Operating System Fundamentals

Prof. Santanu Chattopadhyay
Department of Electronics and Electrical Communication Engineering
Indian Institute of Technology, Kharagpur

Lecture - 29
Process Synchronization

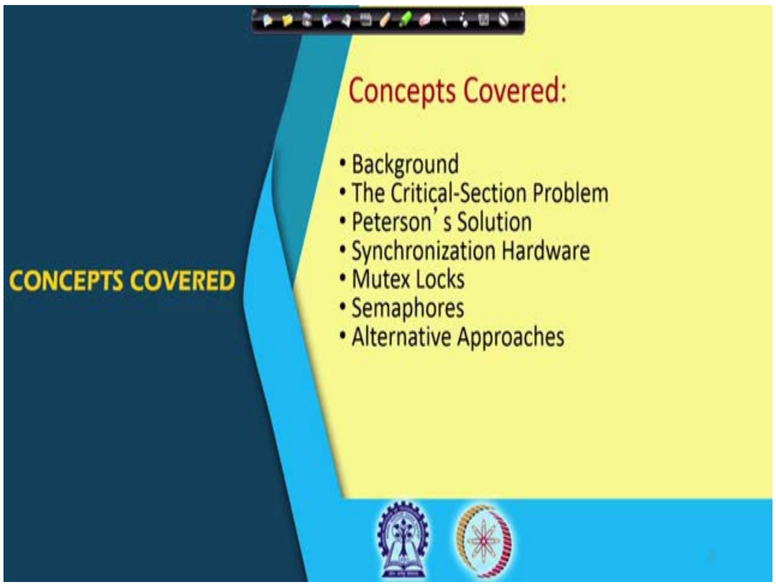
The next important concept in the field of Operating System that we are going to see is Process Synchronization. So, far we have seen that in a computer system we can have several processes that reside in the ready state and depending upon the scheduling policy, so they are scheduled in certain fashion. So, it can very well happen that the processes are scheduled round robin fashion. And when it is done the process might have executed a few statements and after that it has got descheduled, another process has got scheduled and that is there now running.

Now, between this process 1 and process 2 that I am talking about, there may be some shared resources. So, that shared resource if it is not protected properly, so we have to be there may be some problem. So, how this can be done? A typical example can be like this like say, there is a printer attached to a system, so one process is running so it is the printing something on to the printer. Now your second process comes and that also tries to write something on to the printer. Now how do you control the situation?

Because once the printer is allocated to process 1 so, it cannot be given to process 2 at the same time process 1 was running, so at that time a priority process 2 has arrived. So, that has got that has to be given the CPU. So, how to take care of this thing? So, we have got the issue of synchronization. So, another important example maybe like this that ok, there are in a banking system so there are different bank accounts and there are transactions that are going on simultaneously.

Now, if two transactions they try to modify same account simultaneously, then that may lead to some difficult situation some critical problem may come up about the consistency of the whole database. So, that way the transaction has to be controlled that in some sense. So, transactions are ultimately converted into processes. So, the process synchronization becomes an important issue. So, in this particular discussion, so we will be looking into how this synchronization is done and what are the different primitives by which we can do this process synchronization?

(Refer Slide Time: 02:38)



CONCEPTS COVERED

Concepts Covered:

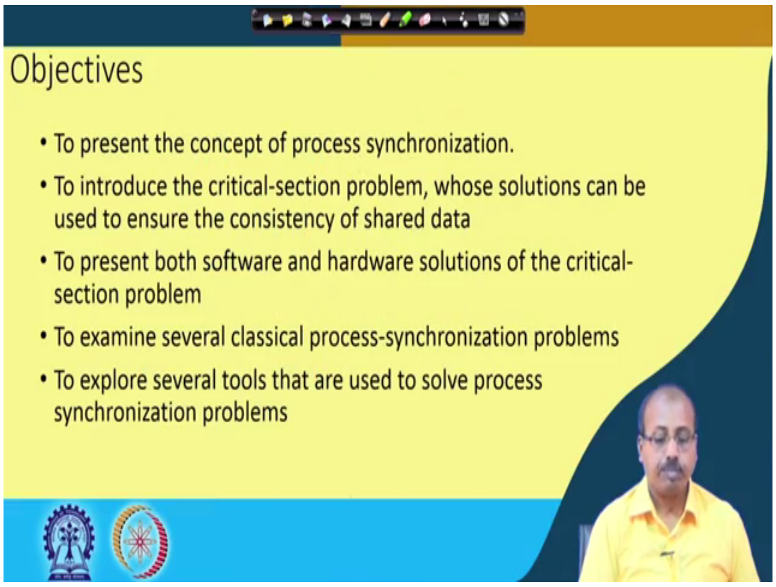
- Background
- The Critical-Section Problem
- Peterson’s Solution
- Synchronization Hardware
- Mutex Locks
- Semaphores
- Alternative Approaches

The concepts that we are going to cover are like this, apart from the background that discussions that we will have on this. So, we will be discussing about the critical section problem. So, critical section is basically a portion of code at which these some shared resources is being accessed. So, maybe I have written a 500 line program and in that in some only in 10 lines, so it is accessing some shared resource.

So, when I am trying to tell that this process should run very carefully. So, I am basically bothered about those 10 line because when the process is executing those 10 lines, it should not happen that the process is taken back the CPU is taken back from the process and all so something like that. So, that becomes a critical section, then there are several solutions to the critical section problem one of them is known as Peterson’s solution, then there are some synchronization hardware.

So, this architecture people so they try to come up with some solution to this critical section problem. So, that they provide some mechanism by which this critical section problem can be solved. And, they are basically some hardware solutions that are provided. Then there are Mutex locks which is mutual exclusion locks then semaphores and other alternative approaches. So, these are the things that we are going to see in this particular chapter.

(Refer Slide Time: 04:08)



Objectives

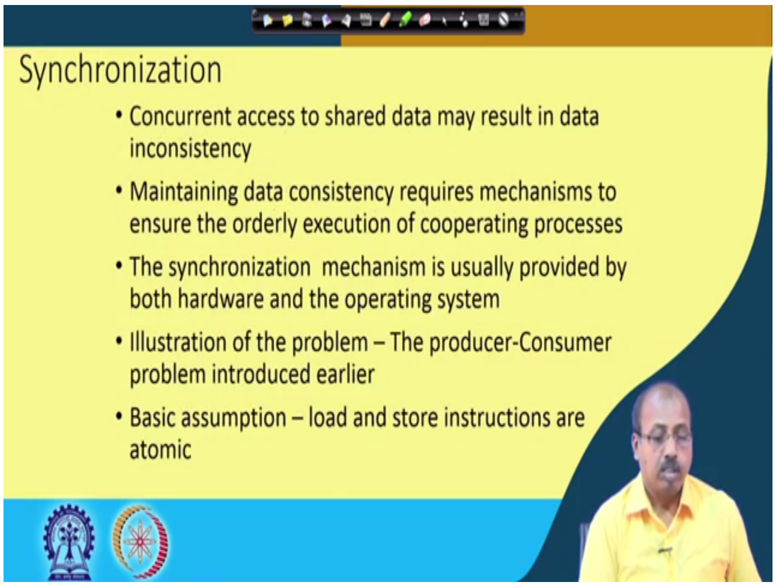
- To present the concept of process synchronization.
- To introduce the critical-section problem, whose solutions can be used to ensure the consistency of shared data
- To present both software and hardware solutions of the critical-section problem
- To examine several classical process-synchronization problems
- To explore several tools that are used to solve process synchronization problems

So, looking into the objectives, so first we will present the concept of process synchronization, then we will introduce the critical section problem and solution can be used to ensure the consistency of shared data. So in a program, wherever you are accessing some share data so that becomes a critical section because of some reason that we will explain. Then to present both software and hardware solutions for the critical section problem, we examine several classical process synchronization problem. So, this classical process synchronization problem one of them we have already seen, which is the producer consumer problem that we have discussed previously.

But there of course, we did not consider the situation where the producer and consumer, they are accessing the shared buffer and the buffer may be modified by both of them simultaneously. So, that way it may create some difficulty or there are if there are multiple producers and multiple consumers then they the multiple producers may try to write on to the same location or multiple consumers may try to read from the same location.

So, that way we have got this the classical process synchronization problems. And many of the real problems that you find in computer systems, so they can be mapped to one of these classical problem. And once this mapping can be done, so we can take help of the solutions to solve those problems. And then we explore several tools that are used to solve process synchronization problems so that we will do.

(Refer Slide Time: 05:44)

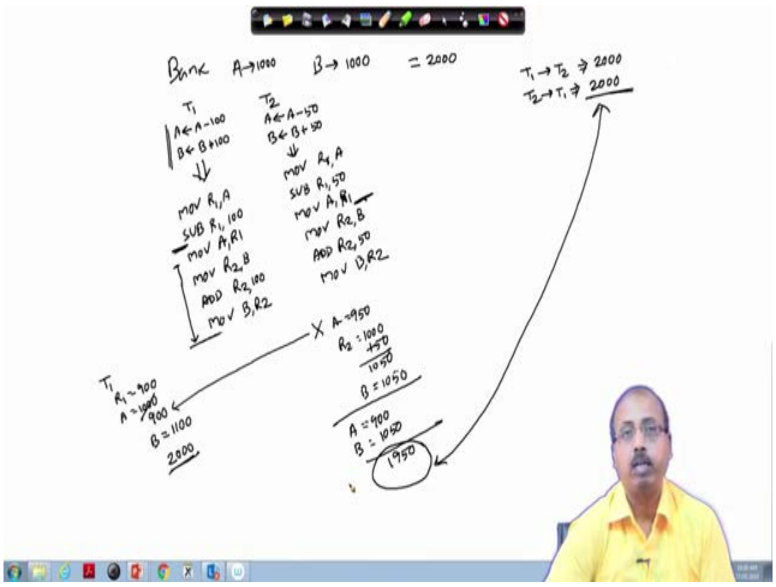


Synchronization

- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- The synchronization mechanism is usually provided by both hardware and the operating system
- Illustration of the problem – The producer-Consumer problem introduced earlier
- Basic assumption – load and store instructions are atomic

Now, what is synchronization? So, concurrent access to shared data may result in data inconsistency.

(Refer Slide Time: 06:00)



So, I will take an example and try to explain with respect to that. Say, suppose we have got in a bank, so we have got an account A. So, you have got a account A and account B; account A was initially having say 1000 rupees in it and account B was also having 1000 rupees in it. So, total amount is say is equal to 2000.

Now, there are two transactions T 1 and T 2. So, in transaction T 1 what we do? We transfer 100 rupees from account A to account B. So, what it tries to do this operation is A equal to A minus 100; and B equal to B plus 100. And transaction two so, it is transferring say 50 rupees from account A to account B. So, it is A equal to A minus 50; and B equal to B plus 50. Now if these two transactions occur in a particular sequence, that is T 1 followed by T 2 in that case the total sum after doing this transfer after executing these transactions, will be equal to 2000 because account A will be reduced by 150 rupees and account B will be credited with 150 rupees.

So, that way T 1 to T 2 if it is T 1 after T 2 then the amount is 2000. And similarly if it is T 2 after T 1 so then also the amount is the final amount is equal to 2000 the sum of the two accounts. Because in this case first 50 rupees will be transferred from A to B and then 100 rupees will be transfer from A to B.

Now, you see that in computer system when they are executed. So, they are not executed like this, but they are converted into some process and the process is executed. So, assuming that I have got registers R 1 and R 2 in the computer. So, the operation that is done is MOV R 1 comma A subtract R 1 comma 100 then MOV A comma R 1 then MOV R 2 comma B, then ADD R 2 comma 100 and then MOV B comma R 2.

So, this is the set of operations if we will look in terms of a process. So, these are the instructions to be executed for doing this operation. Similarly so this piece of this transaction, so it will give rise to a piece of code like this MOV R 1 comma A, then subtract R 1 comma 50 then MOV A comma R 1 then, MOV R 2 comma B then ADD R 2 comma 50 and then MOV B comma R 2.

Now, if this code is executed properly then there is no problem because if it is scheduled at first T 1 then, T 2 or first T 2 then T 1 then there is no problem. But in a time shared system what can happen is that maybe we start with T 1, so T 1 executes up to say this much, up to this much and then it gets descheduled. So, at this point of time, so what is the content of R 1? So, R 1 equal to first it got 1000 here and then subtraction has taken place, so R 1 has become equal to 900, but before saving this value. So, it got de scheduled now say T 2 came and T 2 it started executing and it executed say up to this much ok. So, if it had executed up to this much. So, you see when it does MOV R 1 comma A, so A value is still equal to 1000 ok.

So, A at this point also A is remaining equal to 1000. So, when it is accessing R 1 comma A, so R 1 comma 1000. So, 50 substituted to 950 then MOV A comma R 1, so, A gets the value 950 fine. Now after that suppose T 1 comes so T 2 gets de scheduled at this point T 1 comes into execution and so T 1's context is restored. So, R 1 gets back its old value of 900 and then MOV A comma R 1. So, this starts with this statement as a result the value of A becomes equal to 900.

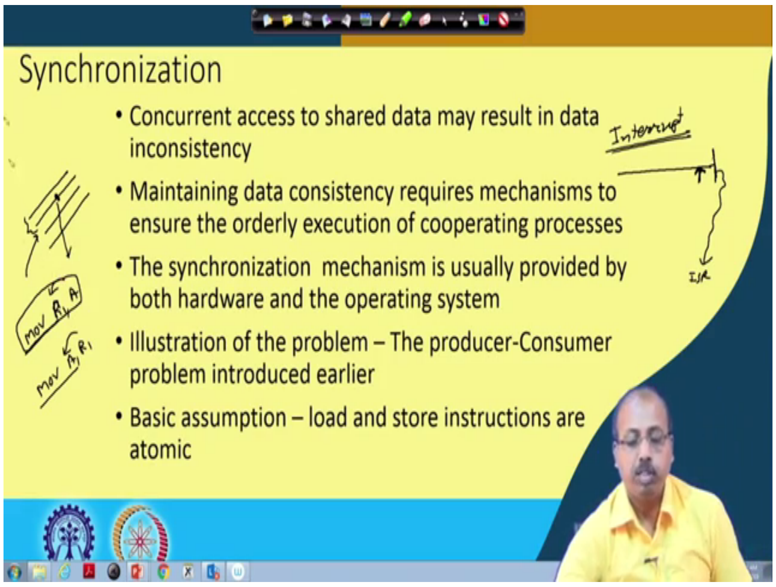
So, this 950 value that was there so, that is over written now and this gives the value 900. After that suppose that this part of the code executes property so it started with this point and the next descheduling occurs only at the end of the code. As a result B will be getting the value equal to. So, 100 will be added so B will get the value 1100. So, what is the sum after doing this set of transfer? So, this is this is 900, so this is equal to 900 plus 1100 so this is equal to so, this becomes equal to 2000 ok.

Now what about this statement? So, this statement so this will be coming now; and then R 2 comma B so B will be getting the value of. So, at this point T 1 has ended and this is the situation. Now T 2 will come back So, T 2 will come to this point and T 2 starts executing. So, R 2 gets the value equal to 1000 and then add 50; so, plus 50 so this is 1050. And then this value will B saved in B, so B will finally, get 1050. So, when so this was the situation when T 1 has finished, but T 2 was half way. So, when T 2 was also finished now the value of A is equal to 900; value of A is equal to 900 and value of B is equal to 1050 fine.

So, what is the total now? So, total is equal to 1 1950 ok. So, why this thing has happened? This thing has happened because when a T 1 was accessing the value of a in between T 2 came and it got some inconsistent value of A and with that it is preceded. So, as a result so the final content of A plus B becomes equal to 1950. Whereas, in a proper execution if T 1 was executed completely before T 2 or T 2 was executed completely before T 1 the sum should have been 2000, so there is an inconsistency here ok.

So, if you think about descheduling of T 1 at many other points also. So, you can find that you are getting some different-different results for the sum of the two account. So, that is that is undesirable because what we expect is if both of them are executed properly, both the transactions are executed properly then I should do the total amount of sun should remain 2000 only. So, this is the problem of concurrency.

(Refer Slide Time: 17:21)



Synchronization

- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- The synchronization mechanism is usually provided by both hardware and the operating system
- Illustration of the problem – The producer-Consumer problem introduced earlier
- Basic assumption – load and store instructions are atomic

So, we will see that how this can be solved. So, this concurrent access of shared data may result in data inconsistency. So, that point is understood and maintaining this data consistency it requires mechanisms to ensure orderly execution of operating processes. So, if one possibility in the previous example is that somehow we ensure that T 1 is executed completely before T 2 or T 2 is executed completely before T 1 so that is one thing.

Or, when T 1 is accessing A T 2 should not be allowed to access A and when T 1 is accessing B T 2 should not be allowed to access B or vice versa when T 2 is accessing A T 1 should not allowed. So, whenever some shared access is taking place some other process should not be able to access that a resource, until and unless this is done by the first process. So, maintaining data consistency it requires mechanism to ensure orderly execution of cooperating processes the synchronization mechanism is usually provided by both hardware and operating system.

So, we will see some mechanism by which this synchronization may be done by hardware or by the operating system. So, this producer consumer problem that we have discussed previously, so that is there so all this discussions will have. So, we will assume that machine instructions they are atomic. So, this load store instructions are atomic. So, you when the content is being loaded from CPU like by the statements we had previously like MOV A comma R 1 or say R 1 comma A which is a load instruction. And value of A comes to R 1 or the instruction like MOV A comma A comma R 1, when R 1 is the value is being stored in A.

So, this is a load and store instructions, so this is basically a load type instruction this is basically a store type instruction. So, we assume that this is atomic, so you can you cannot stop the execution in between ok. So, if you look into any computer architecture you will find that, so when a when a processor is executing instructions; then to stop it in between or to deschedule the process etcetera what we have to do is that we have to send some sort of interrupt.

So, that interrupt may be generated by a timer or maybe from some IO device or whatever. So, that interrupt is generated, but whenever this interrupt occurs so if processor is currently executing an instruction. So, if the interrupt occurs anytime in between so the first thing that it does the processor does is it completes the current instruction. And, then it may from this point it may branch to some I interrupt service routine, but it is not that as soon as the interrupt has occurred it will immediately branch to ISR. So, maybe the processor was executing this MOV R 1 comma A instruction and somewhere in between the interrupt has occurred.

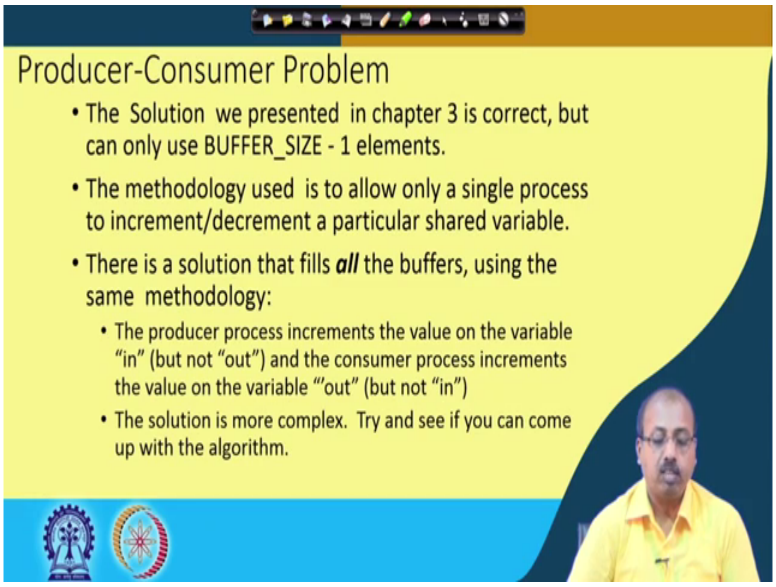
So, the processor will finish this instruction and then go to the ISR it is not that it will go in between. So, that is the that assumption is there. So, any architecture book that will find or any microprocessor where book that you find, we will see that this atomicity is maintained. For example, in case of 8085 processor you may find out that the interrupt occurrence is checked at the last, but one clock cycle of every instruction execution.

So, that after it is the last, but one clock cycle the interrupt conditions are checked and if the interrupt is there, then after finishing the next clock cycle only so it will be going into the interrupt service routine. So, execution of a single instruction cannot be stopped in between. So, it will always go on to the end and then only it can be the processor can branch to some interrupt service routine.

So, this is the basic assumption that load and store instructions they are atomic in nature. So, you cannot wait the execution in between, so you cannot say that this instruction is half done. Whereas if whenever you have got a sequence of instructions, so if I have got a sequence of instructions, then after executing two instruction. So, there can be an interrupt and then the processes switches to some other process that can happen.

But it cannot be that it has executed up to this much and then the interrupt has occurred and it came out of execution from this point itself so that does not happen. So, we will look into this as we proceed so, we will see the techniques for we will see the techniques for doing this synchronization.

(Refer Slide Time: 19:10)



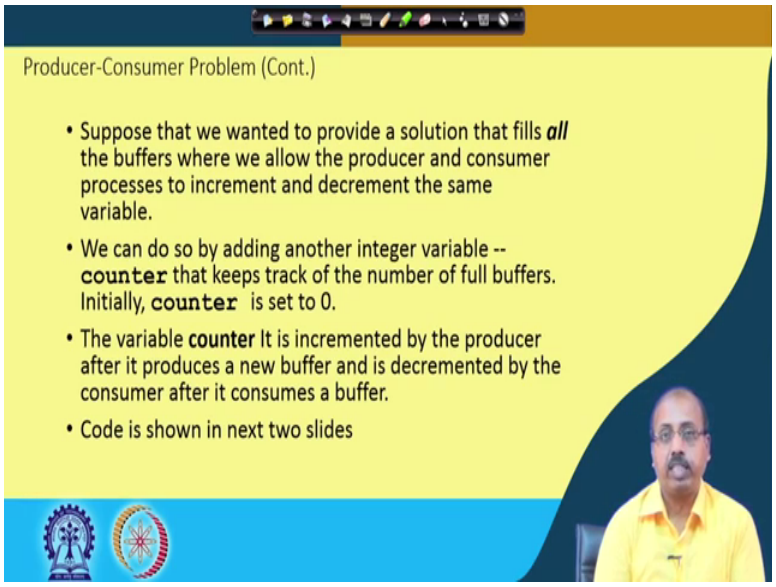
Producer-Consumer Problem

- The Solution we presented in chapter 3 is correct, but can only use BUFFER_SIZE - 1 elements.
- The methodology used is to allow only a single process to increment/decrement a particular shared variable.
- There is a solution that fills ***all*** the buffers, using the same methodology:
  - The producer process increments the value on the variable "in" (but not "out") and the consumer process increments the value on the variable "'out" (but not "in")
  - The solution is more complex. Try and see if you can come up with the algorithm.

The first problem that will be looking into is the that we have got this producer consumer problem. So, this is a producer consumer problem that we have already seen a solution in the previous chapter, but it can only use buffer size minus 1 element. So, buffer is never full, so that is at least 1 element that is area that is filled. The methodology used to allow only a single process to increment or decrement a particular shared variable.

So, this was the technique that that was followed. So, only one process can increment or decrement the shared variable. There is a solution that fills all the buffers using the same methodology the producer process instruments the value on the variable in, but not out. And the consumer process implements the value of the variable out and, but not in. So, that way this solution can be done, but the solution becomes pretty complex. So, if you can you can try out this process, but this becomes pretty complex.

(Refer Slide Time: 20:14)



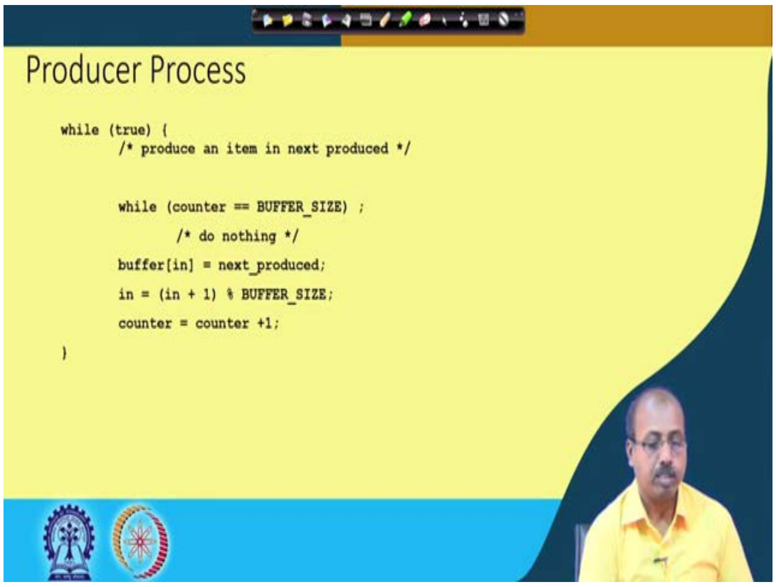
Producer-Consumer Problem (Cont.)

- Suppose that we wanted to provide a solution that fills ***all*** the buffers where we allow the producer and consumer processes to increment and decrement the same variable.
- We can do so by adding another integer variable -- **`counter`** that keeps track of the number of full buffers. Initially, **`counter`** is set to 0.
- The variable **counter** It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.
- Code is shown in next two slides

So, using this synchronization problem, so we will see that there can be better solution for these producer consumer problems. A typical example is typical problems that can occur is like this, suppose we wanted to provide a solution that fills all buffers that we allow the producer and consumer processes to increment and decrement the same variable.

So, we can do this by having another integer variable counter. So, this counter variable it keeps track of the number of full buffers how many buffer areas are free? So, initially counter is set to 0 and the variable counter is incremented by producer after it produces a new buffer and decremented after the consumer consumes a new buffer. So, this counter value at any point of time it holds the number of elements that are filled in the buffer.

(Refer Slide Time: 21:11)



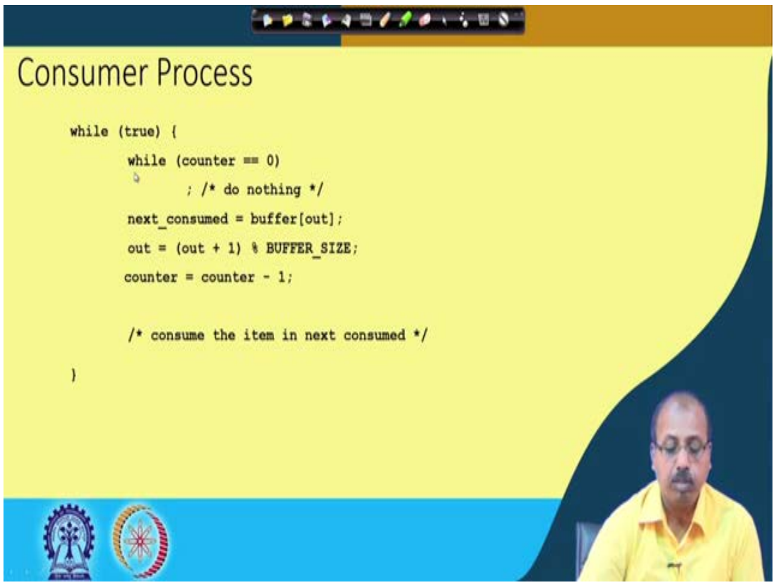
Producer Process

```
while (true) {
        /* produce an item in next produced */

        while (counter == BUFFER_SIZE) ;
                /* do nothing */
        buffer[in] = next_produced;
        in = (in + 1) % BUFFER_SIZE;
        counter = counter +1;
}
```

So, initially the value is 0 as the producer is producing further items so, counter value is incremented and as the consumer is consuming some item the counter value is decremented so that is the situation. So, this is the producer process. So, produce an item in next producers. So, and then while counter equal to buffer size then of course, you cannot do anything. Then otherwise, so when the buffer is free when the counter is not equal to buffer size, so in that case the next produced item is put into the buffer then the in pointer is incremented by in plus 1 percent buffer size and then counter is incremented by 1, so counter equal to counter plus 1.

(Refer Slide Time: 21:38)



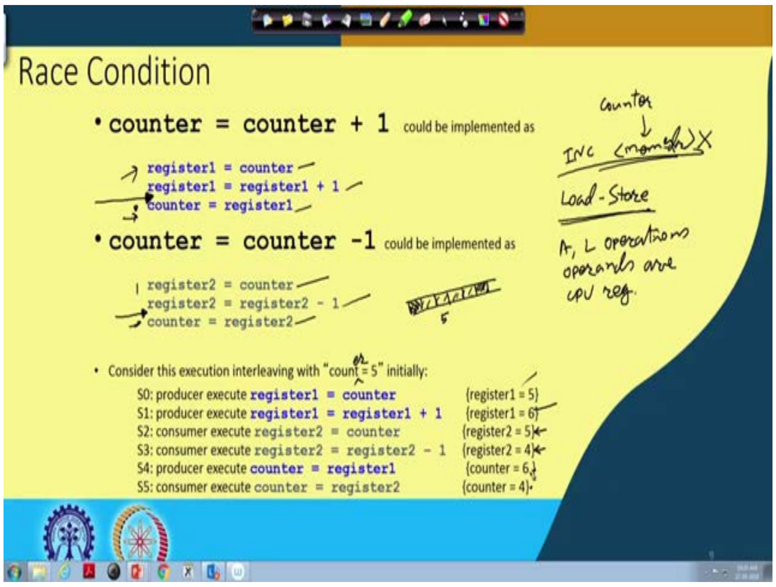
Consumer Process

```
while (true) {
        while (counter == 0)
                ; /* do nothing */
        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        counter = counter - 1;

        /* consume the item in next consumed */
}
```

On the other hand the consumer process, so it checks for the counter value to be equal to 0 and if it finds that the counter value is equal to 0, then there is nothing in the in the buffer. So, the counter the consumer process will wait. Then from the out position, so, it will read the next item to be consumed. So, next consumed equal to buffer out, then out pointer will be implemented and the count value will be decremented. So, this is the very standard solution that we can have for solving this producer consumer problem.

(Refer Slide Time: 22:12)



Race Condition

- `counter = counter + 1` could be implemented as

```
register1 = counter
register1 = register1 + 1
counter = register1
```

- `counter = counter -1` could be implemented as

```
register2 = counter
register2 = register2 - 1
counter = register2
```

- Consider this execution interleaving with "count = 5" initially:

```
S0: producer execute register1 = counter            {register1 = 5}
S1: producer execute register1 = register1 + 1      {register1 = 6}
S2: consumer execute register2 = counter            {register2 = 5}
S3: consumer execute register2 = register2 - 1      {register2 = 4}
S4: producer execute counter = register1            {counter = 6}
S5: consumer execute counter = register2            {counter = 4}
```

But there is an issue, just like that problem that we had previously when we had this two bank accounts and two transactions trying to access those accounts simultaneously. So, here also you see there in this in this producer and consumer processes what is happening is that this counter variable, so that is shared between the producer and the consumer.

So, producer is trying to implement it, consumer is trying to decrement it. And if this thing happens in an uncontrolled fashion, then it can lead to some inconstant value in the counter; so that we are trying to see. Suppose this counter equal to counter plus 1 and assuming that the underlying processor it does not have the increment memory instruction. So, some processor may so counter is ultimately a memory location, so if I can directly have an statement like as so some processor. If it has some increment some memory address, increment some memory address so then that instruction can be used for implementing this counter.

So, counter address can be given here and this increment counter will be working there. So, for that type of processors this discussion is not valid. So, we assuming that we do not have this memory address as the operand of the implement instruction. So, all the arithmetic logic and so many processors we will find that all arithmetic logic operations are in the CPU registers only. So, they are commonly known as load store architecture and most of the processors that we have now many of them they have got this load store architecture.

So, in load store architecture, what it means is that all the arithmetic logic operations arithmetic and logic operations, so their operands must be in the CPU registers, so operant are CPU registers. So, if you want to do some increment operation then what you have to do is that first we have to load the counter value into some register. Then the register value has to be incremented and then the value of the register should be stored back into the counter. On the other hand if you are trying to do say counter equal to counter minus 1.

So, this has to be done like this may be some other registers register 2 is loaded with the counter value it is decremented and the value stored in the counter location. So, this is our two system and suppose the sequence of operation happens like this. So, initially the count value was equal to 5, the counter value was equal to 5 and; that means, the buffer had 5 locations full, so this 5 locations were full. Now suppose producer produced the next item as a result it is so it executes so it produced next item.

So, naturally so it try to increment this counter value. So, it executes the statement register equal to counter register 1 equal to counter. So, register 1 is equal to 5 then producer executes the next statement register 1 equal to register 1 plus 1, so register 1 gets equal to 6, but after executing this two statements the producer process got descheduled at this point. Now, the consumer process came and the consumer process started executing from this point. So, register two equal to counter. So, register 2 gets the counter value which was equal to 5. So, you see though the register 1 value has been upgraded to 6 it is not yet reflected in the counter.

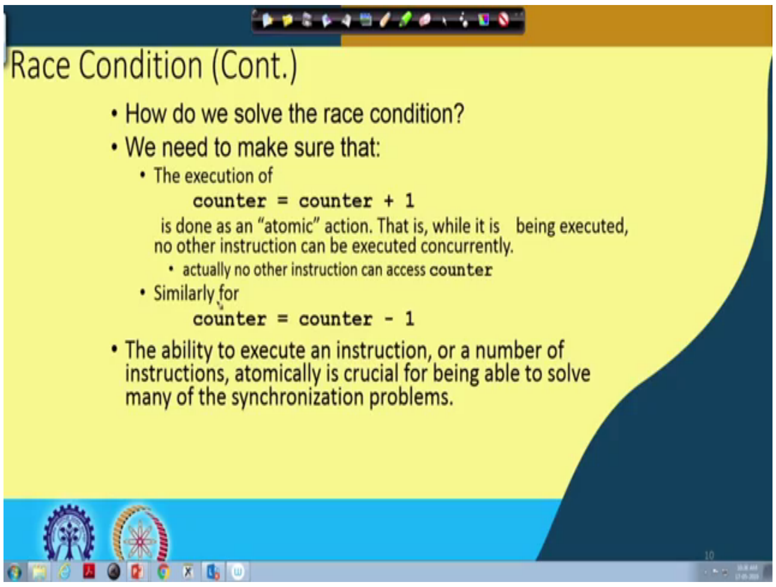
So, the counter value remains at 5, so register 2 equal to counter so, register 2 gets the value 5. Then register 2 is decremented by 1 so, register 2 value becomes equal to 4. And now suppose the consumer get descheduled after executing these two statements, that the count the consumer gets descheduled and the producer process comes back. And, when the producer process comes back, it starts executing from this point so it says counter equal to register 1. So, as a result counter get the value 6. And, after that this register this producer is over. So, the consumer comes back a consumer executes the last statement that was pending here counter equal to register 2.

So, counter register 2 value was 4 so counter gets the value 4. Whereas what has happened is this after this count the counter value was equal to 5, one more item has been produced, but at the same time one item has been consumed by the consumer. So, the counter value should remain at 5, but due to this scheduling in some intermittent fashion intermittent all that the process is getting descheduled in some intermittent fashion this counter value has got some inconsistent result. So, this counter value has become equal to 4 which is not the correct one. So, this is as if the two processors they are contending between each other, they are racing between each other to update the shared variable. So, this is a very common phenomena that you can find across a number of processes in a computer system, whenever they are trying to use some shared resource.

And, as you know that this was whenever we have got this threads are running parallelly. So, threads they one thread modifying some variable in the data segments, so all other threads will also see the effects as a result. So, there is a race among the threads. Similarly, there if the at the process level, so if the processes they have got some shared variable created in the shared memory then there will be race across the processes to update that shared memory location. So, this update this race condition is very common and this race condition has to be solved carefully. And unfortunately the OS designer cannot do much about this thing because ultimately this programs this processes are developed by users.

So, users must be knowledgeable like how can I do this synchronization? So, operating system will provide mechanism by which this synchronization can be done, but their proper utilization the responsibility lies with the system users ok. So, as a system level programmer I should be able to do the write my program in some synchronized fashion. And, whenever I am writing parallel program, so it is expected that I have got enough understanding and enough maturity that this I can figure out where this race conditions can take place. And accordingly take a decision and try to solve this race condition properly.

(Refer Slide Time: 29:13)



Race Condition (Cont.)

- How do we solve the race condition?
- We need to make sure that:
  - The execution of

    ```
    counter = counter + 1
    ```

    is done as an "atomic" action. That is, while it is being executed, no other instruction can be executed concurrently.
    - actually no other instruction can access `counter`
  - Similarly for

    ```
    counter = counter - 1
    ```
- The ability to execute an instruction, or a number of instructions, atomically is crucial for being able to solve many of the synchronization problems.

So, if we so, for solving this race condition, so we have got this situation like how do we solve this race condition? So, basically what we need to do is that the execution of this counter equal to counter plus 1, it has to be done as an atomic operation. So, it is not that I can do it so it should not be interrupted. In between that is while it is being executed no other instruction can be executed concurrently. Actually no other instruction can access the counter.

So, this has to be done similarly for this counter equal to counter minus 1, we have to have the facility that no other instruction is accessing this counter simultaneously. The ability to execute an instruction or a number of instruction atomically is crucial for being able to solve many of the synchronization problem. So, almost always you will find that this ability is required we want to have a piece of code that can be executed in an non-interrupted fashion so a atomic.

So, we will see in our next class how this atomicity can be ensured across these executions.