

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

Lecture – 32
Process Synchronization (Contd.)

So, the last solution that we have seen for this mutual exclusion problem using this test and set instruction, the problem was that the bounded waiting was not satisfied. So, as I took the example like before P 2 can get a chance for execution. Maybe P 3 is scheduled before that and P 3 actually tries to see like, P 3 checks the test and set condition and it gets chance for execution. And that way a process may be waiting in definitely.

So, all other processes they are coming at more opportune moment when this lock was set to falls, ok. So, they came at more opportune moment and got chances for execution. And may be a victim process, so it happens that it whenever it comes it finds the somebody is in the critical section.

(Refer Slide Time: 01:09)

```
do {
    waiting[i] = true; ←
    key = true; ←
    while (waiting[i] && key)
        key = test_and_set(&lock);
    waiting[i] = false; ←
    /* critical section */
    j = (i + 1) % n; ←
    while ((j != i) && !waiting[j])
        j = (j + 1) % n;
    if (j == i)
        lock = false; ←
    else
        waiting[j] = false;
    /* remainder section */
} while (true);
```

Diagram: $P_0, P_1, P_2, \dots, P_{n-1}$
CS (Critical Section)
 $j = r+2$

So, to avoid that type of situation; so, we can have a condition in which this mutual exclusion is satisfied with this test and set at the same time this bounded waiting is also satisfied.

So, how is it done? So, a process P_i that executes so, its code is like this first it raises $waiting_i$ to be equal to true, so it is waiting to enter into the critical section. It sets a variable key to be equal to true, then it checks this it goes into this while loop checks for this lock value. So, if the lock was initially false, if the lock was initially false for the first process actually then this key value becomes equal to false, this key value becomes equal to false. So, it comes out of this while loop and then it sets the $waiting_i$ to be equal to false. So, this now this process is allowed to enter into the critical section, the P_i was the first process, so it enters into the critical section.

Now, after this process P_i has finished its execution in the critical section now who will get chance for execution. Now, in the previous solution that we had, so we did not put any condition like who will enter into the critical section next. What we did we left it absolutely to the scheduler to pick up a process for execution and whichever process is picked up for execution it will find that lock is now available and it will enter into the critical section. But in this case what we do, we do something like this. So, we say that I have got a number of processes P_i, P_{i+1}, P_{i+2} , maybe I have got a number of processes up to say P_n , up to $P_n - 1$.

So, what we do? So, we check these processes in this sequence we check the processes in this sequence $i+1, i+2, \dots, n-1$ and again P_0, P_1 like that. And then you see this j variable is set to be $i+1 \pmod n$ and while $j \neq i$ and not of $waiting_j$, so we increment j . So, we find out which process is waiting. So, if the, so first j is $i+1$, so j comes here and this P_{i+1} if it is not waiting then j will advance to this one. Suppose, this was waiting. So, in that case, so this will come out of this if $j \neq i$ and not of $waiting_j$ and then. So, it will be coming out of this loop.

So, if j is not equal to i , so. So, is if j is if since j is not equal to i , so it will set $waiting_j$ to be false. So, since it makes $waiting_j$ equal to false. So, next time the process comes here. So, it will be the $i+2$ th process that is the j th process here. So, when it will execute this section, so it will be finding that this while loop condition has become false because by setting this $waiting_j$ equal to false, so when j becomes equal to $i+2$, so this $waiting_{i+2}$ has become false. So, when this P_{i+2} will be executing this particular statement $waiting_{i+2}$. So, $waiting_{i+2}$ is already false, so it will be through this critical section. So, it will be entering into the critical section.

So, this is how it is done it is actually trying to check which process after this who is waiting. So, it is checking in this order. So, you can say there is a bounded wait because at most. So, if suppose this process P 2 it has raised request for entering into the critical section, since the checking maybe like this, so we know that at most n processes or n minus 1 processes will be allowed before P 2 gets chance for execution.

So, there is a bounded wait here which is determined by this particular order of this checking. So, this particular statement, so this particular statement, so this will be ensuring that I am checking in this order and ensure that this j value this waiting j is made false here. So, if it finds that nobody is waiting. So, for a so happen that none of the processes are waiting; so, in that case j will become equal to i after sometime and it will come out of this while loop.

Now, you see j equal to i then lock variable is set to be equal to false. That is now right now nobody is willing to enter into the critical section, so now, field is open. So, anybody executing this while statement this lock test and set lock statement, so it will find that lock to be equal to false and the key value will become equal to false and that will enter into the critical section. So, if nobody is waiting then there is no problem. If there is a number of processes waiting then it will go in that particular order and accordingly it will be setting the waiting variable to be equal to false so that it can come out of the while loop. So, this is how we can modify this test and set instruction and we can make it a bounded wait type of condition.

(Refer Slide Time: 06:19)

compare_and_swap Instruction

□ Definition:

```
int compare_and_swap(int *value, int expected, int new_value) {
    int temp = *value;

    if (*value == expected)
        *value = new_value;

    return temp;
}
```

Atomic

□ Properties:

- Executed atomically
- Returns the original value of passed parameter "value"
- Set "value" to "new_value" but only if "value" == "expected".

That is, the swap takes place only under this condition.

Next we will see another statement which is known as compare and swap instruction so, basically the swap instruction. So, this compare and swap. So, this statement is like this. So, if you look into the meaning of the statement, again I should tell you that this whole thing is available at the processor level and this whole thing is atomic. So, atomicity is ensured by the processor designer that the whole thing is available. This compare and swap if it is available as an instruction. So, it is set to a value it is a set to be an atomic instruction.

So, what are the things that we are passing? So, we are passing the address of a variable value and expected value and the new value, and integer variable expected and an integer variable new value. So, in the variable temp we are getting the content of the current value of. So, we have got a memory location whose name is value, ok. So, this temp we get, in this temp variable we get the value of this location then if value is equal to the expected value then value will be set to new value.

So, we are passing an expected value here and we see that if this one and this one, these two values match in that case this new value will be copied into this value variable. And then this temp was temp will be returned. So, whatever we are returning is definitely the old value of this memory location value and we are setting it to new value if the value a current value is equal to the expected value.

So, this is the thing the meaning of the statement. So, it is executed automatically and returns the original value of pass parameter value and set value to new value, but only if value is equal to expected. So if the of a previous content of this variable value was equal to whatever we have passed as expected. So if that is condition is satisfied then only the value variable will be set to the new value. That is swap takes place only under this condition. So swap is happening only if we have got this new value sorry, the expected value is same as the previous value that is stored in the variable value.

(Refer Slide Time: 08:39)

Solution using compare_and_swap

- Shared integer "lock" initialized to 0;
- Solution:

```
do {  
    while (compare_and_swap(&lock, 0, 1) != 0)  
        ; /* do nothing */  
        /* critical section */  
    lock = 1;  
        /* remainder section */  
} while (true);
```

- What about bounded waiting?
- Solution results in busy waiting.

Now, how this thing helps us in executing this mutual exclusion problem is like this. We shared integer lock is initialized to 0 and then we do it like this while compare to compare and swap address of lock 0, 1. So this is the expected value. So, this is the expected value and this is the new value that we want to set. So, if the previous value of lock was equal to 0 in that case the new value of the lock will be set to 1, otherwise if the new value is already equal to one then we do not do anything so that is quite simple. So so there is nothing to be done.

So, if compare and swap; so this instruction returns the previous value of lock and the previous value of lock was 0; that means, lock is not available then we will be waiting in this while loop in definitely. If it is if we find that this value is not equal to 0. So, it is equal to 1, so this lock variable was initialized to 1 and this particular instruction. So, it will be it will be returning the current value of lock and if that lock value is 1 then it will

be entering into the critical section. After finishing the critical section, it will set lock to 0 and then it will be going into the remainder section.

Again, the same thing so this ensures mutual exclusion there is no problem with that. But bounded waiting is not satisfied because if I have got a number of processes. So, whichever process got scheduled whichever process is more opportune. So, got scheduled after the lock has been reset by the currently current, current process is executing in the critical section, so in that case it will be that process we gets the chance to enter into the critical section. So, there is no bound on the number of times a process may be denied to enter into the critical section.

And this solution definitely results in a busy waiting condition because I am continually checking this compare and swap type of instruction and by executing this instruction I am getting the previous value of lock and checking it whether it is equal to 0 or not. So, that way this solution is a busy wait type of solution. So, that is fine and bounded wait is not satisfied.

(Refer Slide Time: 10:59)

Mutex Locks

- Previous solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve critical section problem
- Simplest tools is the **Mutex lock**, which has a Boolean variable "available" associated with it to indicate if the lock is available or not.
- Two operations available to access a Mutex Lock:
 - `acquire() {
 while (!available)
 ; /* busy wait */
 available = false;
}`
 - `release() {
 available = true;
}`

Handwritten box on the right:
acquire
lock()
release()

So, another type of locks that are available in some processors. So, they are known as mutex locks. So, previous solutions are complicated and generally inaccessible to application programs. So, this test and set, and this compare and swap. So, both these instructions, so they are at the machine level instruction.

And whenever you are writing a program at a high level, so it is very much likely that I do not, I do not write in the assembly code for some part of the program. So, I will like to write at high level. So, the naturally and the semantics exact semantics that is there so that is a very much is dependent on the processor designer. So, from processor to processor this format and all they may vary a bit.

So, it is difficult for a programmer to make a generalize program which will be running on all the systems. So, these solutions are complicated and generally inaccessible to application programmers. So, OS designers build software tools to solve this critical section problem. So, as a OS designer our responsibility is by looking into the facilities that are provided by the underlying hardware underlying processor, we may develop some high level primitive that an application programmer may use for doing this for solving the critical section problems.

The simplest tool is the mutex lock. So, this is a type of lock if a Boolean variable available associated with it to indicate the if the lock is available or not. So, this mutex lock either it is available or it is not available. So, the two operations are available to access a mutex lock. So, there are two, one is one is called one operation is called acquire. So, this is this is a Boolean variable. So, so mutex lock implementation it has, so you can think as if as there is a memory location whose name is say available, this is an attribute. And there are two methods by which you can you modify this one is the acquire, another is release.

So, from an object oriented view point, so you can look this mutex lock as an object that has got a private variable data element called available and it has got two public access routines acquired and release. So, this acquire routine is like this. So, while not available, so it will be waiting there. So, any process calling this acquire routine will be put into an infinite loop if this variable available is equal to 0, if it is false. And after sometime, so available is true in that case it will come out of this while loop and then it will set available to be equal to false.

So, any number of processes executing this acquire function. So, it will be doing this check and it will be executing it. And then the release is nothing, but available equal to true. Now, this mutex lock, so this is provided and this operating system designer will

ensure that this mutex lock acquire routine and released routine so they are executed in an atomic fashion. So, that has to be there.

(Refer Slide Time: 14:16)



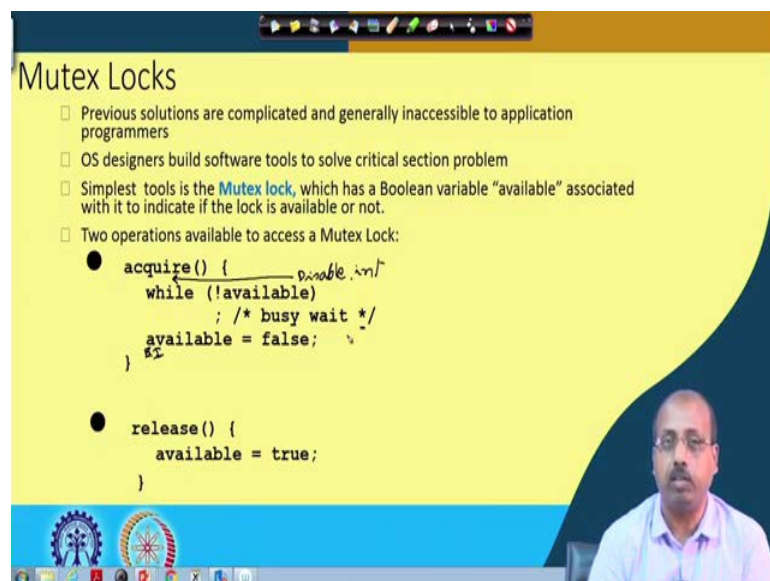
Mutex Locks (Cont.)

- Calls to `acquire()` and `release()` are atomic
 - Usually implemented via hardware atomic instructions
- Usage:

```
do {
    acquire lock
    critical section
    release lock
    remainder section
} while (true);
```
- Solution requires **busy waiting**
 - This lock is therefore called a **spinlock**

So, calls to acquire and release are atomic so that is what I was telling that it they cannot be interrupted in between. So, if it is implemented on a single processor system, so OS designer should ensure that there is a disable interrupt call before going into this acquire routine and enable interrupt call after going into the after coming out from the acquire routine.

(Refer Slide Time: 14:37)



Mutex Locks

- Previous solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve critical section problem
- Simplest tools is the **Mutex lock**, which has a Boolean variable "available" associated with it to indicate if the lock is available or not.
- Two operations available to access a Mutex Lock:
 - ```
acquire() {
 while (!available)
 /* busy wait */
 available = false;
}
```
  - ```
release() {
    available = true;
}
```


So, the previous code that you can have, so you can very easily think that, at this point there is a call to disable interrupt and after this, so there is something called enable interrupt. But of course, there are other issues like if you are if I am waiting in these loop then, so how this disable interrupt will disable all the interrupts, so how this other process will be interrupted, so that that condition is there. But let us not going to that complicacy now, we will see how it how it is resolved slightly later. But so we can assume that we have got this mutual exclusion, this atomicity somehow ensured, this acquired and release state functions, so they are implemented by hardware atomic instructions.

So, usage is like this. So, if you are writing a solution to the critical section problem using this mutex locks. So, it will be doing like this it will first acquire the lock if the lock acquires, so acquire lock. If this statement is true; that means, the process has not been put into infinite loop waiting in the while statement. So, it entered into the critical section, after finishing the critical section it will execute release lock, so the lock will be release then it will go to the remainder section then while true. So, this way this process will be getting the mutex lock. And solution requires busy waiting definitely because against that variable available a process may be waiting.

So, this particular this type of locks so, they have given a technical name called spinlock, ok. So, they are this is given a technical name called spinlock. So, there are many solutions that we will see that are doing this busy waiting. So, they are given the generic name called spinlock. So, that is fine.

(Refer Slide Time: 16:33)

Semaphores

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for processes to synchronize their activities.
- Semaphore S – integer variable
- Can only be accessed via two indivisible (atomic) operations
 - **wait()** and **signal()**
 - Originally called **P()** and **V()**
- Definition of the **wait()** operation

```
wait(S) {
    while (S <= 0)
        ; // busy wait
    S = S - 1;
}
```
- Definition of the **signal()** operation

```
signal(S) {
    S = S + 1;
}
```

Handwritten diagram:
Semaphore
↓
Data Type
↙ ↘
Value that it can contain Operations

Now, next we will be looking into our data structure type based solution or data type variable based solution which are known as semaphores. So, semaphores are synchronization tools that provide more sophisticated ways for processes to synchronize their activities. So, we have got a semaphore S as a variable. So, when I whenever I say that, so for example, semaphore, so this is this is nothing, but a data type. So, when I whenever I say semaphore, so in terms of programming language it is nothing, but a data type.

Now, what do you mean by a data type? So, data type when I want to specify I have to tell two things one thing is that the type whether value that it can contain, value that it can contain and the second thing is the operations that we can do on this data type. So, these are the two things that I need to specify on whenever I am talking about a data type. For example, if I say integer is a data type now any variable which is define to be of data type integer, so it can hold all the positive and negative integer values in terms of this if you think in terms of mathematical definition. So, whatever it is an integer, so it will be having that.

Then operations on integer, so I can say I have got this addition, subtraction, multiplication, integer, division, absolute. So, like that, so these are the operations that I can do on the integer data type. Whenever I say a string as a data type, so as a content I have to say what a variable that is defined as of type string. So, I have to say its content

is nothing, but a sequence of characters terminated by a null character or 0, so that is the content. And as far as operations are concerned I can do concatenation of two strings, I can take a substring, I can find a position of a character in a string. So, these are the operations that I can do on the string data type.

So, whenever we are talking about a data type we have to tell two things one is the value that it can contain and the second thing is the operations that we can do on that data type. So, similarly whenever I say semaphore is a data type, so semaphore I have to tell what is the value it can contain. So, value it can contain is integer. So, it can contain only integer. And the operations that you can do, so there are only two operations wait and signal. So, when this semaphores were first introduced. So, they were given the name P and V, P for wait and V for signal. After that been later time, so the names have change, so they have become wait and signal.

So, in some old literature you can still find reference to this P and V operations understanding fully well that they are nothing, but the wait and signal operation that we are going to discuss. So, the wait operation is like this. So, wait on a semaphore s. So, if the value of the semaphore is less or equal 0 then the process which has given a call to this wait statement, so it is put into an infinite loop. So, while S less or equal to 0, so it is do nothing, so it is waiting in the loop and if the value of S is greater than 0 then S is decremented by 1 and the process returns. So, this is the definition of wait statement.

On the other hand, the signal operation is simple signal will just increment the value of S to S plus 1. Now, you see that as far as the operations are concern, so they can be done by simple integer variables also. So, I can have S an integer variable and write down these pieces of codes to have this wait and signal statements written for the integer variable. But the point is that this wait and signal they are indivisible atomic operation. So, you, so it ensures that when a processor executes this statement this wait statement, so the processor cannot be interrupted.

So, this OS designer should ensure that any process which has given call to this wait statement so the execution cannot stop in between. So, it will be it will be going on doing these things, so it cannot be interrupted. So, it cannot be that it has it has modified half of S and then it called descheduled, so nothing like that can happen. Similarly, signal also,

so this is also atomics. So, these operations wait and signal they are atomic in nature. So, that is the good thing about this wait and signal statements.

(Refer Slide Time: 21:17)

Semaphore Usage

Can solve various synchronization problems

- A solution to the CS problem.
 - Create a semaphore "synch" initialized to 1

```
wait (synch)
CS
_signal (synch);
```

- Consider P_1 and P_2 that require code segment S_1 to happen before code segment S_2
 - Create a semaphore "synch" initialized to 0

P1:

```
S1;
signal (synch);
```

P2:

```
wait (synch); ←
S2;
```

The slide contains two diagrams. The first diagram shows a semaphore 'synch' with a value of 1. Two processes, P1 and P2, are shown with arrows pointing towards it. P1's arrow is labeled '+0' and P2's is labeled '-0', indicating they are both waiting to enter a critical section. The second diagram shows P1 inside a critical section labeled S1. P2 is outside, with an arrow pointing to a semaphore 'synch' that has a value of 0. This indicates P2 is blocked because the semaphore is held by P1.

Now, using this wait and signal, so we can we can have this critical section problems solved very efficiently. You can solve various synchronization problems. For example, solution to the critical section problem is like this. Create a semaphore synch initialize to 1 and then wait on synch. So, if the semaphore is, so if I have got two processes say P 1 and P 2, and then P 1 and P 2 both of them have got the structure like this. So, process P 1 has got this structure and process P 2 also has got this structure.

Now, process P 1, so synch variable. So, it is initialized to 1, so process P 1 comes it executes this wait synch statement. It finds that synch variable is 1, so it is it is greater than 0. So, it will be it will not be waiting in that while loop it will enter, it will decrement the value of synch to be equal to 0 now and it will enter into the critical section.

Now, while P 1 is in critical section, if P 2 comes then also it will find that P 2 is P 2 will find that this synch variable is already equal to 0. So, it cannot do anything. So, it will just be waiting there. And after sometime P 1 will finish off and P 1 will execute this signal statement and when the signal is executed, so this value will again become equal to 1. And next time P 2 comes to P 2 gets scheduled, so it will check that while loop again and it will find that the content of synch is equal to 1. So, it will decrement the

value of synch by 1, if the same value will become 0 and it will enter into the critical section. So, this way P 1 and P 2, so both can execute.

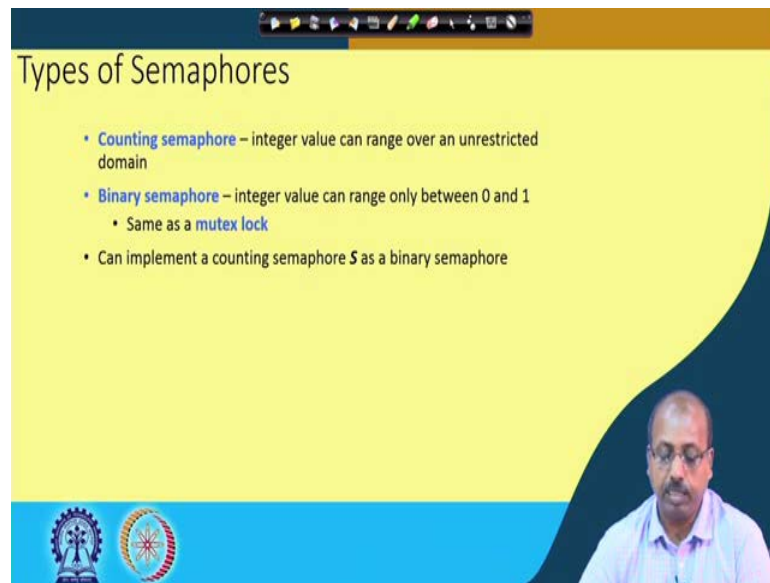
So, this is a these are some typical situation that we have got next. So, consider P 1 and P 2 that require code segment S 1 to happen before code segment S 2. So, so it is like this, so this process P 1 will there is a process synchronization. So, first S 1 will be executed. So, it is like this. So, I have got two code segment in process P 1. So, this is the structure of P 1, so it has got a piece of code S 1. And this process P 2 it has got some code S 2 in it. And the requirement is that S 1 P 1 must execute the up to the end of S 1 then only P 2 should execute S 2.

So, due to this relative speed of P 1 and P 2, so they can be they P 1 may be faster than the other, so there is or 1 may be very slow than the other, but what is required is that P 2 should not execute this code of S 2 before P 1 has finished S 1. So, what can happen. So, P 2 may start first, so P 2 execute this nondependent portion, but when it comes to this point, so it has to wait for P 1 to reach this point. So, how it can be done?.

So, we have got the synch semaphore which is initialized to 0. So, if P 2 comes first, so P 2 execute this much, but when it comes to this point it execute this wait statement and now this is the synch variable is equal to 0. So, P 2 has to wait at this point. So, P 2 cannot proceed further. So, P 2 is waiting there. So, after sometime P 1 will be scheduled by the by the scheduler. So, P 1 will start executing, so it will execute and so there is no barrier here. So, it comes up to this point and after finishing S 1. So, at this point we have got this signal statement.

So, now the signal is done. So, synch variable becomes equal to 1. So, next time P 2 come, so P 2 was waiting at this point, ok. So, it will execute that wait statement and it will find that that the busy loop will be over for P 2 now and it will be entering into this code S 2. So, in this way we can do synchronization between two processes P 1 and P 2. So, P 1 will be finishing off S 1 and then only P 2 will be starting to execute this S 2. So, we can. So, this way many synchronization problems can be solved using this semaphore type of variables.

(Refer Slide Time: 25:29)



The slide is titled "Types of Semaphores" and features a yellow background with a dark blue curved shape on the right side. At the top, there is a navigation bar with various icons. The main content consists of three bullet points:

- **Counting semaphore** – integer value can range over an unrestricted domain
- **Binary semaphore** – integer value can range only between 0 and 1
 - Same as a **mutex lock**
- Can implement a counting semaphore S as a binary semaphore

In the bottom right corner, there is a video inset showing a man with glasses and a light blue shirt speaking. At the bottom left, there are two circular logos: one with a gear and a person, and another with a sun-like pattern.

Now, there are two types of semaphores, one is known as a counting semaphore another is known as a binary semaphore. A counting semaphore integer value can range over an unrestricted domain. So, counting semaphore is the content the value of the variable can be any positive or negative integer, the other end there is a binary semaphore which is a restricted version of counting semaphore. Here the integer value can range only between 0 and 1. So, this is same as the mutex lock.

So, this is a simplified version, but I should tell you that we can implement a counting semaphore as a binary semaphore. So, we can, so we can implement a counting semaphore as a binary semaphore and a binary semaphore as a counting semaphore. So, both are possible.

(Refer Slide Time: 26:16)

Counting Semaphores Example

- Allow at most two process to execute in the CS.
- Create a semaphore "synch" initialized to 2

wait(synch)
CS
signal(synch)

Server 1, Server 2

P₁, P₂, P₃

So, counting semaphore example. So, suppose we want to do this type of thing like that is we want to allow at most two processes to execute in their critical section. So, we are a bit relaxed. So, the original version of the critical section problem it set that only one process is allowed into the critical section, no. Suppose I have got a situation where there are two servers, ok. So, there are two servers, server 1 and server 2. So, this is server 1 and this is server 2.

Now, we want that at, so there is an entry gate and through this gate at most two processes can be there inside the critical section. So, the previously formulated solution that we had so, that cannot solve this problem because that was allowing only one process to enter into the critical section. Now, I want to allow at most two processes.

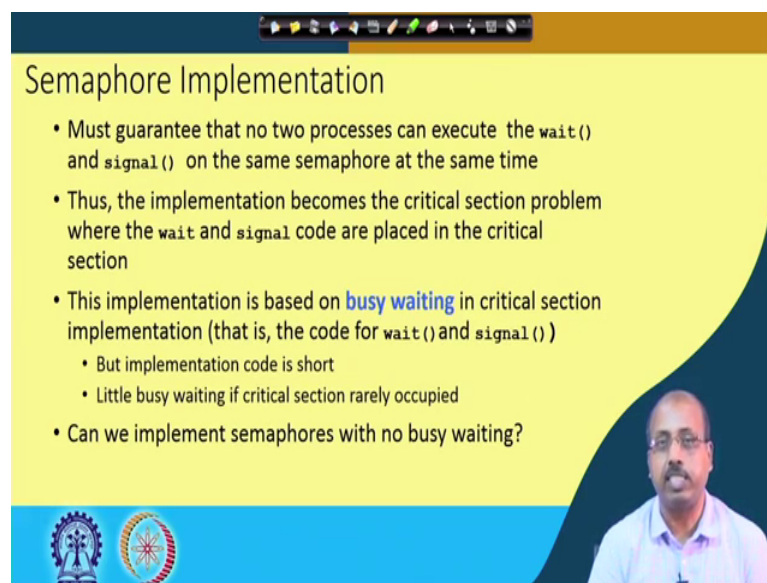
So, situation is the situation can be solved very easily by having this synch semaphore variable initialized to 2. So, if synch happens to be a counting semaphore then it can take up any positive or negative integer values. So, initialize synch to be equal to 2. So, it executes this wait any process trying to enter into the critical section. So, it will execute this piece of code. So, it will wait on synch the first process P 1 comes. So, the synch variable it was equal to 2, so it will decrement to 1 and this P 1 will enter into the critical section.

Another process P 2 comes. So, P 2 will execute this wait on synch, so synch variable will become 0 and then P 2 will also enter into the critical section. But now if a process P

3 comes, so it will execute this wait, but the semaphore variable has already become equal to 0, so P 3 will be made to wait at this point. So, P 3 P 3 will not be entering into the server room; so, P 1 and P 2, so they will be finishing. After sometime suppose P 1 finishes, so it will execute the signal statement. So, this variable will become equal to 1.

Now, if P 3 checks again. So, P 3 we will find that this semaphore value has become equal to 1, so it will be decremented to 0 and it will enter into the critical section. That is it will enter into the service room. So, this way I can have one solution to this to allow more than one processes into critical section say we allow at most, so it ensure that at most two processes are allowed. So, if you have got in general at most k processes are allowed into the critical section. So, you have to initialize this synch variable to be equal to k. So, that can be done.

(Refer Slide Time: 28:54)



The slide is titled "Semaphore Implementation" and features a list of bullet points. The text is as follows:

- Must guarantee that no two processes can execute the `wait()` and `signal()` on the same semaphore at the same time
- Thus, the implementation becomes the critical section problem where the `wait` and `signal` code are placed in the critical section
- This implementation is based on **busy waiting** in critical section implementation (that is, the code for `wait()` and `signal()`)
 - But implementation code is short
 - Little busy waiting if critical section rarely occupied
- Can we implement semaphores with no busy waiting?

The slide also includes a video inset of a man speaking in the bottom right corner and two logos in the bottom left corner.

So, next we will see how this semaphore can be implemented. Now, the problem is that semaphore I have said that this is a data type. So, any operating system designer wants to implement semaphore they have to do it by means of the underlying hardware facilities that are provided. So, it must guarantee that no two processes can execute the wait and signal statements on the same semaphore at the same time. So, I can have a number of semaphore variables, so I have to ensure that only one process is allowed to do a wait or signal on a on a particular semaphore. Now, they can do on different semaphores but on a particular semaphore only one of them.

So, implementation becomes a critical section problem. So, this is you see that for solving critical section problem we are going to another critical section problem. So, if the underline thing is that the semaphore access it has to be mutually exclusive, so that becomes a critical section problem. So, this wait and signal code are placed in the critical section. This implementation is based on busy waiting in the critical section implementation. So, that it is basically that implementation that we have got using this busy waiting sort of thing checking the lock and all. So, implementation code is short, but little busy waiting if critical section is really occupied. So, if critical section is short then definitely this will be solution.

But can we have a solution you can have a better implementation for semaphore without busy waiting. So, busy waiting is not a good situation because the processor time is unnecessarily wasted. The process is not allowed to enter into the critical section, but I am using the processor time. So, a better approach can be that the process is made to wait somewhere else and the processor is doing something more fruitful.

So, without doing this busy waiting, can we solve this implementation problem of the semaphore so, that we will see in the next class.