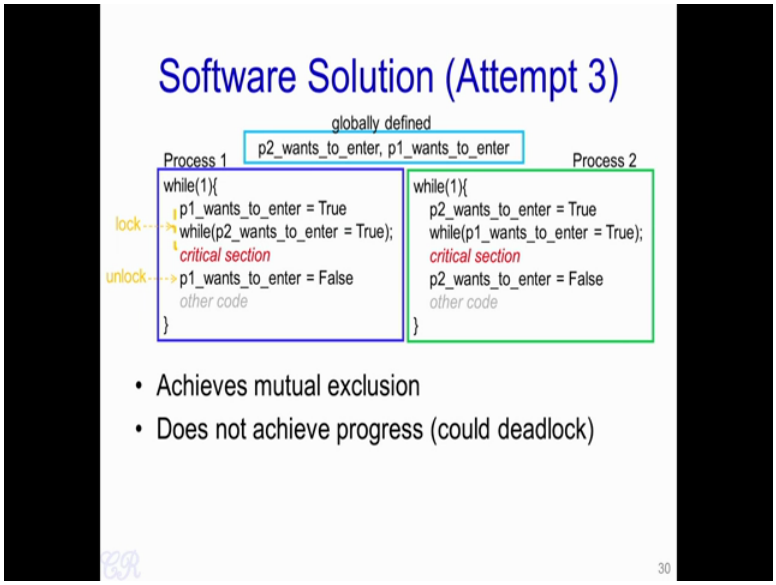
Now immediately after setting this suppose there is a context switch that occurs and process p 1 will continue to execute and it sets p1_inside = true (mentioned in last row). Essentially due to the context switch, it will continue to execute from where it has stopped that is it has stopped just soon after completion of the while loop (first while loop in the table) and then it continues with the next instruction over here (last row in the table) that is setting of p1_inside = true. So, here we see we have a state where p2_inside = true (mentioned in fourth row)indicating that p 2 is in the critical section as well as p1_inside = true indicating that p 1 is also in the critical section. Therefore, we have not able to achieve mutual exclusion.

(Refer Slide Time: 18:15).



So, the main problem with this second attempt to solve the critical section problem was that we had two flags p1_inside and p2_inside. However, they were set after we break from the while loop (second while condition in both process) that is only after this particular while loop completes execution and breaks from this only then are we setting p1_inside = true and p2_inside = true . Essentially, what this means is that p1_inside and p2_inside are set to true only within the critical section, and this is what created the problem. So let us see if we can actually change these ordering a bit.

(Refer Slide Time: 19:03)



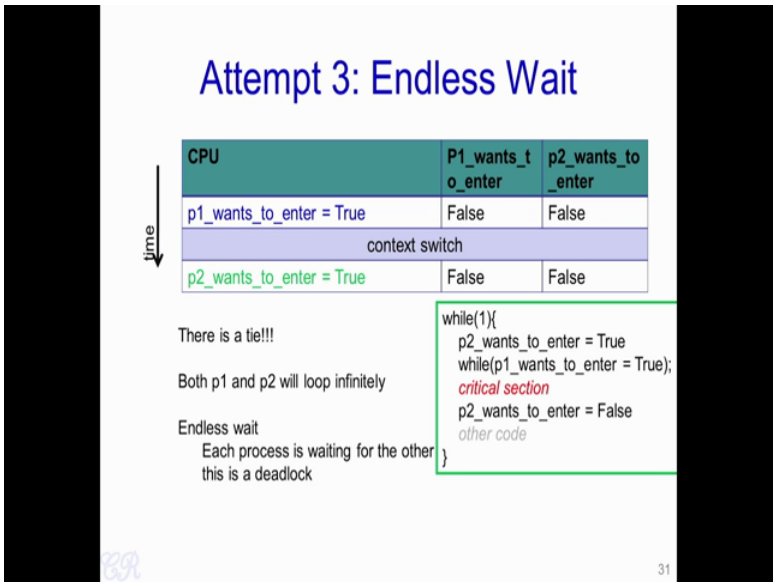
So, we will start with the third attempt that is the third attempt for a solving the critical section problem. And what we will do is just a simple change, so we will have these two flags instead of p1_inside, we will have now p1_wants_to_enter into the critical section, and p2_wants_to_enter into the critical section. Essentially, each of these flags are set to true when the corresponding process wants to enter into the critical section. So, the minor change we do over here with respect to the second attempt is that process 1 first shows his intention to enter into the critical section by setting p1_wants_to_enter = true.

And only then will it try to determine, if process p 2 is in the critical section or not. So, essentially it sets p1_wants_to_enter to true and then it would execute in this while loop until p2_wants_to_enter is false. So, when p2_wants_to_enter it becomes false indicating that the 2nd process has completed execution in the critical section then process 1 will enter into the critical section.

At the end, after the critical section has completed execution p 1 sets the flag p1_wants_to_enter = false, so this setting of false will allow process 2 to enter into the critical section. Note that the only difference of this, corresponding to our previous attempt was that we move the flag from inside the critical section that is after the while loop to before the while loop. So, what we see is that this particular solution achieves mutual exclusion, so, unlike the previous attempt where we obtained a condition where mutual exclusion was not achieved and we had two processes in the critical section at the same time.

In this solution, the mutual exclusion is achieved; essentially, it is guaranteed that when process 1 is in the critical section, process 2 is not in the critical section and vice versa. So, what is not guaranteed is Progress. Essentially, we could obtain a state where both processes are endlessly waiting or endlessly executing in this while loop and we will see how this thing occurs, so such a state is known as a deadlocked state.

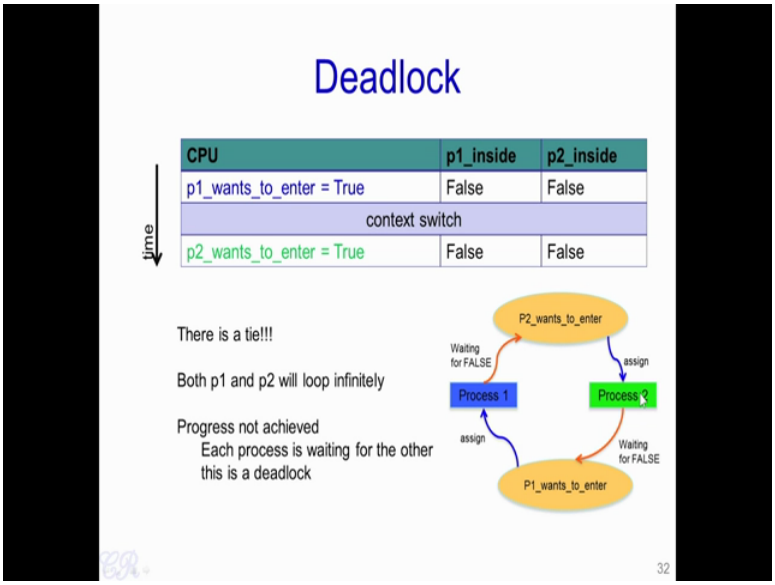
(Refer Slide Time: 21:54)



So let's see the deadlock situation in this particular case. So let us look in this table (refer above slide), which shows how the CPU executes instructions with respect to time, and also this table shows the various values of the flags which are present. Let us start with process 1 executing and the CPU executes this particular statement (first row statement) and it sets the value of p1_wants_to_enter = true. Then there is a context switch, which results in process 2 executing which sets the flag p2_wants_to_enter = true.

Now you see that both processes have set their respective flags to true (as mentioned above). Now when both processes enter into this while loop (second while loop in above program), we see that in both cases process 1 and as well as process 2, this flag is set to true indicating that both processes will continue to execute this while loop endlessly. Essentially, process 1 is waiting for process 2 to set its flag to false, while process 2 is waiting or is waiting for process 1 to set its flag to false. So, essentially we have reached a state which where each process is waiting for the other process to do something. So, this is what is known as a deadlocked situation and it could lead to considerable amongst of problem in operating systems.

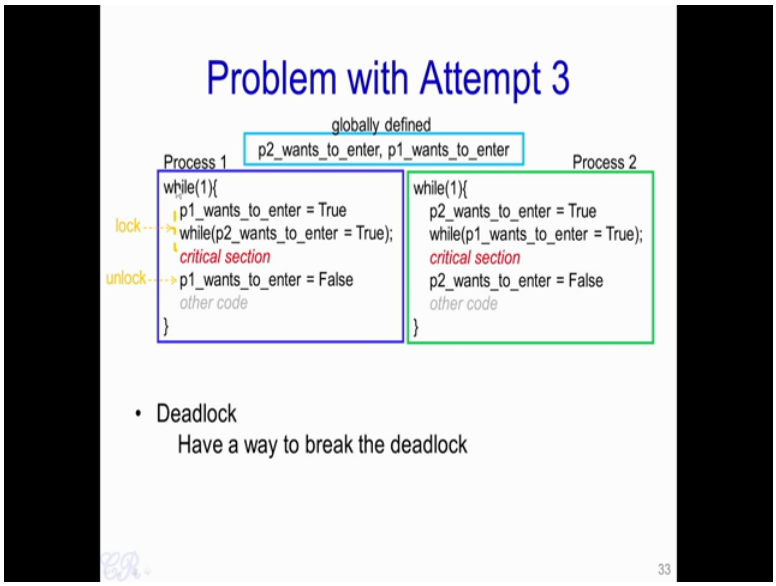
(Refer Slide Time: 23:39).



So, to give a more clearer view of what a deadlock is we have this particular figure here (mentioned in above slide), where we have process 1 (blue colour) and process 2 (green colour) present here. Now process 1 is in the while loop and it is looping continuously waiting for process 2 to set this flag p2_wants_to_enter = false. Similarly, process 2 is waiting in the while loop or looping continuously and waiting for this flag p1_wants_to_enter = false. However, the problem here is that process 1 needs to change the value of this flag (p1_wants_to_enter) because this flag can be only changed by process 1; similarly process 2 needs to change the value of this flag (p2_wants_to_enter) since this flag can be only changed by process 2.

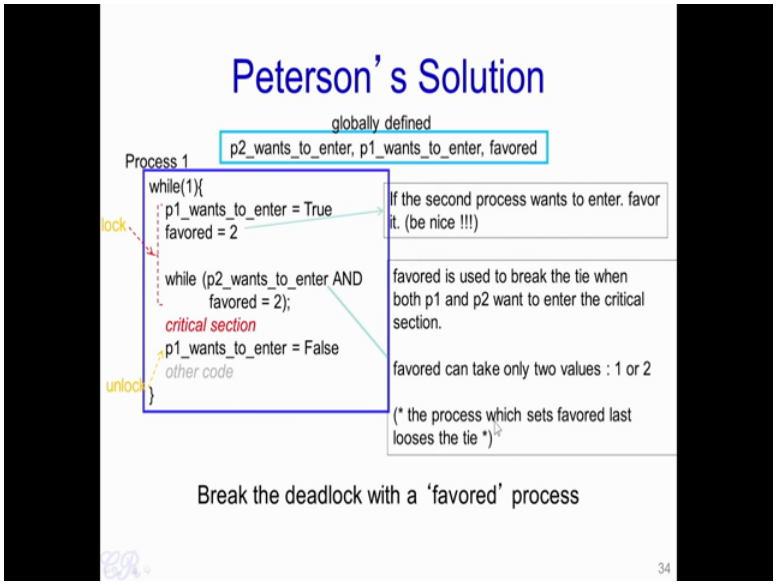
And therefore, we see that there is a tie; process 1 is waiting for process 2 to change the value of its flag (p2_wants_to_enter); however, process 2 cannot change the value of its flag because it is waiting for process 1 to change the value of the flag (p1_wants_to_enter). So, we get some kind of a cycle present over here (refer above slide image). So, this cycle will result in both process 1 and process 2 waiting for each other for an endless amount of time, so this is known as a deadlock. So, in a later video, we will look more into details about how deadlocks are caused and how they can be prevented and avoided.

(Refer Slide Time: 25:18).



So let us see what the problem was with the third attempt. Essentially we had the process setting its flag and waiting for the other processes flag to be set to false. And we have seen that because both processes may enter into the while loop and wait for each other, we would have a deadlock situation. Now the next best thing to do is whenever we have such a deadlock situation, we need to find a way to actually come out of the deadlock situation. Essentially we need to ensure that one and exactly one of these two processes will break from the while loop and enter into the critical section.

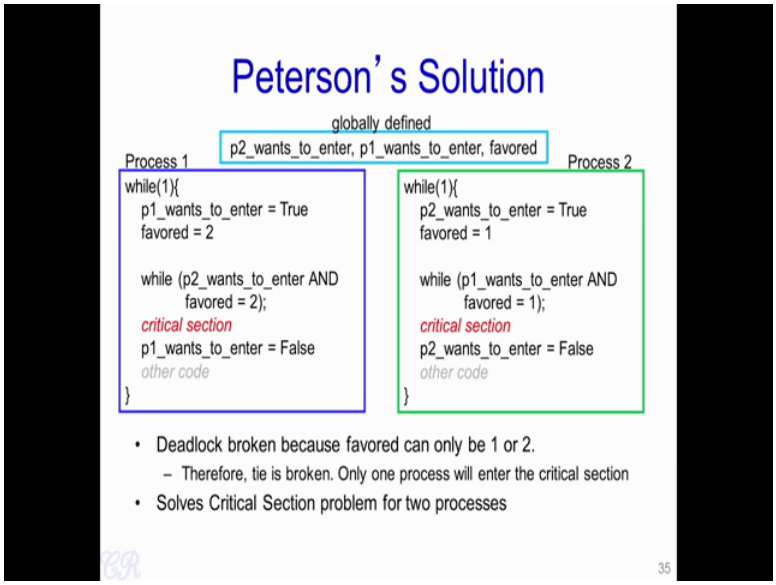
(Refer Slide Time: 26:03)



So, this solution was made by Peterson, and it is what is known as a Peterson's solution, and essentially has an other variable known as favored (refer above slide image), so this is a globally defined variable which is shared among the two processes. Essentially, what the favored variable does is that whenever deadlock situation occurs, it favors one of the processes, and that process would then enter into the critical section. While the other process will continue to wait until the second process sets its flag to false.

So, essentially Peterson's solution sets the value of favor to the other process so that it favors the other process to enter into the critical section. Now favored it is used to break the tie when both p 1 and p 2 want to enter into the critical section that is when both p 1 and p 2 are waiting in this while loop or rather are executing in this while loop. The reason this particular solution works is that, out of the two processes there is exactly two values that favored can take, it can take only one or two, and therefore only one of these while loops will actually break or complete execution.

(Refer Slide Time: 27:31)



So let us see this in more detail. So, we have seen the two processes p 1 and p 2, and everything is as the third attempt that we want that p1_wants_to_enter set to true and then there is a while loop and then the critical section and then p1_wants_to_enter set to false (refer above image process 1). Now we also have a favored added here in the Peterson solution and this favored is set to 2 over here and favored is set to 1 in the 2nd process (mentioned in above slide). So, in all other cases this particular Peterson's

solution behaves like our third attempt to create the critical section solution. However the only difference comes when both processes enter into this while loop at the same time. In such a case, the value of favored will determine which of the two processes enter into the critical section.

So let us say process 2 is favored and it enters into the critical section, and at the end of its critical section it sets p2_wants_to_enter to false. So, remember that during this entire period p 1 is still waiting or still executing in the while loop. Now when process 2 sets its flag to false, it immediately causes this while loop to break (process 1 while loop), as a result process 1 will enter into the critical section. So, you see that only when process 2 exits from the critical section only then will process 1 enter into the critical section, thus achieving mutual exclusion. Also, we had seen that the bounded weight issue is solved and as well as the progress thing is solved. The Peterson's solution works very efficiently for two processes.

In the next video, we will see the bakery algorithm which is used to solve the critical section problem when there are multiple processes in the system. So, the bakery algorithm solves the critical sectional problem when the number of processes is greater than two.

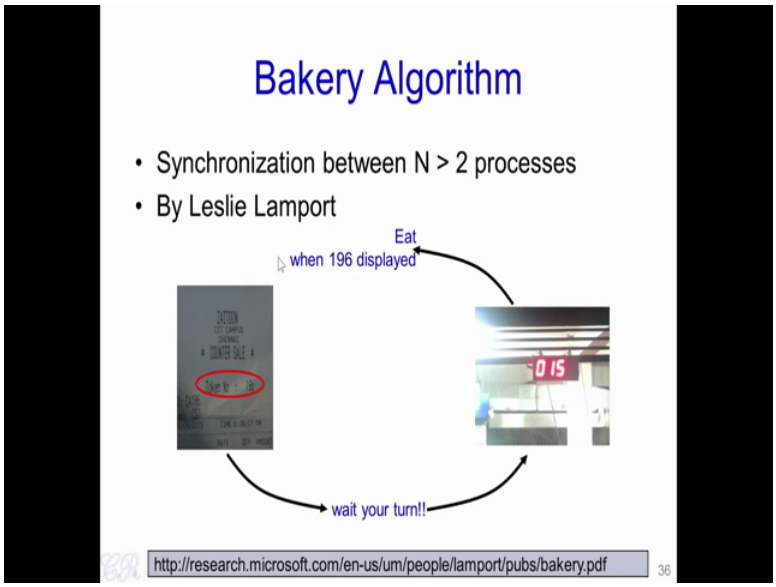
Thank you.

Introduction to Operating Systems
Prof. Chester Rebeiro
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Week - 06
Lecture – 26
The Bakery Algorithm for Synchronization

In this video, we will look at the Bakery Algorithm, which is a software solution for the critical section problem. Essentially compared to the Peterson's Solution which we seen previously, this algorithm is most suited for larger number of processes. Essentially when we have the number of processes which is greater than 2, then the bakery algorithm would work efficiently.

(Refer Slide Time: 00:43)

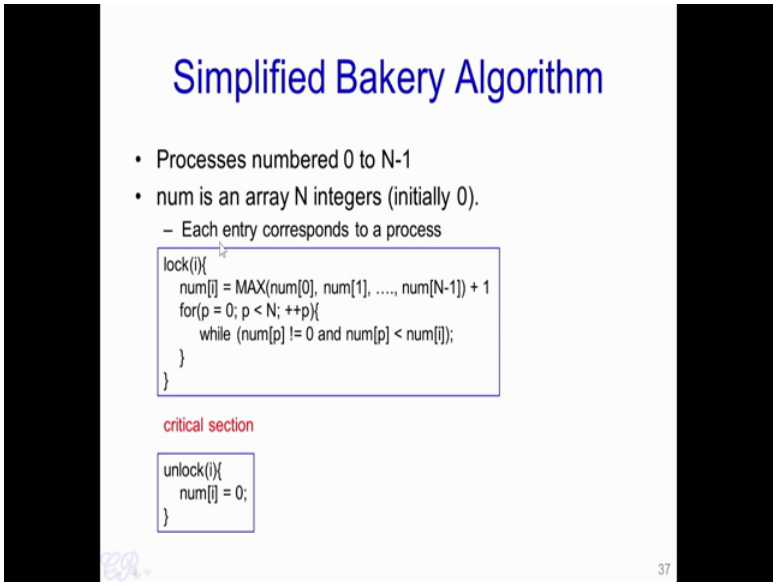


So, the bakery algorithm was invented by Leslie Lamport, and you can get more information from this particular website present here (mentioned in above image). So, the essential aspect of the bakery algorithm is the inspiration from bakeries and banks. In some bakeries, what we see is that when we enter the bakery, we are given a particular token for instance in a bakery, we would be given a token with a number present over here (mentioned in token in above image).

So in this particular case, we have a token number 196 which is present. Now we need to wait for some time until the token number 196 is called out. So, we have a display over

here (mentioned in above slide) which periodically would set a token number, and when the token number of 196 is displayed then you are able to get your food from the bakery and you could eat. So, essentially when we look at this from synchronization aspect, we see that we are trying to synchronize the usage of a particular counter. So all people who have such a token should wait until their number is called, then sequentially each person depending on when the number is called goes into the counter, and is able to collect whatever he or she wants and for instance eat.

(Refer Slide Time: 02:16)



So, we will see how the bakery algorithm is used to solve the critical section problem. So, we will start with the simplified analysis of the bakery algorithm. So, this particular algorithm (mentioned in above slide) is used to solve the critical section problem when there are N processes involved and all these N processes access the same critical section. Now there is also a global data which is shared among these N processes and this data is known as num. So, the length of num is of size N; essentially each process has a particular index in this num array.

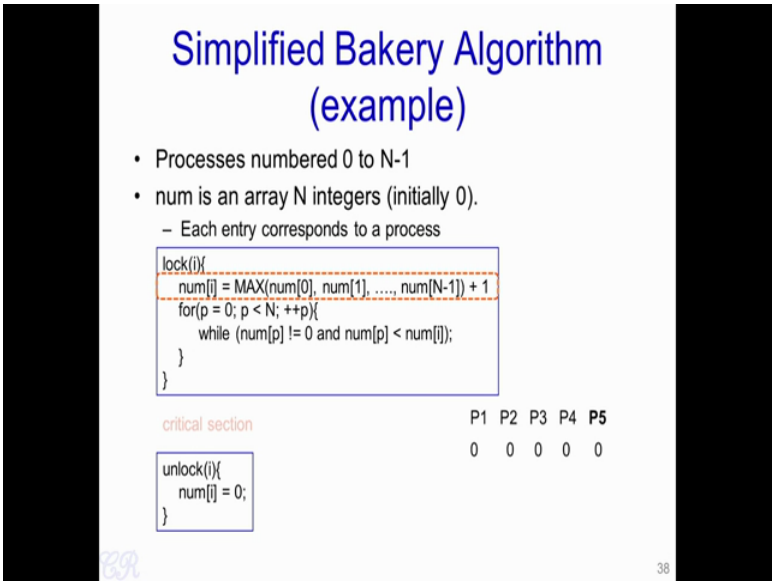
So, for instance process 0 would have a flag corresponding to num[0], process 1 has num[1], process 2 has num[2] and so on. Secondly, at the start of execution, this value of num is all set to 0s. Now in order to enter a critical section, a process would first need to invoke the lock call with the value of i (i.e lock(i)). So, i here is the process number for example, a process id. So, we have N processes so, the value of i could be from 0 to N -

1. And at the exit of the critical section, the function unlock i is invoked (i.e. unlock(i)) where the value of num[i] is set to 0.

So let us see what lock and unlock is actually doing internally. So, when a process invokes lock(i) where i is its number; its corresponding num value. So, num[i] is set to the maximum value, essentially this particular function MAX (mentioned in above image) is going to look at all the numbers corresponding to all of the processes and get the maximum from that and add one to that. So, num[i] is going to get the highest number which is present among the shared num array. Second there is for loop, which scans through all processes. So, for(p=0;p<N;++p).

And within this for loop (as mentioned earlier), there is a while loop which checks two things, it checks that num[p] != 0 and num[p] < num[i]. So, essentially this particular while loop will break, when either num[p] is 0 or this particular condition (i.e num[p] < num[i]) is false. So, essentially num[i] <= num[p]. So, essentially, the process i will enter into the critical section only if it has the lowest nonzero value of num[i].

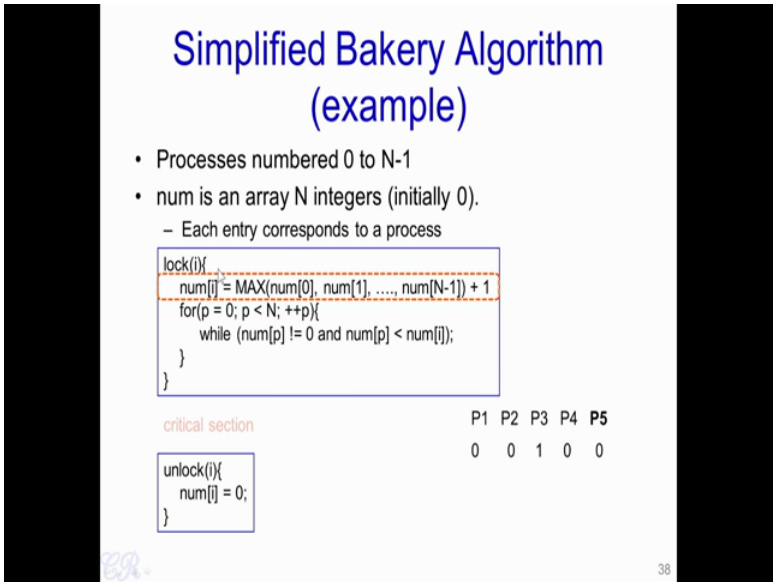
(Refer Slide Time: 05:25)



So let us look at this within an example (refer above slide image). So let us assume that we have five processes P 1 to P 5, and these are the num values (initially 0). So, the num array is also having 5 elements, and each of these elements corresponds to a process. So num[0] corresponds to process P 1, and num[1] corresponds to P 2, num[2] corresponds to P 3 and so on. Now let us also assume that all these processes almost simultaneously

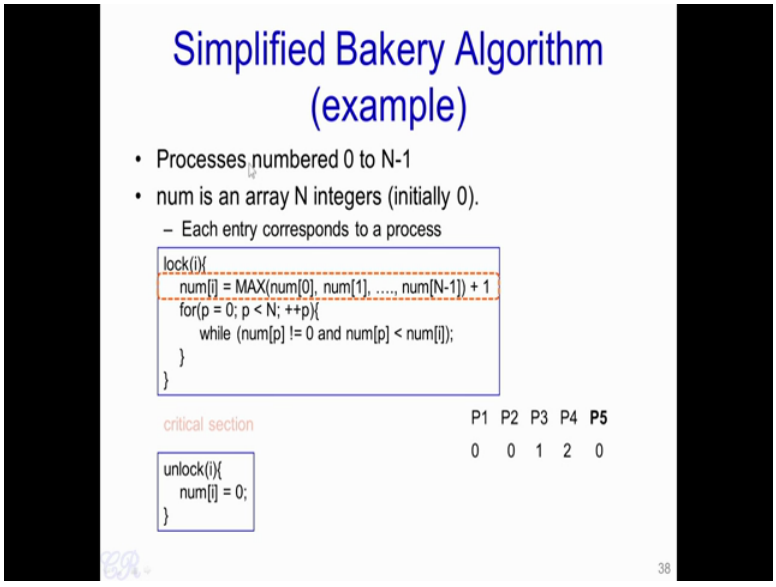
invoke the lock(i) function, essentially all these processes want to enter into the critical section almost simultaneously. Then what would happen?

(Refer Slide Time: 06:11)



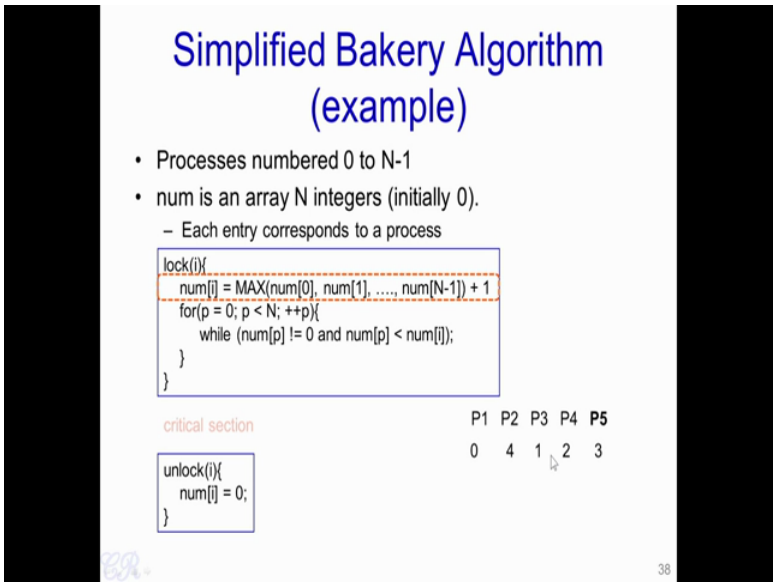
Let us say process P 3 begins to execute. So, it comes here (red box in above image) and it finds, the CPU finds that the MAX of all these numbers which are present is 0, so num of a corresponding to P 3 is set to 1.

(Refer Slide Time: 06:29)



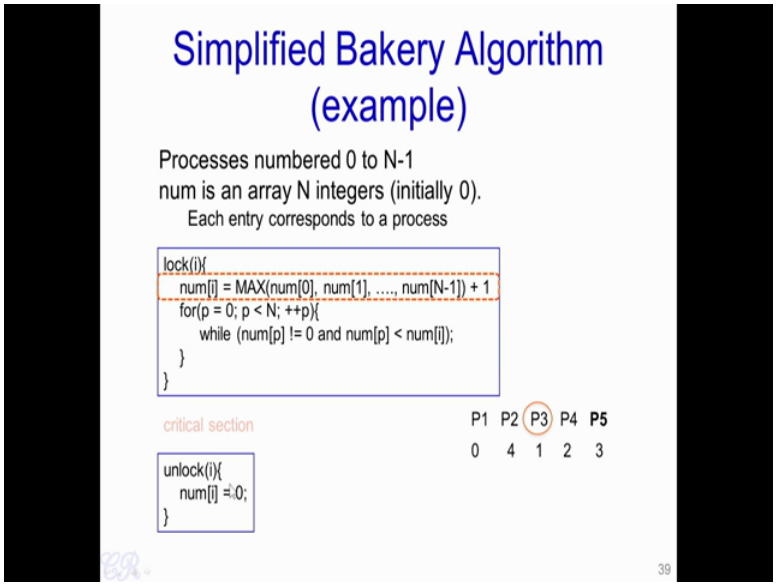
Then let us say P 4 executes and it gets a value of num[4] equal to 2 (mentioned in above slide below P4).

(Refer Slide Time: 06:39)



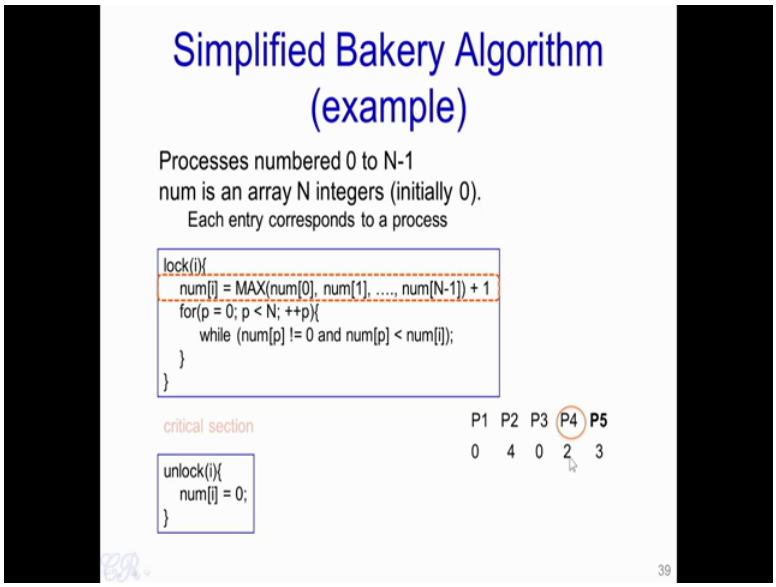
Then P 5 executes, and P 2 executes and they get corresponding values of num as 3 and 4 (mentioned in above image). Now, let us say that we come into this part of the loop (i.e for loop), and we see that the process with the lowest nonzero value of num would enter into the critical section. So in this particular case, we scan through all these particular values of num and we see that P 3 has the lowest value. Therefore, P 3 needs to execute in the critical section.

(Refer Slide Time: 07:10)



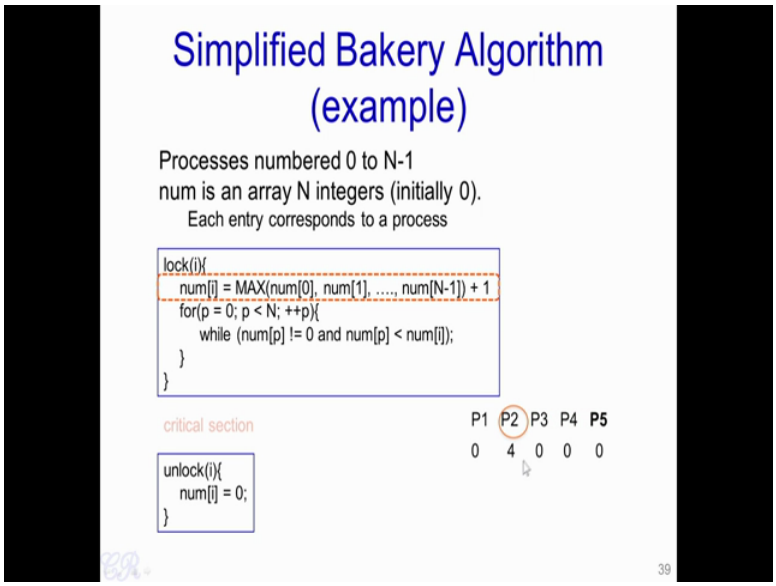
So, P 3 executes in the critical section and at the end of the critical section it sets the corresponding value to 0 (i.e num[i] = 0) then because other processes are also waiting in this particular loop.

(Refer Slide Time: 07:25)



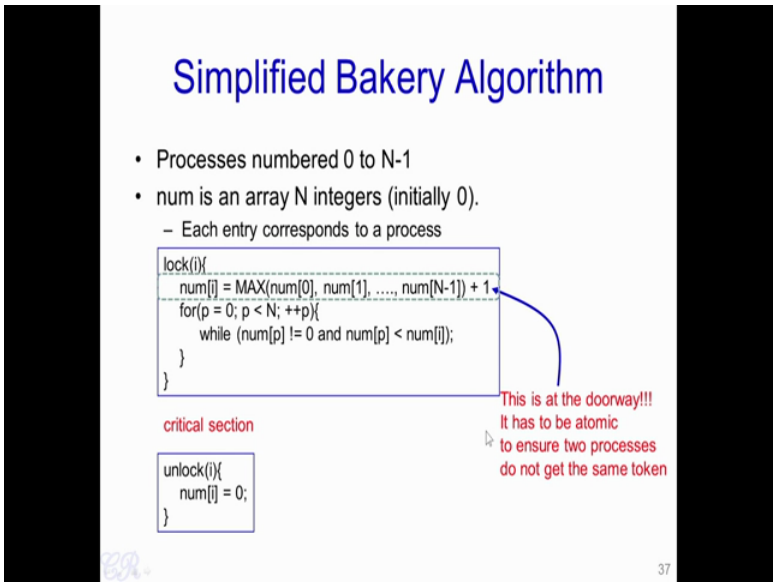
So, the next lowest number which corresponds to the process P 4 and has a value of 2 would get to execute in the critical section. Therefore, process P 4 enters into the critical section and at the end of it the number corresponding to process P 4 is set to 0. And then process P 5 executes and then process P 2 executes (mentioned in above image and setting value of num to 0).

(Refer Slide Time: 07:44)



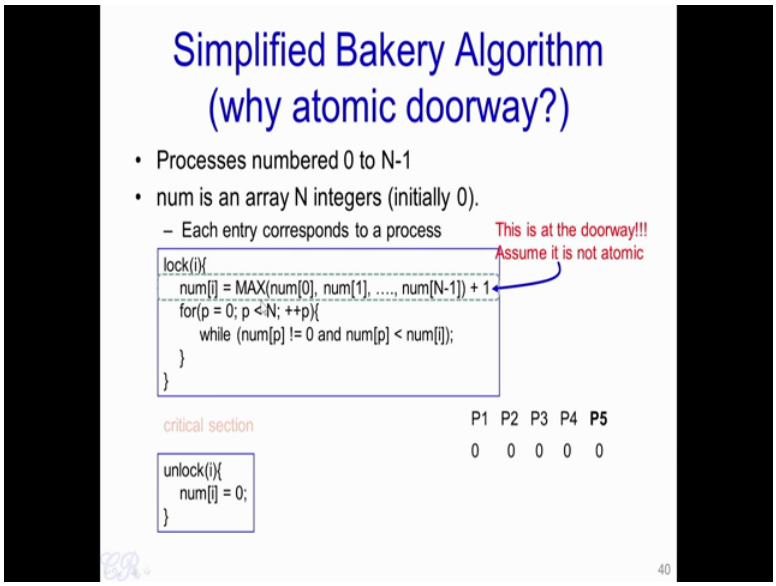
So, process P 2 would execute, because it is the only nonzero number which is present. So, at the end of the P 2 execution, we get all values of num which are back to 0.

(Refer Slide Time: 08:06)



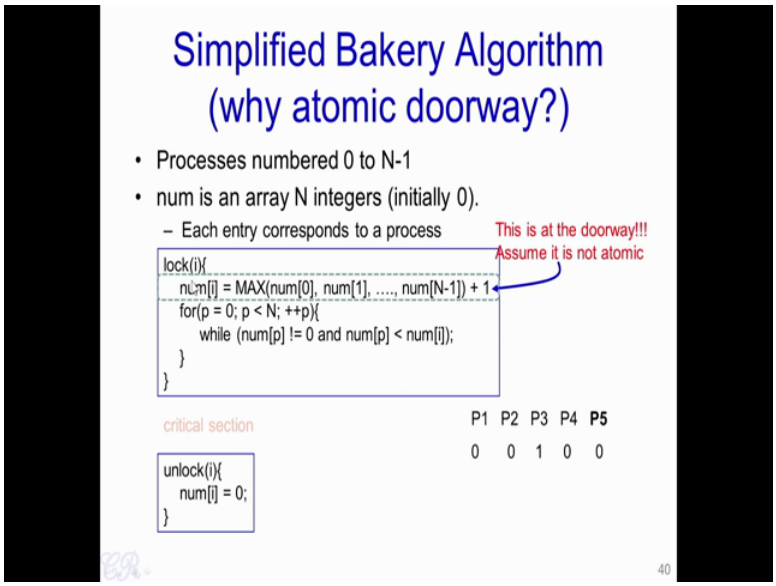
So, one requirement or one assumption that we made over here (blue box statement in above image) is that this particular assignment of MAX needs to be atomic; essentially this is required to ensure that no two processes get exactly the same token.

(Refer Slide Time: 08:23)



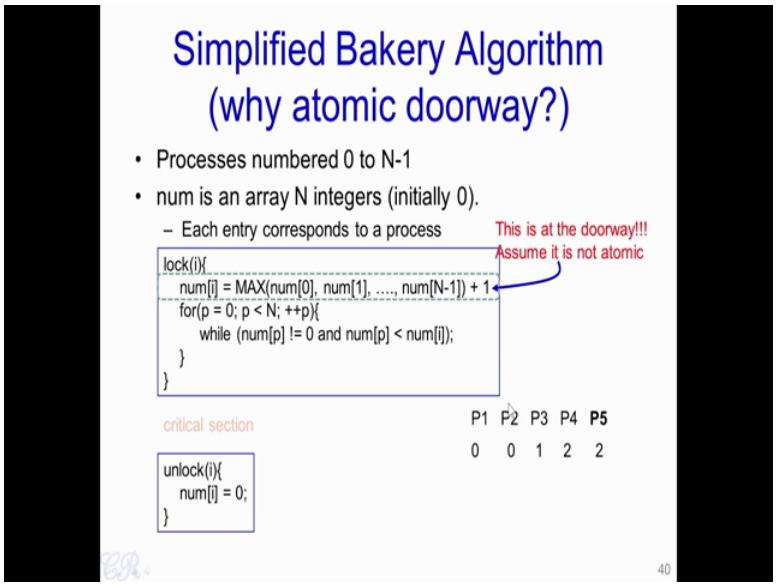
Essentially, it means that when a particular process is executing this particular statement (blue box statement in above image) that is find the MAX of all these numbers and adding 1 to it then no context switch can occur. This entire statement executes as a single entity. So, the reason why we make this particular assumption is that we need to ensure that no two processes get the same number. So let us see what would happen if we actually have two processes having the same number, essentially what would happen if this doorway (refer above slide image) or this statement which is known as the doorway is not atomic. So, we will take our example of the five processes and we will look with respect to this particular example.

(Refer Slide Time: 09:19)



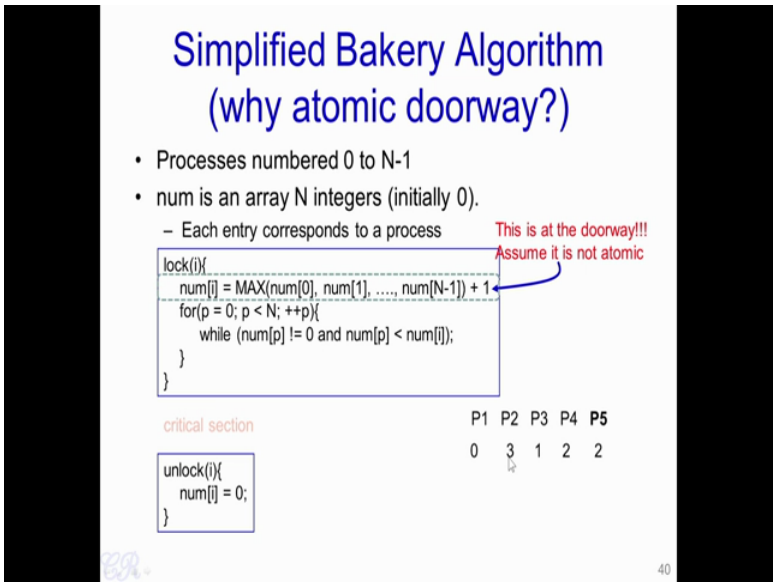
So, as usual let us say process P 3 invokes lock first, and it obtains a number 1 because is the smallest number all other numbers are 0.

(Refer Slide Time: 09:34)



Then let us assume that process P 4 and P 5 simultaneously execute MAX, resulting in both of them getting the value of 2 (refer above slide image).

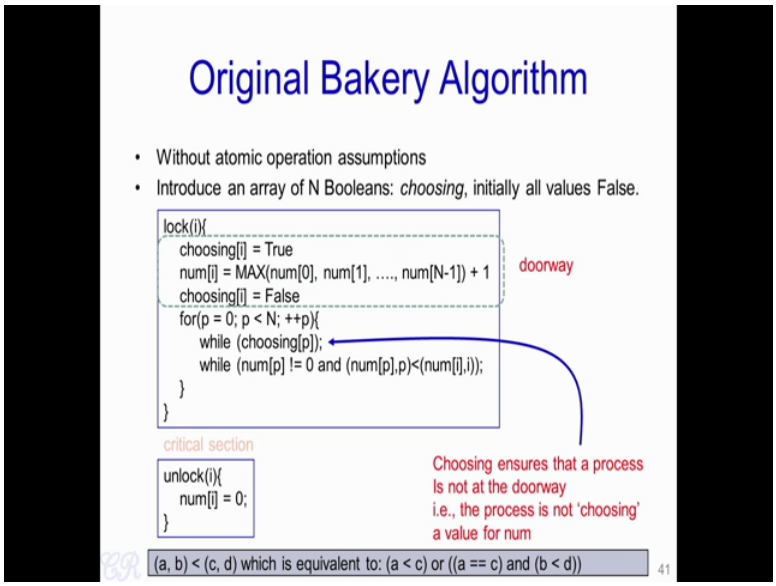
(Refer Slide Time: 09:45)



And then of course, we have the process p 2 which gets the value of 3. Now, what would happen in the second part of this lock (blue box statement)? So, we would see as usual process P 3 is going to execute first, because it has the lowest number. And once it exit from the critical section, P 3 is going to set its corresponding number to 0, therefore this number (1 corresponding to p3 as mentioned above) corresponding to P 3 is set to 0. Now next there are two small numbers corresponding to P 4 and P 5 which are equal (i.e 2). So, as a result of this, we have process P 4 as well as process P 5 which enter into the critical section simultaneously.

And thus we do not achieve the mutual exclusion, therefore it is required that this MAX operation is atomic. So, this will ensure that no two processes get the same value for num; and thereby it will ensure that the critical section is executed exclusively by a process at any given instant of time. Next what you are going to look at is the relaxation of this particular assumption, so we are going to look at what is known as the original bakery algorithm where we do not require to make this statement atomic.

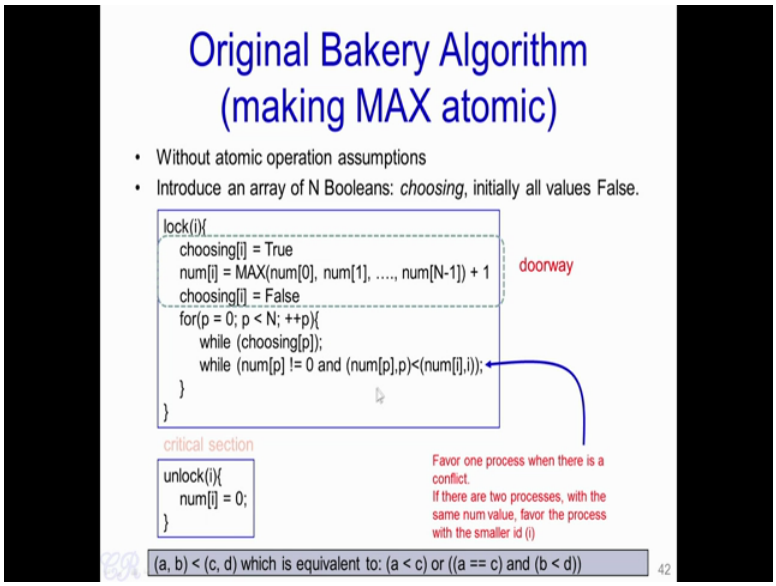
(Refer Slide Time: 11:05)



The original bakery algorithm is as follows (refer above slide image); in addition to the shared array num, which is present as before. We also have a shared array called choosing. So, this is a Boolean array and could have the value of true and false; and length of this array is N that is each process will have a particular element in the choosing. So, essentially this particular choosing is set to true before the process could invoke MAX and after this particular MAX function is invoked and 1 is added then the process would set its choosing value to false (3 statements in blue box mentioned in above image).

So, there are also some minor changes in the second part of the algorithm. First, we have a statement called while(choosing[p]);, which is present here (mentioned above as blue arrow). So, this particular statement would ensure that a process is not at the doorway that is it will ensure that the process is not currently being assigned a new number through this MAX, that is the process is not choosing a new value of num.

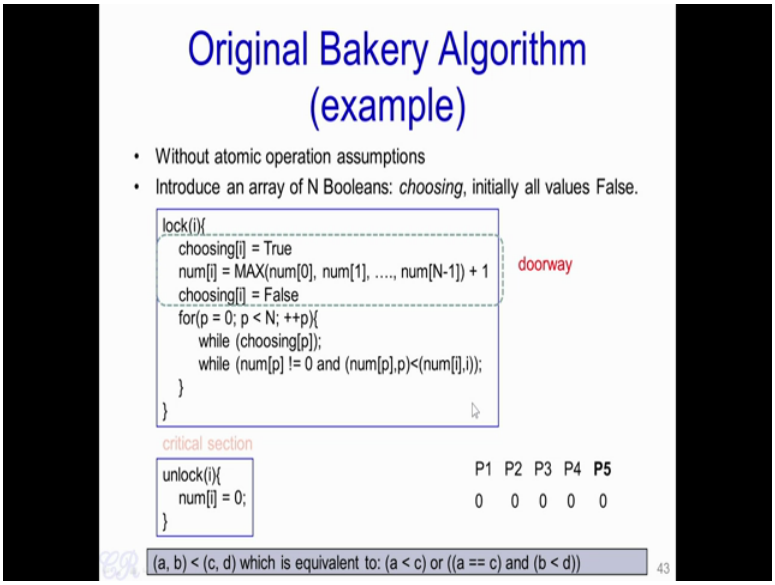
(Refer Slide Time: 12:26)



Secondly, we have the second part of the while loop which essentially changes over here in the condition check (blue arrow in mentioned in above slide), which we have a tuple num[p],p checked with less than num[i],i. So, what this condition checking means is written over here and it means the following (mentioned in above slide image). If (a, b) < (c, d) is the same as (a < c) or ((a == c) and (b < d)).

This particular complex looking check is used to break the condition when two processes have the same num value. So, as we have seen before if two processes are given the same value of the num then this condition (mentioned above) is used to going to resolve the issue, and ensure that only one of these two processes would enter into the critical section. When num[p] = num[i] that is both of the numbers have the same value we need to favour one of the processes. So in such a case, we favour the process with the smaller value of i.

(Refer Slide Time: 13:44)



So let us see this with the same example as we seen before. So let us look at this example again and let us say as usual process P 3 executes this MAX first and is given the smallest number, then P 4 and P 5 happen to get the same number of 2 and then process P 2 gets the next highest value of 3. Now, let us look at the second part of this locking (refer above slide image). So, the first process to execute in the critical section is quite obvious that is P 3, because it has the lowest number, so P 3 executes. And at the end, it will set the value of num[3]= 0. Now the next process to execute could be either P 4 or P 5.

So, how do we choose between these two processes? We have seen that both num[p] and num[i] are 2 in such a case, therefore, in order to favour one of the processes, we look at the second part that is p and i (only p and i mentioned in while condition), so, based on this we favour the process which has a lower number and therefore, process P 4 executes in the critical section. So, after P 4 executes its value is set to 0, and then quite naturally P 5 executes.

And after P 5 executes as usual P 2 will execute. Thus we see the addition of choosing the Boolean array over here as well as a more complex conditional check would help resolve the need for an atomic operation of MAX. So, this is the original bakery algorithm which was proposed by Leslie Lamport; and it efficiently helps to solve the critical section problem when the number of processes is greater than 2.

Thank you.

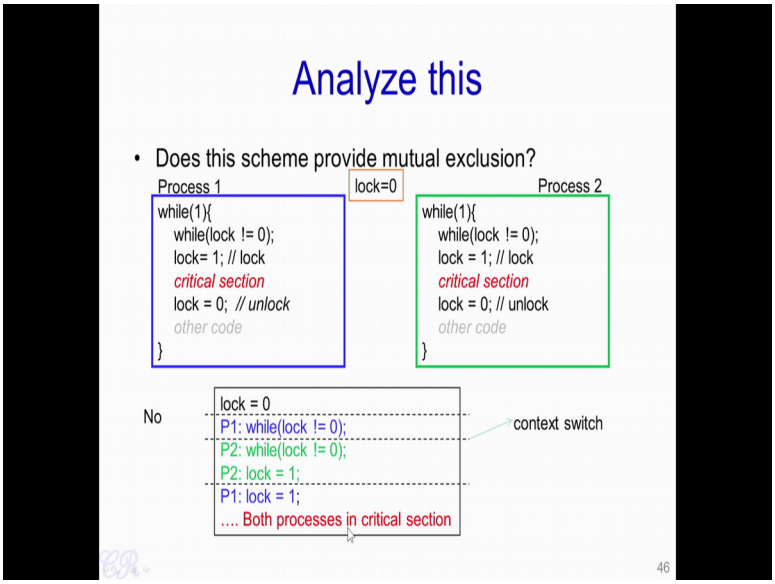
Introduction to Operating Systems
Prof. Chester Rebeiro
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Week – 06
Lecture – 27
How to implement Locking (Hardware Solutions and Usage)

Hello. In this particular video we will look at Hardware Solutions to Solve the Critical Section Problem.

So in the previous video we had looked at Software Solutions to Solve the Critical Section Problem. However, in practise these solutions are not very efficient. So in practise both in the operating system as well as in applications, hardware techniques are used to efficiently solve the critical section problem. So we will start this video with the small motivating example.

(Refer Slide Time: 00:52)



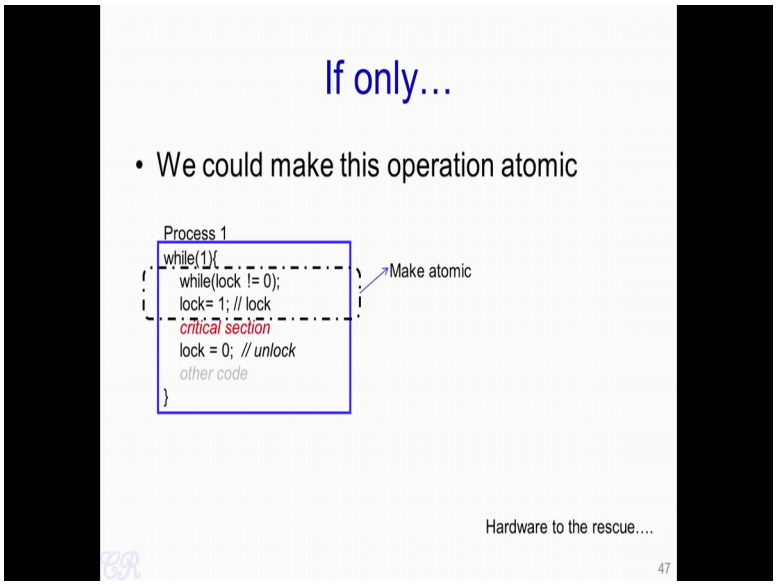
So let us start with this particular example (refer above slide image). Let us say we have the two processes; process 1 and process 2 and both are having a similar critical section that accesses the same shared data. And also let us say that we have this shared variable call lock (mentioned in above slide), so this variable is shared between process 1 and process 2. Now what each of these processes do is two things; first in order to lock the critical section, the process would first execute this while loop until lock becomes 0 (i.e

while(lock != 0)). When lock becomes 0 then the process would enter here (below while condition) and set lock to 1 and then execute the critical section. At the end of the critical section the process would set lock to 0 in order to unlock the section. The question that I would post over here is; does this particular scheme achieve mutual exclusion? The answer is no.

Essentially we have seen such things in the previous video as well. Due to context switching between the two processes we would reach a state some times when both processes execute in the critical section. Thus, we will not be able to achieve mutual exclusion. So, for instance if we actually look at this (refer above slide 3rd block of statements), we see that lock has an initial value of 0. Then suppose process P1 executes in the CPU, and it executes while(lock != 0) and since, we have lock as 0 over here (before while condition), so this particular while loop will break, however we have a context switch (2nd dotted line in 3rd block) that occurs before the process could set lock equal to 1.

Now, this context switch results in process P2 executing. Now P2 will also see the value of lock equal to 0, and therefore break from this (while condition of P2) while loop and then set lock to 1. Now again if there is a context switch occurring (3rd dotted line) and process P1 will continue to execute from where it stopped, it will see the old value of lock which is 0 because, the context is returned back to the old value of the process before the context switch occurred. And therefore, process P1 would see the value of lock as 0. And therefore, it would set lock equal to 1 (i.e. lock = 1) and enter into the critical section. Thus we have reached the state where both processes are in the critical section.

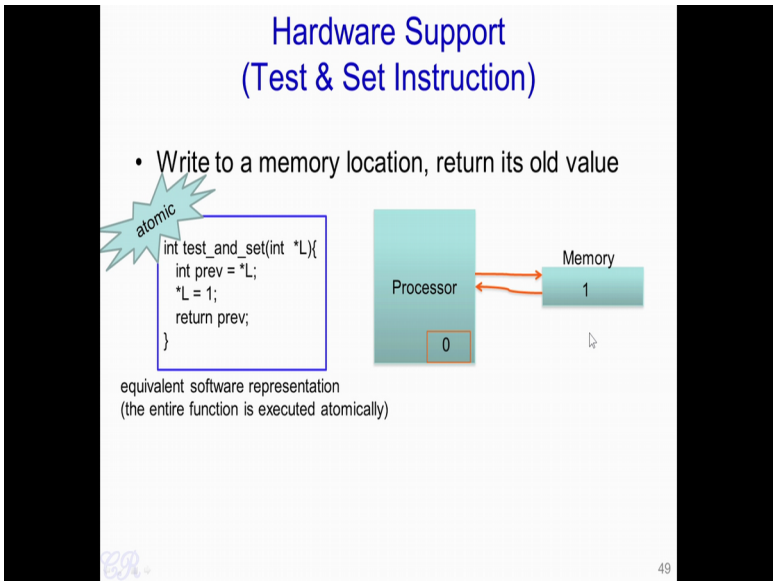
(Refer Slide Time: 03:42)



Now, the main reason that this particular scheme failed was that as we have written this (set of statements) in software, we were unable to make these two statements atomic. That is, we are unable to ensure that a context switches do not occur in between these two statements. Said another way, we are unable to make these two statements execute as a single unit. And thus, due to this reason we were unable to achieve mutual exclusion. Now most processors such as the x86 Intel processors have dedicated instructions that will ensure that these statements are executed atomically.

Let us take an example of such an atomic instructions through which we could implement this set of statements (refer above slide image) in an atomic manner. So we will take a very general instruction first and then analyse specifically for the Intel x86 type of processors.

(Refer Slide Time: 04:51)

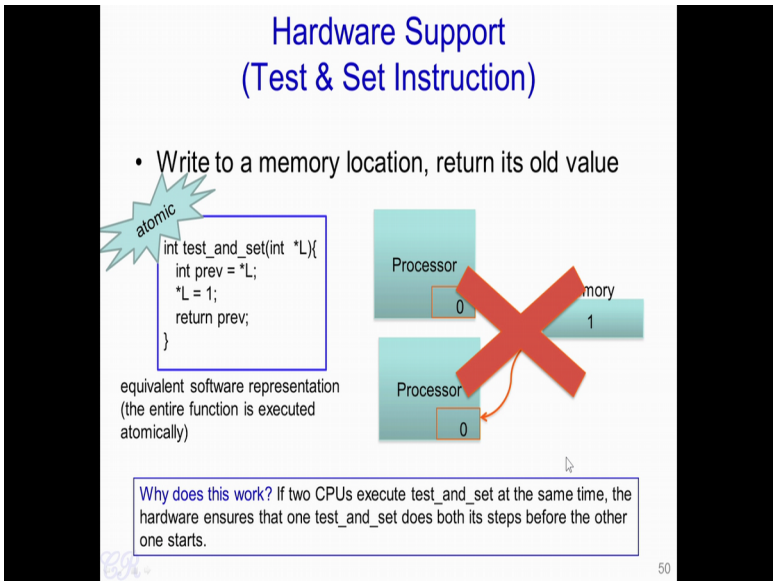


The first instruction that we would see today is the Test and Set Instruction. Essentially if we were to write this instruction in C, it would look something like this (mentioned in above image). It is a function which takes a pointer to a memory location. So in this function (i.e test_and_set) first the contents of that memory location is stored into this variable called previous i.e prev, then that memory is set to 1 (i.e *L = 1) and the value returned is the previous value. In other words this function would return the previous contents of the memory and set that memory to a value of 1.

Now, when we look at this (above mentioned function) from a hardware perspective and essentially from the processor perspective, this entire function is an atomic function. Essentially we would have one instruction that would do all of this thing in one shot that is, we would have one instruction that would perform all of these operations atomically. So let us look at this particular diagram to demonstrate how this thing actually works.

So let us say that we have the processor and the memory location pointed to by L is over here. So, what would happen when this test and set gets executed, is the following (mentioned in above image slide); first the previous contents of the memory get loaded into a register present in the processor and then the memory contents is set to 1. So let us look at it another time; first the memory contents pointed to by L is loaded into a register present in the processor and then the contents of that memory location is set to 1.

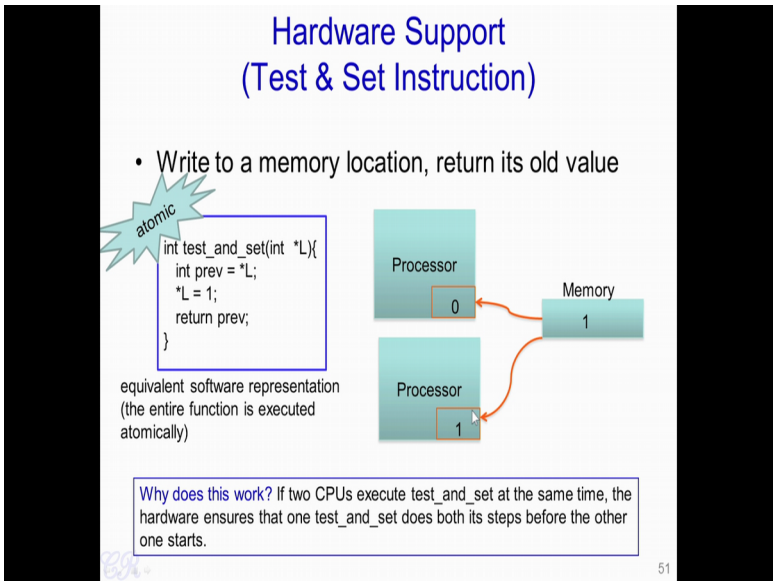
(Refer Slide Time: 06:46)



Now, the atomicity of the test and set instruction can be defined or explained as this statement here (mentioned in above image in box) which stats that, if two CPU's execute test and set at exactly the same time, the hardware or that is a processor ensures that one test and set for one of these processors does both its operations. That is, reading contents of the memory to a register and setting that particular memory location, so both these steps are done before the other process starts executing.

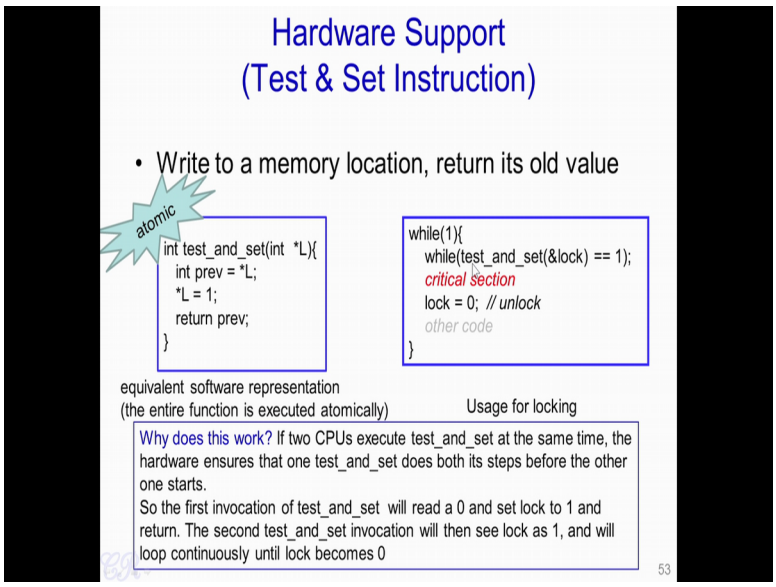
In other words, it is not possible for two processes to execute the test and set instruction and both processes read the previous value of the memory location. So, essentially this thing is wrong (mentioned in above imsge), is not correct. That is it is not possible for both processes to simultaneously read the contents of the memory location (i.e 0 mentioned above) and set the value of memory (i.e setting 1).

(Refer Slide Time: 07:56)



However, what actually is guaranteed by the processor hardware is that when processes run on each processor and both processes execute the test and set instruction at exactly the same time, the hardware will ensure that one process completes its entire instruction before the next process could execute its instruction. Therefore, in such a case one process would read the value of 0 while the other process would read the value of 1 (mentioned in above image). Now we will see how this particular instruction is used to solve the critical section problem.

(Refer Slide Time: 08:30)

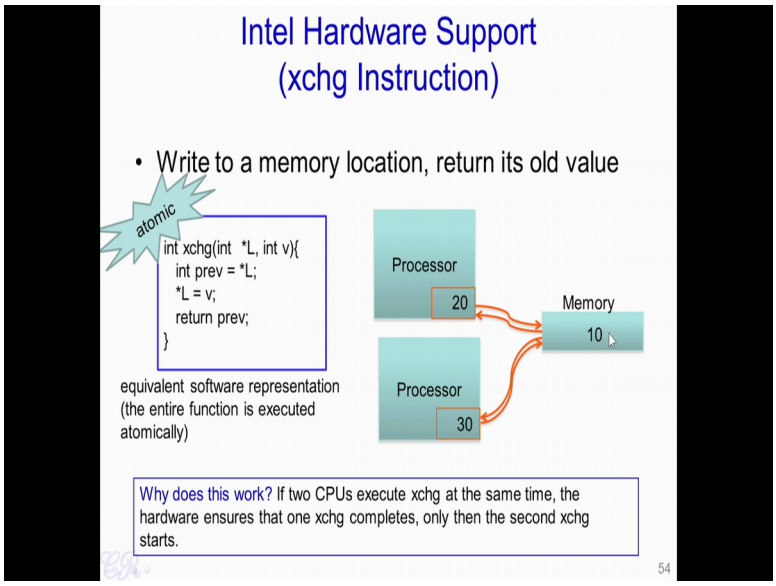


So this particular snippet over here (while condition mentioned in above image) shows how a process could use the test and set atomic instruction which hardware supports in order to solve the critical section problem. Essentially, in order to lock the critical section we would have a while loop present over here (second while condition in above snippet) which would invoke the test and set with a lock, that is with the memory location (&lock). Now, this while loop will execute continuously until the value returned by the test and set is 0; So when the value returned by test and set instruction is 0 then critical section is entered and at the end of this the value of lock is set to 0.

When there are multiple processes having such a code snippet, the hardware guarantees that the test and set for exactly one process would return a value of 0. All other processes will loop continuously in this particular while loop. So when this process has completed executing the critical section, the value of lock would be set to 0. So, setting the value of lock could to 0 would result in exactly one other process to obtain a value of 0 from the return of the test and set. Therefore, exactly one other process would then enter into the critical section.

So the first invocation of the test and set will read a 0 and set the lock to 1 and return. The second test and set innovation will then see the lock value as 1 and will loop continuously until the lock become 0. The value of the lock becomes 0 only when the first process unlocks it that is exclusively sets the value of lock to 0. So in this way we see that by a little help from the hardware it is feasible to solve critical section problem very efficiently.

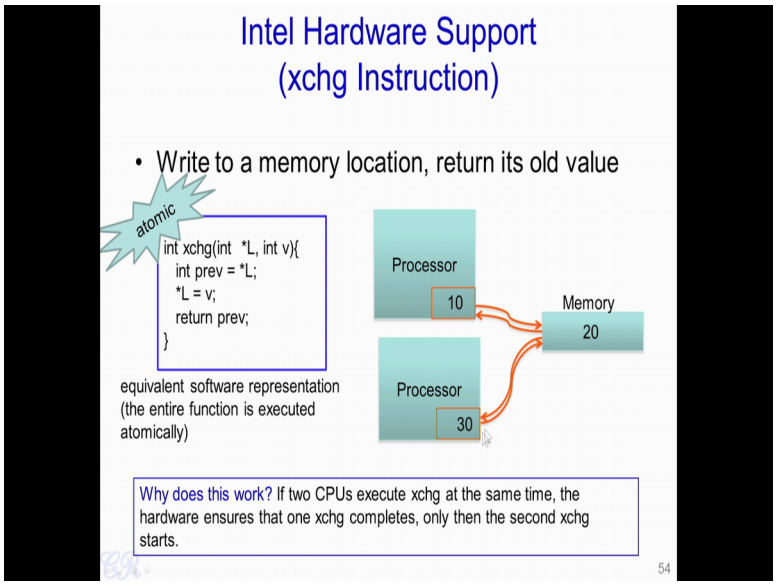
(Refer Slide Time: 10:41)



So, Intel systems do not support the test and set instruction. On the other hand it supports an instruction known as exchange; xchg. So an xchg instruction is represent over here and this is also an atomic instruction and the xchg instruction takes two parameters, so it takes a memory address and an integer value. Now what is done is that the contents of that memory location is stored in previous, then the value of v that is the value which is passed to xchg is then stored into that memory location and the previous value is returned (mentioned in above slide image).

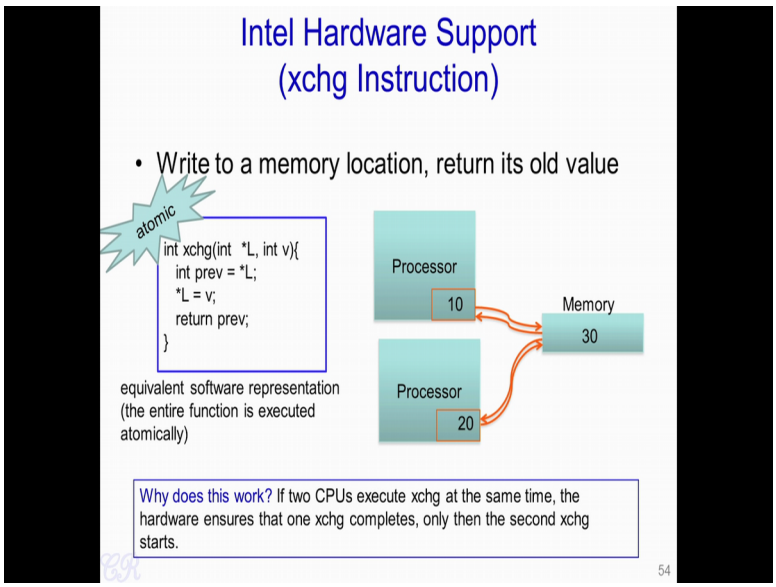
So, essentially what this achieves is that we are able to xchg register variable (i.e int v) with a memory location. To take an example, so look at this (mentioned in above image) when once an xchg instruction is executed by a process running in this processor, it will exchange a register value with that of a memory location and this entire thing is done atomically.

(Refer Slide Time: 11:49)

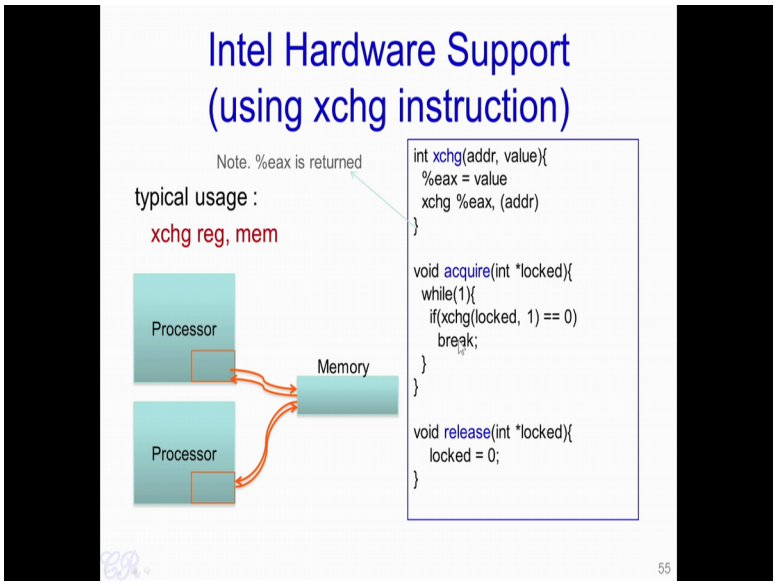


So even if another process executes the xchg at exactly the same time, the Intel hardware ensures that these two operations are done distinctively. So, first one process would execute and complete the xchg instruction and only then the second process would execute and exchange the data.

(Refer Slide Time: 12:14)



(Refer Slide Time: 12:14)

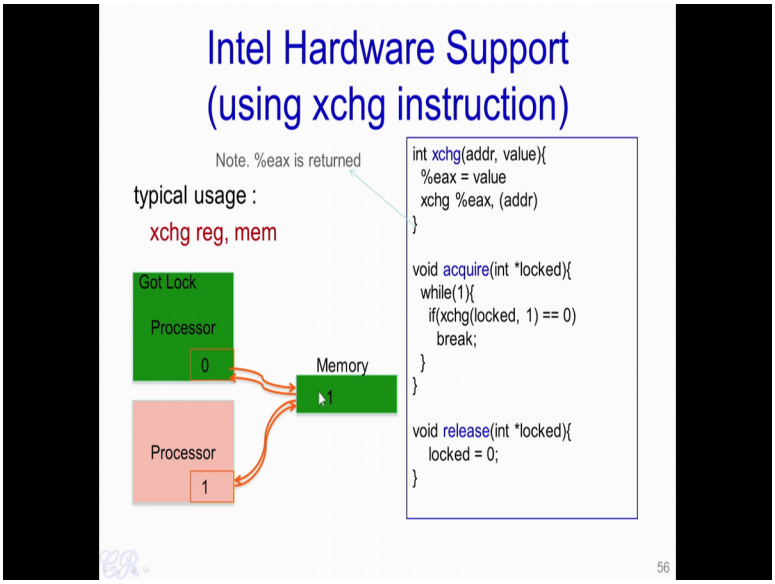


So this particular slide (mentioned above) shows how an xchg instruction in an Intel hardware is used to solve the critical section problem. We could have two primitive functions called Acquire and Release. So the acquire is used to lock a critical section, while a release is used to unlock a critical section. So this, acquire and release functions are passed a pointer to a memory location known as Locked. So in acquire in an infinite rather in acquire in a loop, this function xchg locked is invoked. So, xchg is present over here (function mentioned in above image) and it is passed two parameters; the address of the memory location in this case the address of locked and the value which you want to set (i.e xchg(addr, value)). So first the value in this case one is moved into the eax register in the Intel processor and then the xchg instruction is invoked.

As a result the memory or the data which was stored in memory gets loaded into the eax register. And the value in this case one which was loaded into the eax register gets stored into memory. Now this particular eax register (mentioned as note in above image slide) is what is returned back to acquire. This loop (loop mentioned inside acquire function) would break if the value returned by xchg that is the contents of the eax register is 0. In order to release the lock, we simply set the locked value to 0 (mentioned in release function).

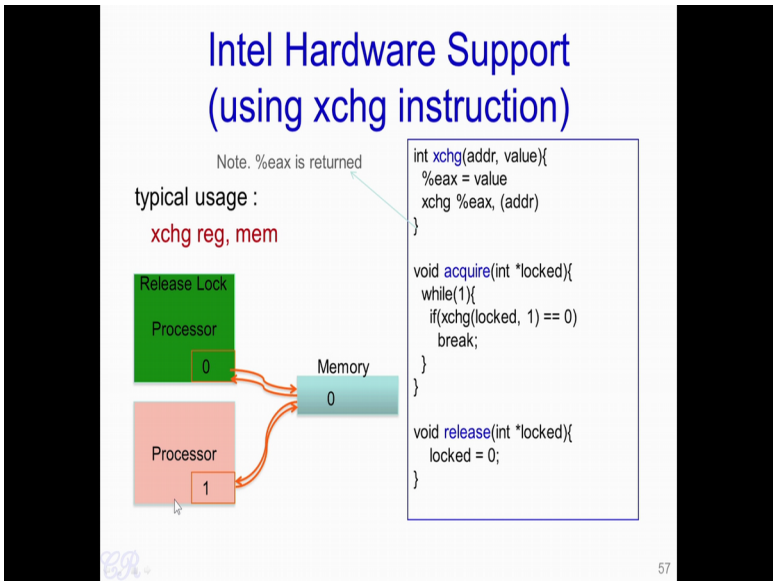
Let us see how it works. So first, the memory location over here would be 0 and when the process invokes xchg instruction that 0 value is pushed into the eax register, and the value of 1 which was present in the eax register comes into the memory.

(Refer Slide Time: 14:29)



As a result, the memory has a value of 1 while the eax register present in the processor has a value of 0 (mentioned in above slide image) and this is what is used to break from this while loop and enter into the critical section. Now in order to release the lock, we have simply setting the value of locked to 0. Now when a second process invokes acquire, it will continue to loop in this while loop until the value of this memory location is set to 0 by the first processor.

(Refer Slide Time: 15:08)

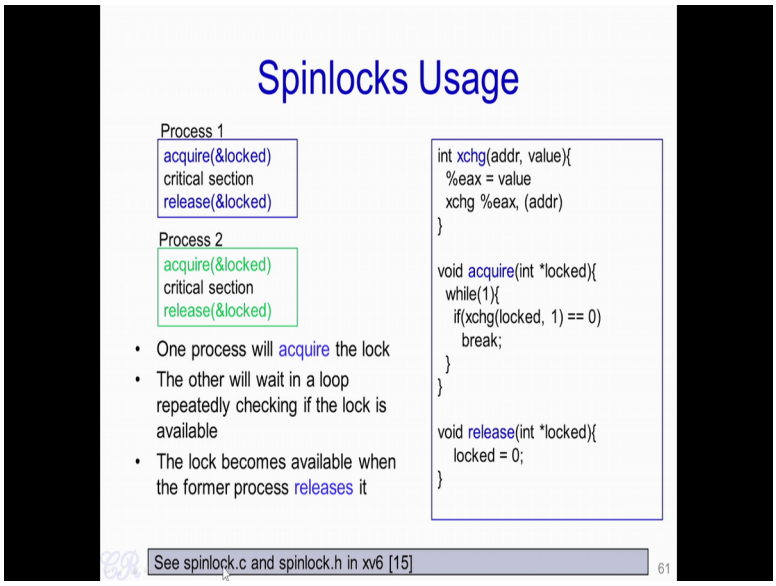


We see over here (as mentioned above), so in order to release lock we simply set the value in memory to 0 and then in another processor perhaps would be able to obtain the value of 0 and then acquire the critical section.

(Refer Slide Time: 15:23)

So this instruction like the xchg is used to build higher level constructs which are used in various critical section problems. So, we will look at Spinlock, and Mutex in this video and Semaphore in a later video.

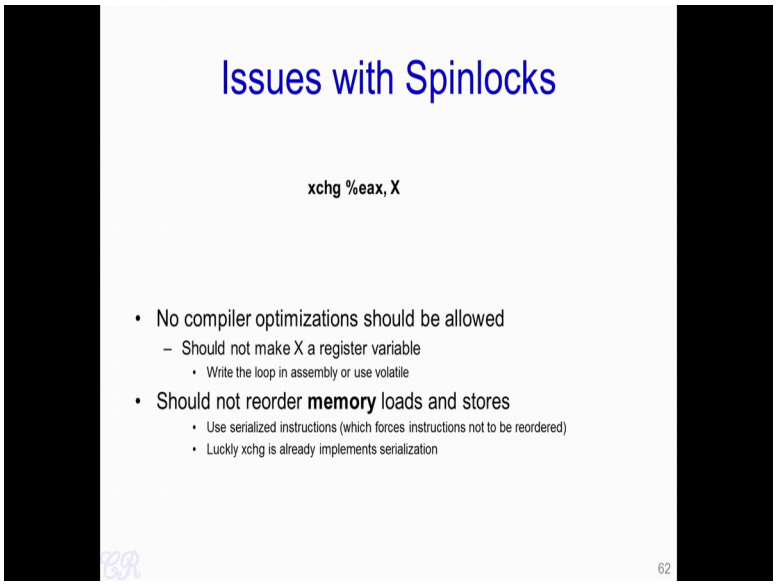
(Refer Slide Time: 15:39)



So Spinlock is what we have seen before, it has two functions acquire and release. While the acquire is used to gain access to the critical section essentially it is used to lock the critical section, while the release function is used to unlock the critical section. So in order to use this we could have two processes; process 1 and process 2. In order to enter into the critical section the process has to acquire the lock and this function acquire will continuously loop until the value return by xchg is 0. So when the value of xchg is 0 then this loop breaks and the process 1 would enter into the critical section. Releasing the lock is just done by setting the value of locked to 0.

So, one process will acquire the lock at a given time while the other process will wait in this particular loop and continuously invoke the xchg function until it obtains 0. So the first process in order to release the lock would have set the value to 0, this would cause the second process to break from this loop (mentioned above in acquire function) and enter into the critical section. So you could see more details about the way spinlocks are implemented in xv6 by looking into these 2 files that is spinlock.c and spinlock.h (mentioned above).

(Refer Slide Time: 17:13)

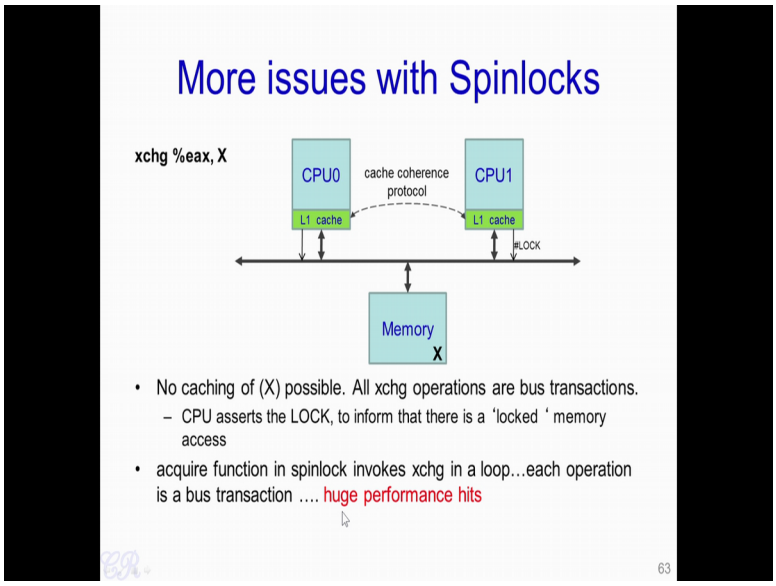


We will now look at some of the issues that go about with the use of the xchg instruction. So as we have seen the xchg instruction is the most crucial part of any of the constructs such as the Spinlocks that we have seen in the previous slide. So therefore, it's important to understand what are the issues related to the xchg instruction. Essentially this particular format of the xchg instruction (i.e xchg %eax, X), exchanges data between a register eax and a memory location X. Now it should be ensured that there are no complier optimization that are done on this variable X.

So, some of the common optimization that are possible is to make the value of X stored in a register. In such a case, we are simply exchanging data between a register eax and an other register which will not solve the purpose. Therefore, we should write this particular loop in assembly or use the keyword volatile.

Another requirement while implementing the xchg instruction is to ensure that memory operations are not reordered. The CPU should not reorder memory loads and stores. So in order to achieve this we use serialized instructions, which force instruction not to be reordered. So luckily for us the xchg instruction by default already implements serialization. So there is nothing much we need to take care about this.

(Refer Slide Time: 18:46)

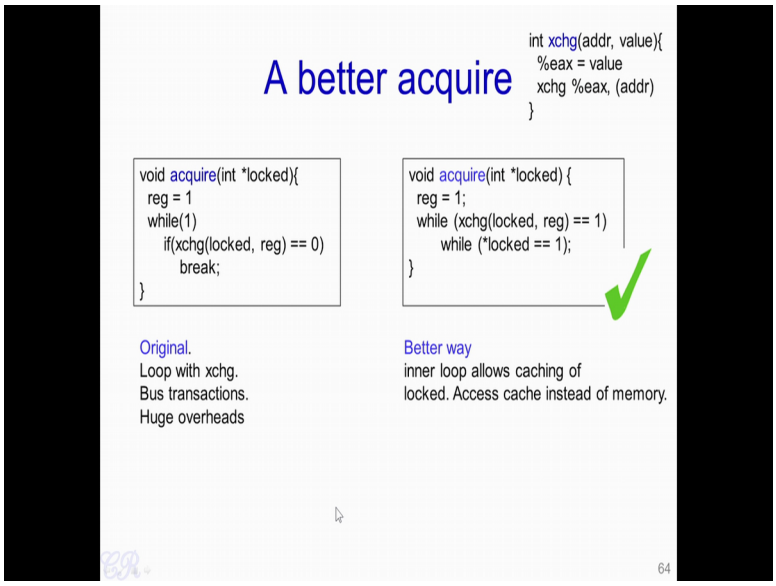


However, what we need to look at more closely is the fact of cache memories. Now, recollect that xchg instruction exchanges data between a register and a memory location. Now each CPU present in the system could have their own private cache, for instance CPU 0 has its own private L 1 cache; similarly CPU 1 has its own L 1 cache (mentioned in above slide image). So it should be ensured that this value of X (mentioned in xchg instruction) is not a cached in each of these CPU's, essentially caching of CPUs is not possible. But rather, each and every execution of this xchg instruction should actually go to memory and load or store the value of X.

Secondly it should also ensure that when one CPU is reading and writing to this X (mentioned above in memory box), that is in other words when one CPU is invoking the xchg instruction during this time no other CPU can modify the value of X, as this will break the atomic nature that is present. In order to do this the CPU asserts what is known as a lock line to inform all other CPU's in the system that there is locked memory access that is going to take place.

So as a result of this lock instruction, all xchg instruction which read and write data would result in the reading and the writing data from a memory. So, this may be tremendous amounts of performance hits.

(Refer Slide Time: 20:30)



Therefore, in order to make a better acquire, an acquire which is more efficient we will look at a small tweak to this particular acquire function (mentioned in above slide). So originally, we use to have a loop over here (mentioned in above in original side) and in this loop we would continuously keep invoking the xchg function and checking whether it returns a value of 0.

So note that, each of these xchg instructions has a huge performance overhead because it requires that the locked keyword that is the X value would go all the way to the memory and read and write data from the memory. Essentially, the caching is not possible and therefore huge overheads.

On the other hand if we make a minor change to this acquire function as shown over here (mentioned above in better way side) it would improve performance quite significantly. So we have two loops one is the regular xchg loop which as you know would result in a bus transaction and has huge overheads, while does an inner loop which simply loops checking the value of the memory location locked.

So this particular internal thing (i.e second while condition mentioned above) can is cacheable and therefore, will not incur much performance overhead, while, the external while loop (i.e first while condition) will have significant overheads, but it's not invoke too often. Most of the time this particular cached memory (i.e second while condition mentioned above) would be read.

Now the cache coherency protocol will ensure that when another process changes the value of locked, this process would see the value of locked changing from 1 to 0 and it will exit the while loop (inner loop) and go back to this particular outside while loop. The xchg function would then exchange the locked value with the register value and set the value to 1, and therefore break from the while loop.

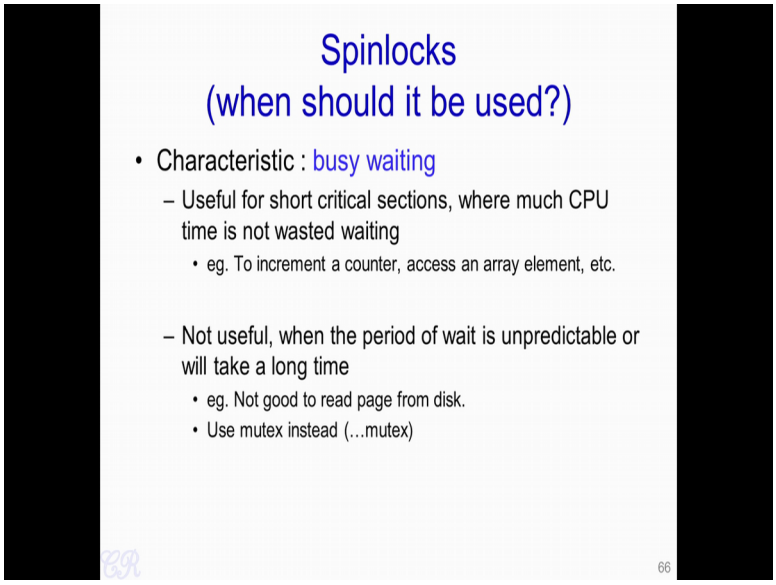
Thank you.

Introduction to Operating Systems
Prof. Chester Rebeiro
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Week - 06
Lecture – 28
Mutexes

Hello. In this video we will look at a Mutexes, which is a construct used to solve the critical section problem.

(Refer Slide Time: 00:25)



So we will start with where we stopped off in the last video, with Spinlocks. Essentially, the main characteristic of spinlocks is that it uses busy waiting. That is, we had seen that in order to have a lock, there was a while loop and in that while loop the xchg instruction was continuously invoked and the while loop would only exit when the xchg instruction returned a 0.

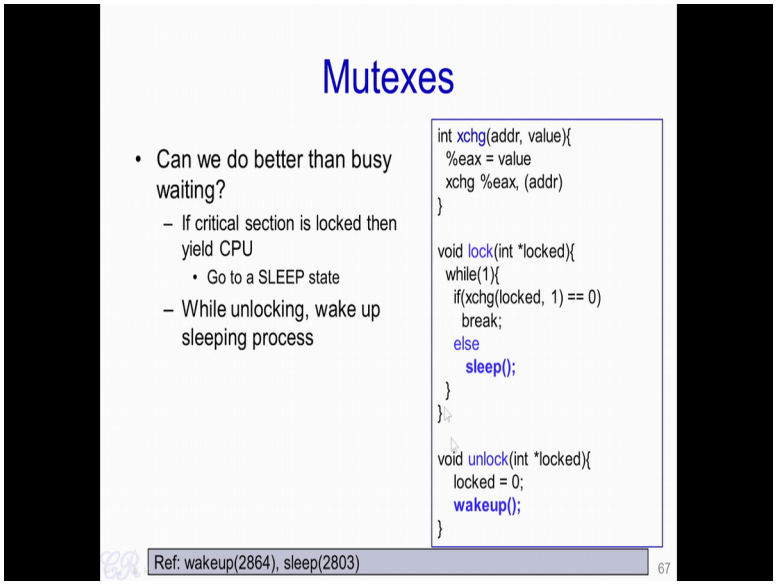
So this busy waiting is not ideally what is required. Essentially, busy waiting causes the CPU cycle to be wasted, leading to may be things like performance degradation as well as huge wastage of memory.

So, where would we actually use Spinlocks? Spinlocks are useful when we have short critical sections and we know that we do not have to waste too much time in waiting. So

for instance, if we just want to increment a shared counter then we could possibly use a spinlock or another thing is to access an array element then a spinlock would be preferred. Essentially, these things we assume would will not have too much of overheads. Therefore, even if another process is accessing the counter we are certain that the process is going to spend too much time incrementing the counter.

Therefore, the waiting process will not have to waste too many cycles waiting to enter into the critical section. However, spinlocks are not useful when the period of waiting is unpredictable or it will take a very long time. For instance, if there is a page fault and resulting in a page of memory which is loaded from the hard disk into the main memory. So, this would take a considerable very long time and you do not want your process to be actually wasting CPU cycles during this entire operation. In such a case, we use a different construct called a Mutex.

(Refer Slide Time: 02:43)



This over here (mentioned in above slide image) shows how typically the mutex is implemented. So, it again relies considerably on the xchg instruction which is used and just like the spinlock, we have the lock and unlock and a memory location which is shared between all processes.

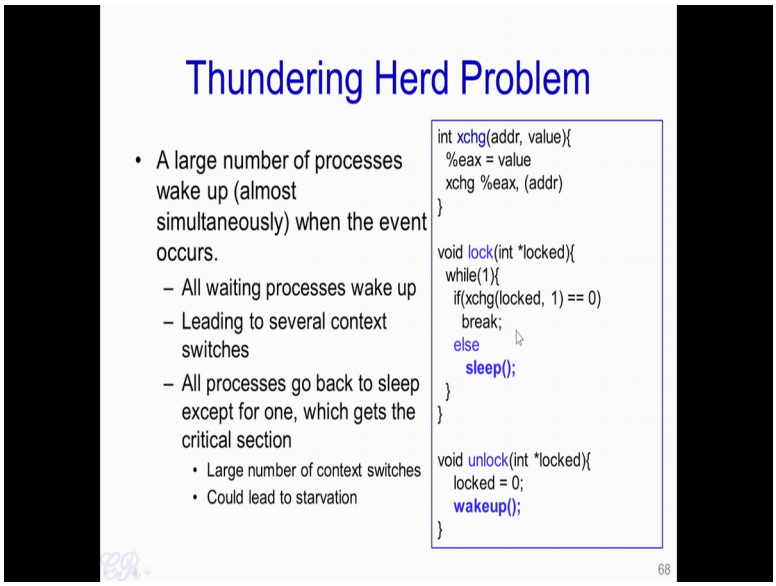
Now, in order to obtain the critical section, a process would need to invoke a lock and in this lock function we have a while loop (as mentioned above). So, as in this spinlock case the xchg instruction is invoked in the while loop and this instruction would either

return a value of 0 or something not equal to 0 or typically 1. So if the value is equal to 0, then we break from this loop and that process would then have acquire the lock and execute in the critical section. However, if the xchg returns a value which is not 0 then we go into this else part and execute this function called sleep() (mentioned in lock function in above image).

Now, this sleep function would cause the process to go from the running state into the blocked state. Essentially, the process is waiting for a particular operation to arrive. So, until this operation arrives the process will not get any CPU time. Now, this event which the sleep is waiting for is the wake up event. So, when other process invokes wake up, it will result in the sleeping process to be woken up from the blocked state and put on to the ready queue. Now, if it is lucky when it executes the xchg instruction again it would get 0 and it would get into the critical section.

On the other hand, if it is unlucky it would execute the xchg instruction and get something which is non-zero and it would go back to sleep. And it would continue to sleep until woken up by another process. So essentially, we see over here (in mutex lock function) that instead of doing a busy waiting as was done in spinlocks, in mutexes we put the entire process into a sleep state. The process will continue to be in a sleep state until it is woken up and when it is woken up, it is going to try the lock again and if it achieves a lock then it enters into the critical section.

(Refer Slide Time: 05:34)



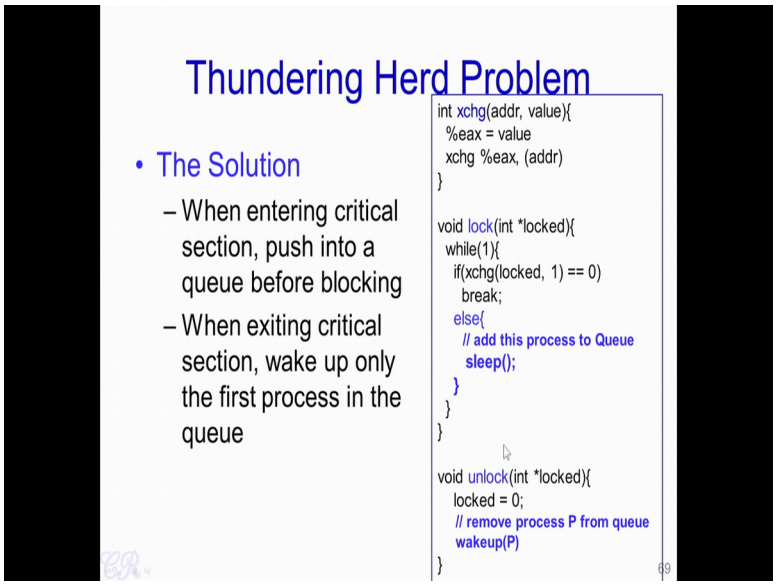
So one issue with mutex is what is known as a Thundering Herd Problem. So, the thundering herd problem occurs when we have large number of processes. So, each of these processes let us assume is using the same critical section and invokes the lock in order to enter into the critical section, and at the end of the critical section would invoke unlock which would then wake up another process waiting for the critical section. So, it could happen if we have large number of processes that, there are several processes present in the sleep mode while one process enters into the critical section.

So, when that process executes unlock it invokes wake up. This results in all the processes which are sleeping to be woken up. So, all these processes would then go from the blocked state into the ready queue and the scheduler would then sequentially execute each of these processes. So, each process is then going to continue its while loop and executes the xchg function. So out of all these processes because of the atomic nature of the xchg instruction, only one process would acquire the lock and all other processes would go back into sleep state. So, this continues every time.

So every time, whenever an unlock is invoked by a process just completing its critical section, it will invoke wake up (i.e wakeup() inside unlock() function) and it will result in all processes waiting on that mutex to be woken up and all, except one would go back to sleep. There would be exactly one process which would gain the lock and enter into the critical section. So, as a result of this, what we see that whenever there is a wake up invoked there would result in several context switches occurring, in order that all the processes execute and check the xchg instruction again. Now, by the way exchanges implemented in the hardware, all processes except one will enter into the critical section.

So, the issue over here and why it is called a Thundering Herd Problem is every time there is a wake up, there is a huge avalanche of context switching that occurs because a large number of processes are entering into the ready queue and this could lead to starvation.

(Refer Slide Time: 08:19)

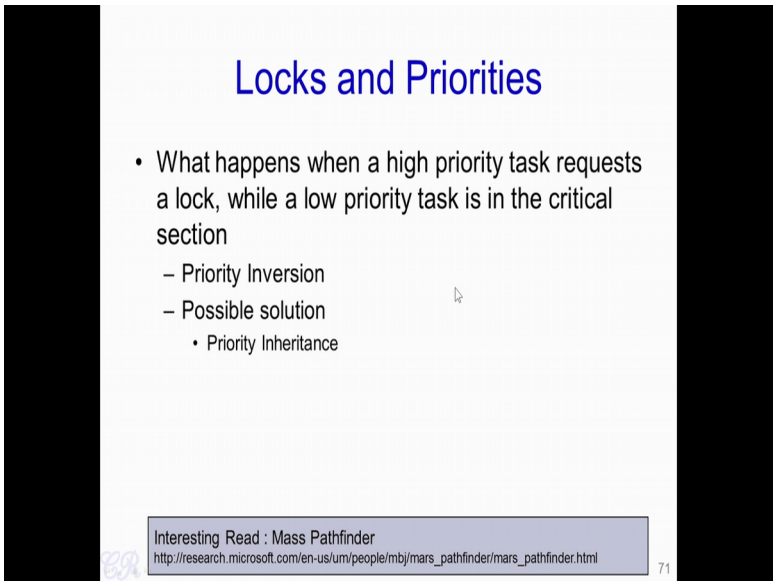


One solution to the Thundering Herd Problem is to modify the way mutexes are implemented by incorporating queues. In this implementation of a mutex, whenever the lock is invoked and the xchg instruction return something which is non-zero, the process gets added into a queue and then goes to sleep.

Now, when a process invokes unlock, the sleeping process is removed from the queue and a wake up specifically for only that process is invoked. So, unlike the previous cases where all processes are woken up, in this case there is exactly one process which is woken up, so this process P which is specified here (in unlock() i.e wakeup(P)) would wake up from sleep and since it is the only process which has woken up, it would typically go into this while loop, check the exchange and most likely it would get the lock and execute the critical section.

Similarly, when it unlocks, it would pick out the next process which is waiting in the queue and it will wake up only that process. Now, this second process would then enter into the critical section.

(Refer Slide Time: 09:42)



So, when we are talking about synchronization primitive such as a spinlocks and mutexes, it is important to also consider the case when a priority based scheduling algorithm is used in the operating system. So let us consider this particular scenario.

So, let us say we have a high priority task and a low priority task which share the same data and have a critical section. Now, let us say that the low priority task is executing in the critical section and at this particular time, the high priority task request for the lock in order to enter into the critical section. So, the scenario we are facing here is that the low priority task is executing in the critical section, while during this time the high priority task invokes something like a lock and wants to enter into the critical section.

Now, the dilemma we are facing here is that we have a high priority task which is waiting for a low priority task to complete. So, this is known as the Priority Inversion Problem. Essentially, we have something important - a task which is important and given a high priority and it is waiting for a lower priority task to complete its execution. And if you look at this particular link present here (mentioned in above slide image), you will see quite an interesting case where such a priority inversion problem had occurred, essentially with a path finder.

So, one possible solution for the priority inversion problem is known as the Priority Inheritance. So, essentially in this solution whenever a low priority task is executing in the critical section a high priority task request for that critical section, what happens is

that the low priority task is escalated to a high priority. Essentially, the priority of the low priority task becomes equal to that of the high priority task. The low priority task then would execute with this high priority until it releases the critical section. So, this would ensure that the high priority task would execute relatively quickly.

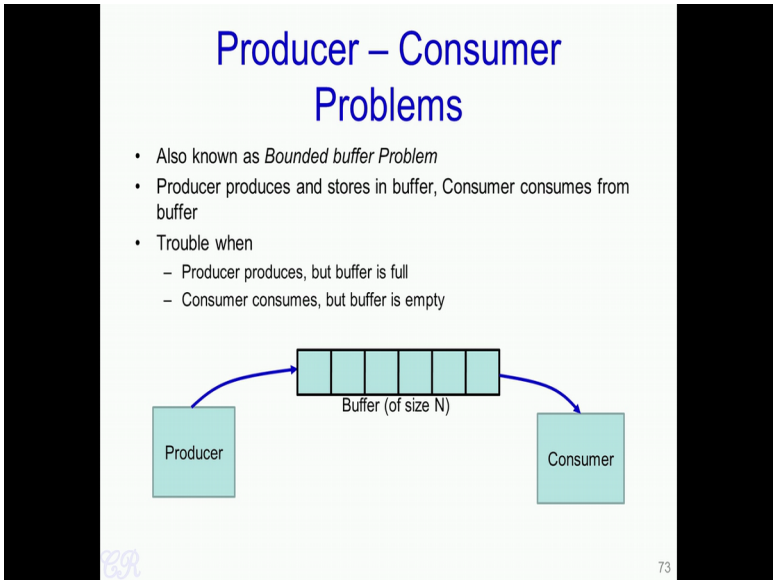
Thank you.

Introduction to Operating Systems
Prof. Chester Rebeiro
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Week – 06
Lecture – 29
Semaphores

Hello. In this video, we will look at another synchronization primitive known as Semaphores. As usual we will start with the motivating example and then we will show the Application of Semaphores.

(Refer Slide Time: 00:35)



So let us start with the very popular example known as the Producer-Consumer Problem. So this is also known as the Bounded buffer Problem. Essentially, what we have here are two processes; one process is known as the Producer and the other process is known as the Consumer. Now the producer and the consumer share a bounded buffer. So this is a normal buffer and it has a size of N that is it has N data elements which can be stored in it.

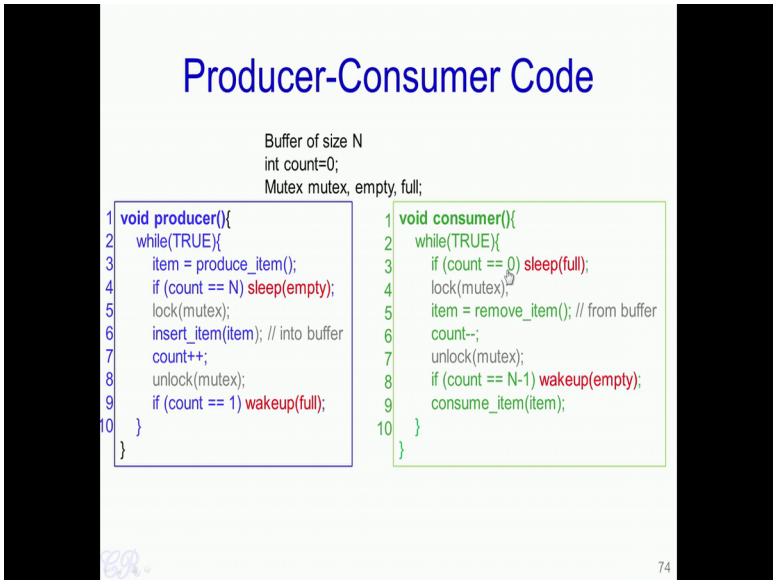
So in this particular case for example, there are 6 elements that can be stored in the buffer. Now the producer produces data. So for instance, it could be a data acquisition

module which collects data such as the temperature, pressure and so on, so this data is pushed into the buffer. Now on the other side, the consumer takes from the buffer and then processes the data. So for example, this consumer process could perhaps compute some analytics on the producer data.

So, everything would work quite well, that is the producer produces data, puts it on the buffer and on the other side the consumer takes from the buffer and begins to consume the data (mentioned in above slide image). For instance computes something with the data. The trouble will occur when for instance the consumer is very slow compare to the producer. In such a case the producer will produce quite a bit of data at a faster rate compare to the consumer and therefore very quickly the buffer will be full.

So, what does the producer do next? An other problem could occur where the consumer is very fast compared to the producer. And therefore, it could very quickly consume all the data in the buffer, and resulting in a buffer which is empty. So, what should the consumer do next? So this requires a synchronization mechanism between the producer and the consumer. Essentially, when the buffer is empty the consumer should wait until the producer fills in data into the buffer. Similarly, when the buffer is full the producer has to wait until the consumer takes out data from the buffer.

(Refer Slide Time: 03:25)



So this is the general producer and consumer code (mentioned in above slide), and we are trying to solve the producer-consumer problem by using Mutexes. Essentially, we are using 3 mutexes; empty, full, and a Mutex just called a mutex (as mentioned above).

So, the producer code (mentioned in above slide image) essentially would produce an item, insert the item into the buffer and increment the count, count++. While, the consumer code would remove the item, decrement a count and then consume the item. Now in order to take care of the troubled situation, that is when the buffer is full or the buffer is empty we are using these mutexes; empty and full. Essentially, before inserting the item the producer would check if count == N that is its going to check if the buffer is full. And if the buffer is full, it is going to sleep on this particular mutex called Empty (i.e sleep(empy)).

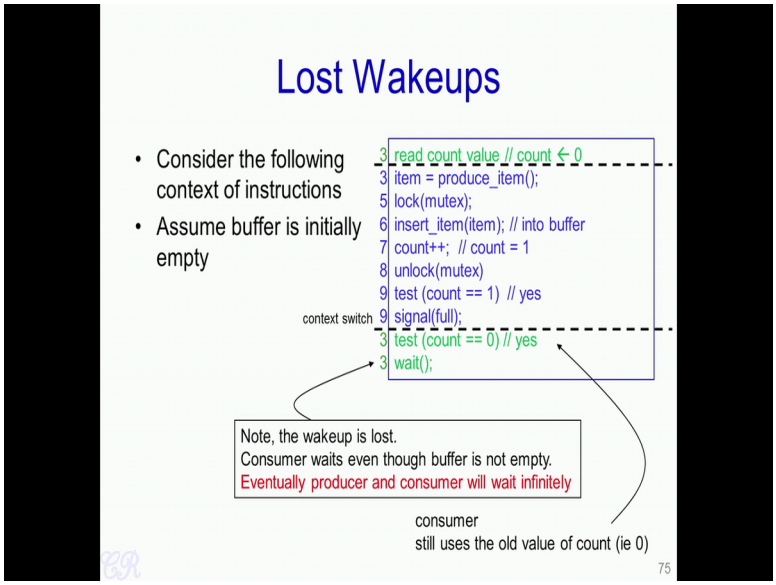
And on the other side, the consumer would check if count == N - 1 which means that it has just removed one element from a full buffer. So if this is so then it is going to wakeup empty (i.e wakeup(empty)). This wakeup is a signal to the producer to wake up from its sleep and then the producer can insert the item into the buffer and increment the count. On the other hand, if the consumer finds that the buffer is empty, that is the count equal to 0 (if count == 0) then it is going to sleep on this particular mutex called full (i.e sleep(full)). So, it will block on this (sleep) mutex until it gets a wakeup from the producer.

Essentially, if the producer finds that when he inserted the item and incremented count that there is exactly 1 item present in the buffer then he will send a wakeup signal to the consumer, so wakeup(full). And therefore, this wakeup(full) will cause the consumer to unblock and put it back into the ready queue and it would allow the consumer to execute, and remove that item and then consume that item. So after he removes the item the counts goes back to 0. Now in addition to this empty and full mutexes, there is also the third mutex which is used. So, this mutex is essentially used to protect or synchronize access to the buffer. So, before inserting an item and incrementing the count, the producer needs to lock the mutex and unlocking is done after the item is pushed and count incremented (as mentioned in above slide).

On the other side, before the buffer is accessed to remove item and also count is decrementing. Essentially you notice that count and the buffer is shared among these two processes that is the producer and the consumer. And therefore, this mutex will help synchronize access to the buffer and to the count value, so this solution seems to be work fine that is with the 3 mutexes. So, while this scheme seems fine we will show that under a certain condition the producer and the consumer will block infinitely without any progress.

So that condition is based on the fact that this particular line, if (count == 0) actually comprises of two steps which are non atomic. The first step is that the count value which is stored in memory will be loaded into a register in the processor, and the second step is when the register value is checked to be 0 or not. So let us look at the problem that could occur because this particular execution or this particular statement is non-atomic.

(Refer Slide Time: 08:12)



So let us say, that the consumer starts executing first and its starts executing with an empty buffer. So it reads the value of count from a memory location into a register, since we are assuming that this is the initial state so the value of count that is loaded into the register would be 0. Let us then say that there is a context switch that occurred and the producer has executed. Now the producer produces an item, increments the count to 1,

and then inserts the item into the buffer. Now after it executes, let us say that there is a context switch again, as a result the consumer continues to execute from where it had stop that is from this point (point 3 in green color as mentioned above).

Now we know that it has already loaded the register previously with the value of 0. Now it is going to test whether count equal to 0, which is true in this case and therefore the consumer is going to wait (i.e last 2 green lines in above image). Essentially, it's going to wait till it receives a signal from the producer. However, the actual value of count is 1, because the producer has pushed an item into the buffer, and thus we see there is a lost wakeup that occurs. The consumer has missed a wakeup signal which the producer has sent. Now there is nothing stopping the producer from pushing more items into the buffer.

So, eventually the entire buffer is full and the producer will then wait for the consumer to remove some item. However, this will not occur because the consumer itself is waiting. Thus, we have a producer waiting for the consumer to remove an item, while the consumer is also waiting because it has missed the wakeup. Thus, we eventually reach a particular state where both producer and consumer will wait infinitely. So we see that using 3 mutexes will not solve the producer-consumer problem.

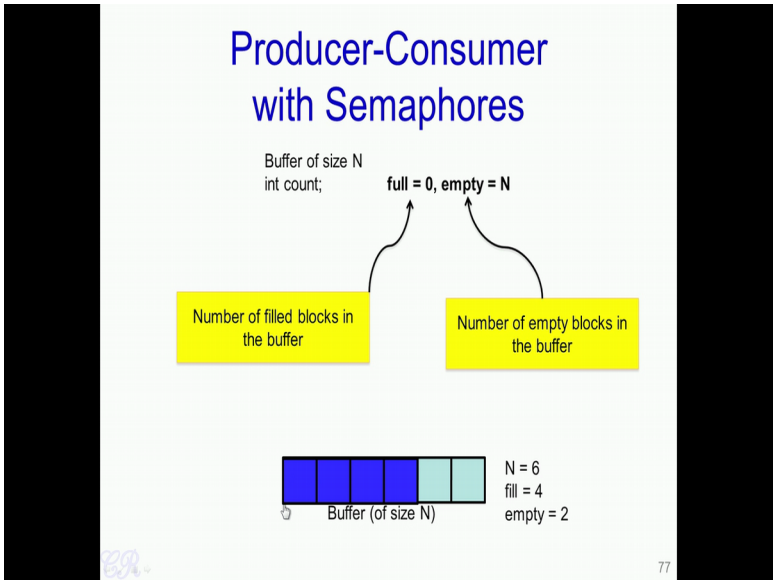
(Refer Slide Time: 10:39)

Let us look at another primitive known as Semaphores. So this semaphores is another synchronization primitive it was proposed by Dijkstra in 1965. And semaphores are implemented with 2 functions called down and up which we assume are atomic (mentioned in above slide image). So these are the two functions and thus requirement is that both these functions need to be atomic. So there is a shared memory location which is termed as S and in the down function, the while loop will test whether S <= 0.

So, as long as S has a value which is less than equal to 0 this particular loop (i.e while(*S <= 0)) will execute. When S takes a value which is greater than 0, then the loop would break and the value of that memory location S would be decremented by 1. In the up function which is also atomic the value of the memory location S is incremented by 1 (as mentioned above). So, the down and up functions are sometimes called as the P and V functions respectively from there Dutch names. And we could also have two different variants of the semaphores; we could have a blocking semaphore and a non-blocking semaphore as well. So, a non-blocking semaphore is shown over here (mentioned in above image in box), essentially it is a while loop which is resulting in a busy waiting much like a spinlock.

On the other hand we can make a small modification and have a blocking semaphore, where this particular statement (i.e while(*S <= 0)) will result in the process going to a blocked state, while a signal from the up would wake up the process. So if the values of S was initially set to 1 then a blocking semaphore is similar to a mutex, while a non-blocking semaphore is similar to a spinlock. So now, let us see how we could use the semaphore to solve the producer-consumer problem.

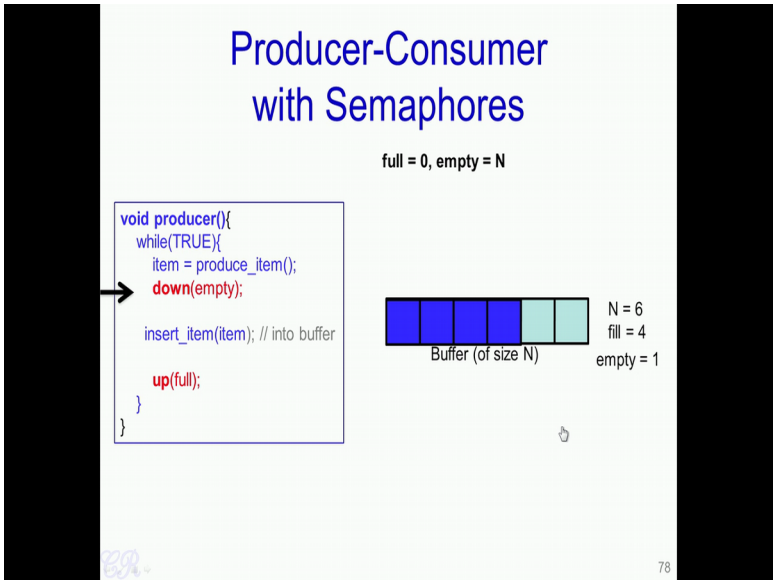
(Refer Slide Time: 12:55)



So in order to solve the problem we require two semaphores; one is known as full. So when I say two semaphores it means two memory locations which we specify by this S over here (mentioned in slide time 10:39). So we have two memory locations or two semaphores, full and empty. So, full is given the initial value 0, while empty is given the initial value N, where N here is the size of the buffer (mentioned in above slide). The semaphore full indicates the number of filled blocks in the buffer, while the semaphore empty would indicate the number of empty blocks in the buffer.

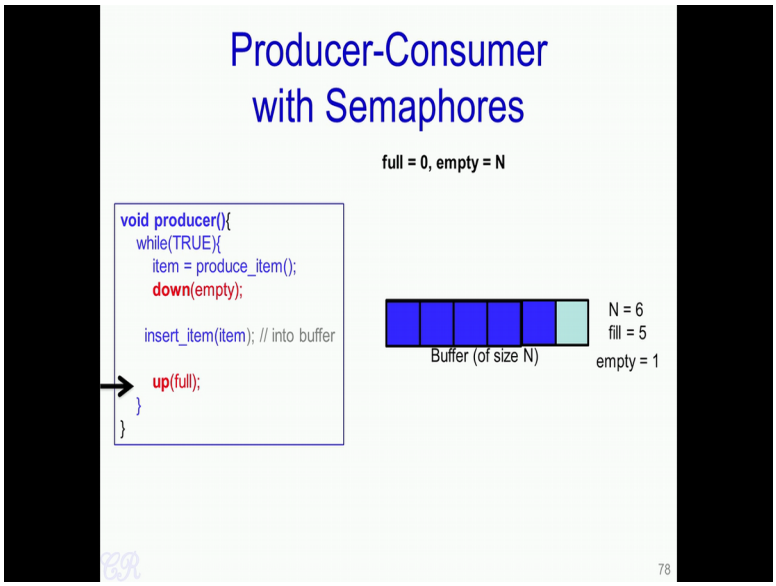
In this particular case, where N equal to 6, fill will have a value of 4 because there are 4 filled blocks and empty will have a value of 2 because there are 2 empty blocks (mentioned in above slide image). So, the initial states just before the start of execution of the producer and consumer will have full equal to 0 and empty equal to N, because there is no data items in the buffer; essentially, because the buffer is empty.

(Refer Slide Time: 14:13)



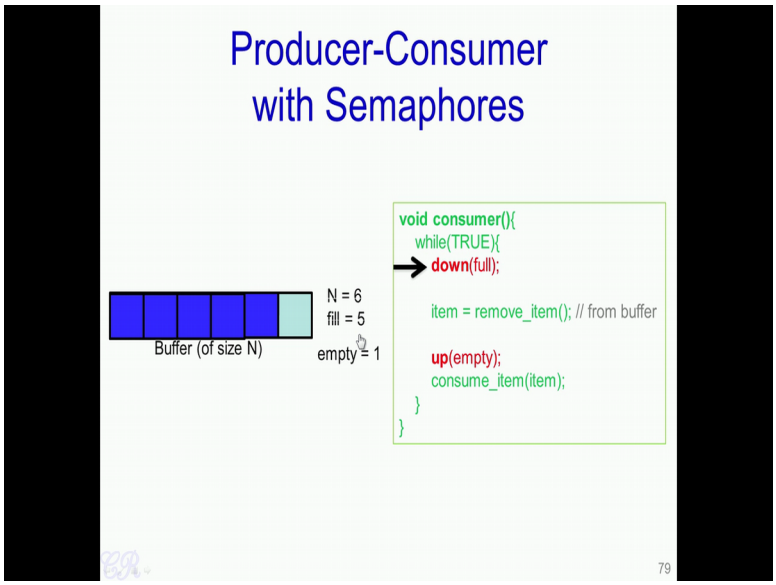
So let see, how the semaphores are used. Let us look at the producer, so the producer produces an item and we will take this particular example (mentioned in above slide image) where a fill is 4 and empty is 2, and then when the item is produced it invokes the down semaphore (i.e down(empty)). So the down semaphore, as we have seen will down the empty semaphore, so empty will go from 2 to 1. So, this is an atomic operation.

(Refer Slide Time: 14:52)



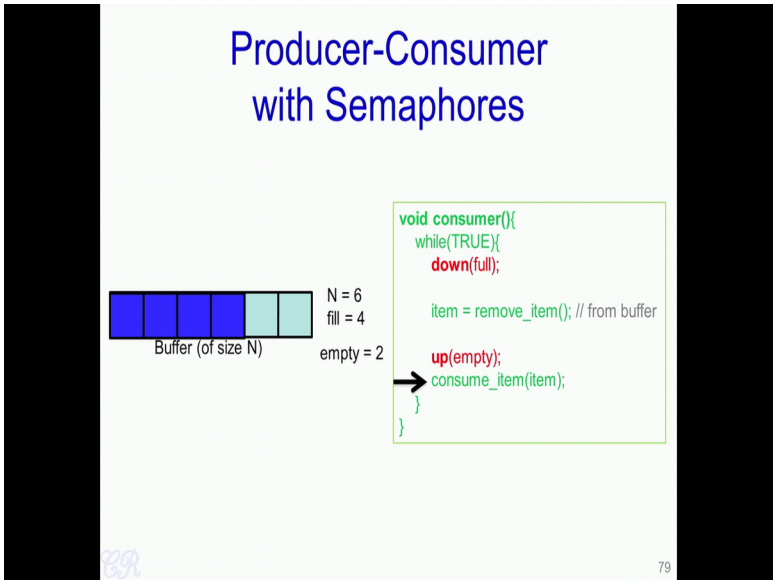
Then the next step in the producer is to insert an item, so a new item gets inserted into the buffer. Then there is an up(full) (as mentioned above) that is the semaphore full will get a value of 5.

(Refer Slide Time: 15:08)



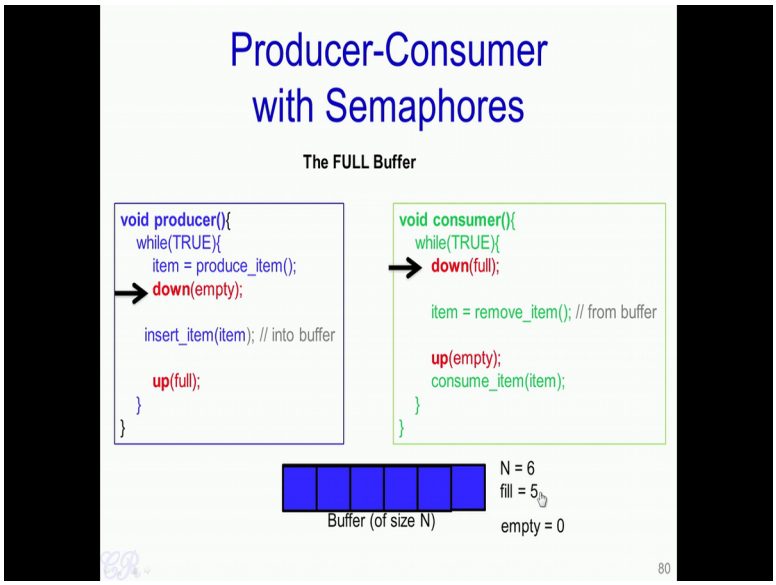
Similarly, in the consumer part is as follows (mentioned in above slide image). First there is the down(full), so the value of full will go from 5 to 4 as seen here.

(Refer Slide Time: 15:25)



Then an item is removed from the buffer and the value of empty is set to up. So, up the value of empty will become 2, and then the consumer will consume the particular item (as mentioned in above slide).

(Refer Slide Time: 15:41)

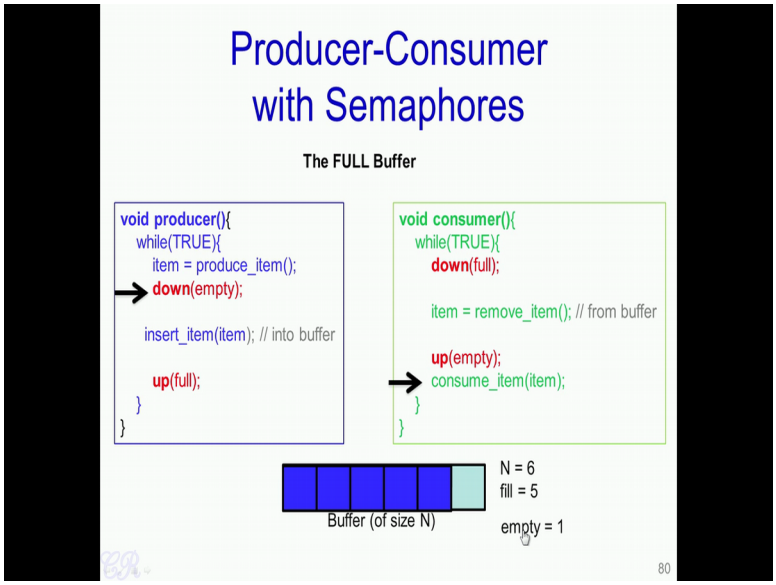


Now let us see both the producer and consumer and the case of a full buffer (as

mentioned above). So we have, let us assume that the buffer is full, so in such a case the full value has 6 which is equal to N, while the empty has a value of 0 indicating that there are no empty blocks in the buffer, and full of 6 indicates that there are 6 full blocks in the buffer. So the producer as usual will produce an item and then it will down empty (mentioned in producer() arrow). Now you see that empty has a value of 0. So if you go back to the down function of the semaphore it would cause the while statement to keep executing continuously. So, the producer would be blocked or waiting on this particular down semaphore.

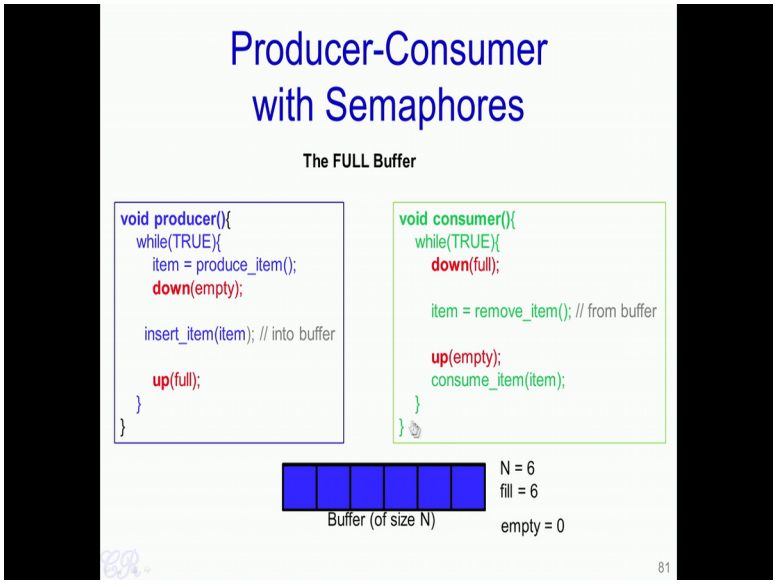
Now, after a while when the consumer begins to execute, so it will execute the down(full); so as a result, it is going to consume one particular element, so it is going to decrement the value of full as will be seeing over here (mentioned above), so full goes to 5. And then it is going to remove an item and it is going to up(empty).

(Refer Slide Time: 17:00)



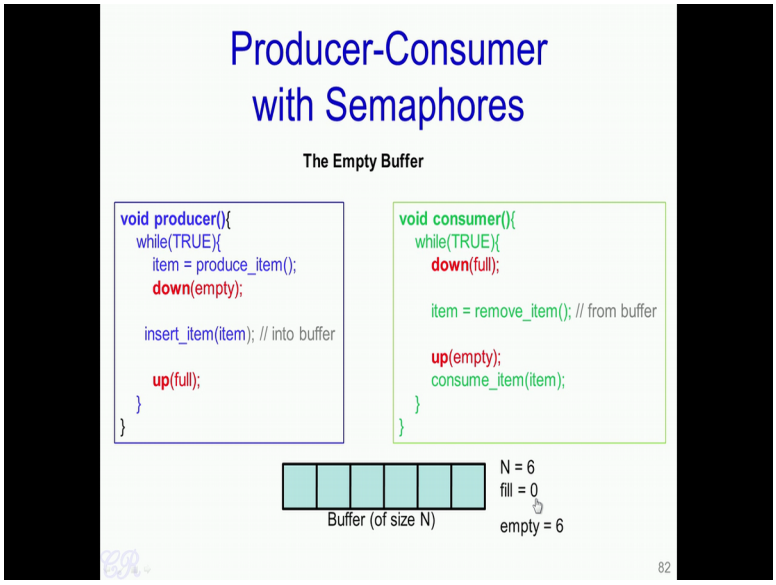
So now empty is set to 1, and then of course it is going to consume the item. Now setting empty to 1 would result in the loop in the semaphore down to wakeup or for the loop in the semaphore to complete to break (i.e down(empty) semaphore in producer()), and then the producer will set empty back to 0 and insert the item into the buffer and then the full value is set to 6 yet again.

(Refer Slide Time: 17:31)



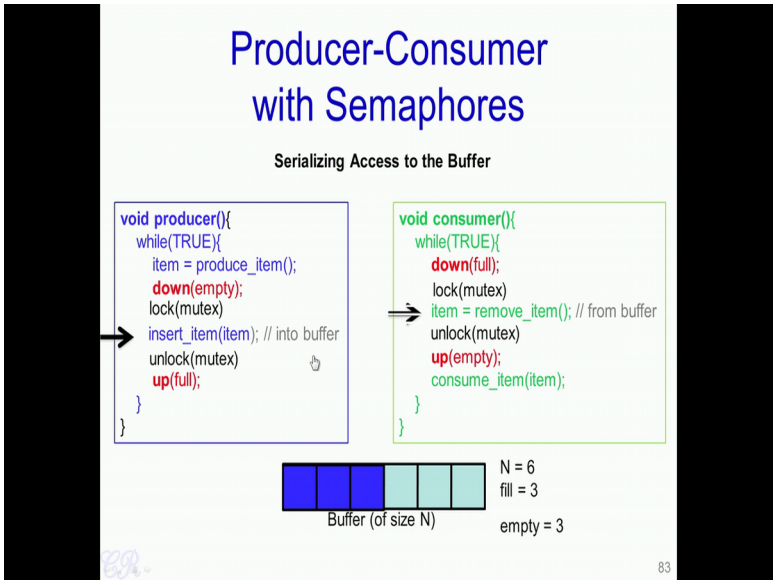
So in this way, the semaphores full and empty are used to solve the producer-consumer problem when the buffer is full.

(Refer Slide Time: 17:49)



A similar analysis can be made when the buffer is empty. In such a case the values of full and empty are 0 and 6 respectively.

(Refer Slide Time: 18:03)



So, one thing we have not taken care about so far in the producer and consumer code is that, we are not synchronizing access to this particular buffer. As a result it could be possible that the producer may be inserting item into the buffer and at exactly the same time the consumer may be removing an item from the buffer. So in order to prevent such a thing to occur we use a mutex to synchronize access into a buffer.

So before accessing this particular buffer, a producer as well as the consumer would need to lock the mutex i.e lock(mutex), and after accessing the buffer we unlock mutex i.e unlock(mutex) needs to be invoked. As a result of this mutex we can be guaranteed that only one of these two processes are executing in this particular critical section, that is accessing the buffer at any given instant of time.

So, now there are several other variants of the producer-consumer problem, and there are various schemes in which semaphores could be utilized to solve these various problems. But, for this particular course we will stop with this particular example.

Thank you.