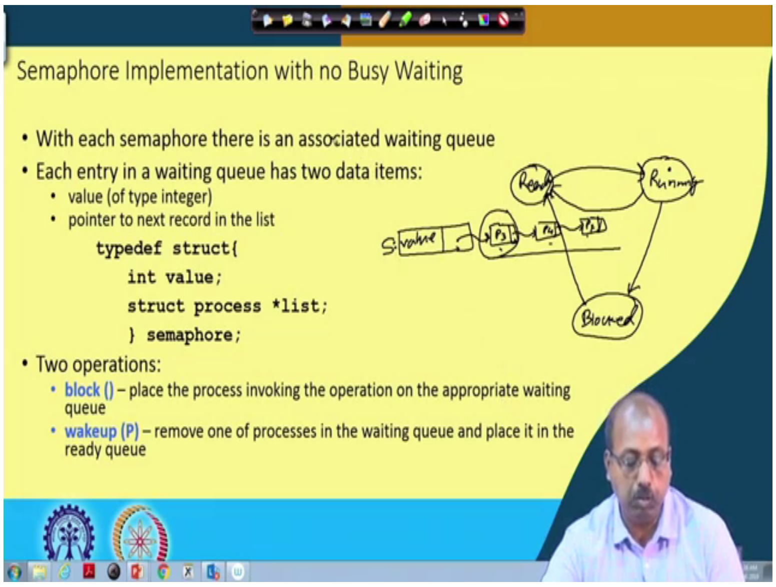
Operating System Fundamentals

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

Lecture - 33
Process Synchronization (Contd.)

(Refer Slide Time: 00:27)



Semaphore Implementation with no Busy Waiting

- With each semaphore there is an associated waiting queue
- Each entry in a waiting queue has two data items:
  - value (of type integer)
  - pointer to next record in the list

```
typedef struct{
    int value;
    struct process *list;
    } semaphore;
```

- Two operations:
  - block () – place the process invoking the operation on the appropriate waiting queue
  - wakeup (P) – remove one of processes in the waiting queue and place it in the ready queue

So, the basic problem that we have seen in the last class, so busy waiting based semaphore implementation. It was that a process who is not able to enter into the critical section, because the semaphore variable value be being less or equal 0. So, it was waiting in a loop there and checking the value of the variable. So, whenever the process is scheduled, so it actually checks that variable value and it after so it loops there. And after some time when the time slices for the process expire, so it is put back into the ready queue.

So, if you remember that state transition behaviour of the processes, so the from the ready state, so the process is put into the running state. And after going to the running state what the process is doing it is simply checking whether the semaphore is available or not. So, if it is not available, so it is waiting in the it is executing that busy wait type of while loop. After some time its time slice expires and it comes back to the ready state. So, that is the way the entire CPU time slice that is given to the process is wasted waiting for the semaphore variable to become greater than 0.

Now, this particular solution that we are going to see now that does not use this busy waiting facility busy waiting feature. So, what we do with every semaphore we have got an associated waiting queue. So, we have got a semaphore say S. So, S has got two component in it; one is the value of it, one component is the value of it and the other component is a list, other component is a list. So, this actually holds a list of processes that are waiting for the semaphore variable to become equal to become greater than equal to 0 ok.

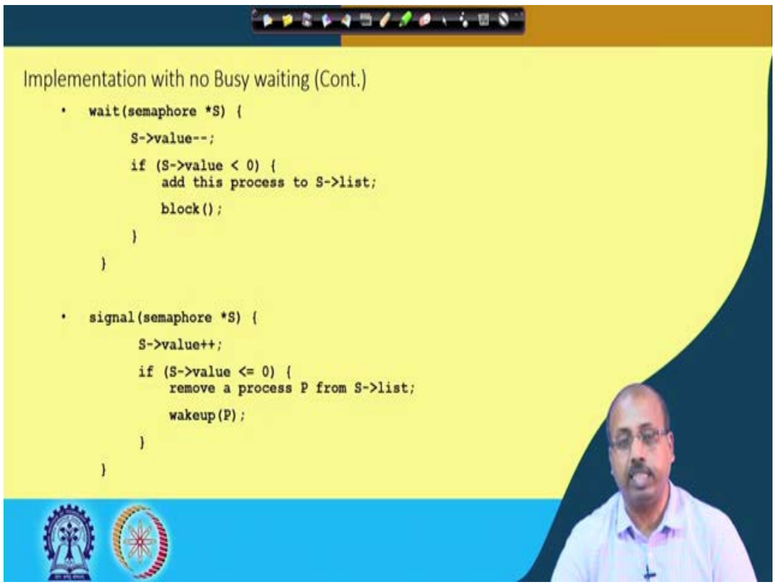
So, suppose at present there are three processes p 3, p 4 and p 7 who are waiting for the semaphore variable to become more than 0, so they are put into this list. So, this is how will be implementing it. So, with each semaphore, there is an associated waiting queue. So, this is the waiting queue. So, p 3, p 4, and p 7, they are waiting for the semaphore to be become greater than 0.

Each entry in a waiting queue has two data item, the value and the pointer to the next record in the list. So, value is so this value is there so of type integer and this process lists. So, this process list is as we have seen; it is the list of process that is the. So, this whole thing defines a semaphore data types. So, any semaphore S defined to be of type semaphore, so it has got a structure like this.

There are two operations that will be using one is called block that will place the process invoking the operation on the appropriate waiting queue. So, if your process is calls this block statement, so it is put into and wait it is put into a queue and the process makes a transition from the running to the blocked state. The process makes a transition from running to the block state, and it is put into one the queue for the of the corresponding semaphore.

And there is a wakeup; wakeup P, so this will remove one process in the waiting queue and place it in the ready queue. So, anything that executes a wakeup, so it will take up one process from this ready from this block state, either p 3, p 4 or p 7. Based on some policy, one of them will be picked up; maybe first come first serve whichever process is there so like pc p 3 came first to this the so that process is wakeup or maybe they have got some priority, so whichever process has got highest priority is wakeup that wakes up, and that is going back to the ready state. So, we can have this thing. So, remove one process in the waiting queue and place it in the ready queue. So, you have got two new operations block and wakeup.

(Refer Slide Time: 04:19)



Implementation with no Busy waiting (Cont.)

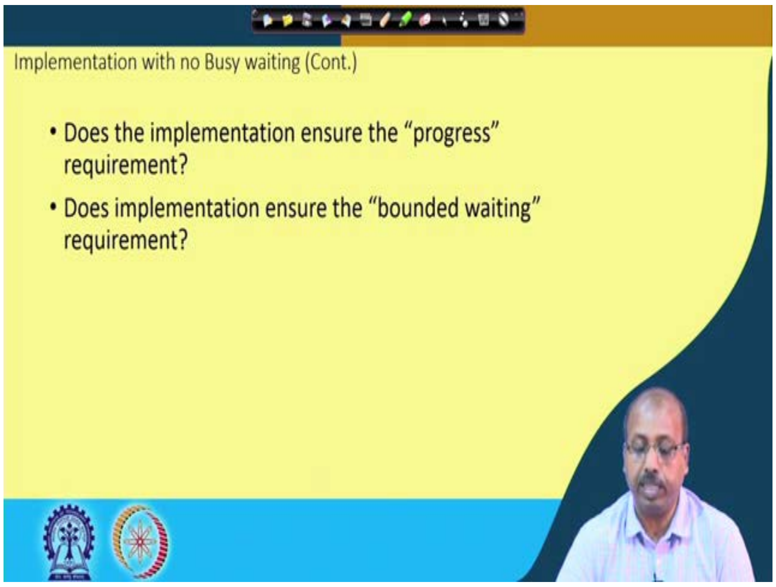
```
• wait(semaphore *S) {
        S->value--;
        if (S->value < 0) {
            add this process to S->list;
            block();
        }
  }

• signal(semaphore *S) {
        S->value++;
        if (S->value <= 0) {
            remove a process P from S->list;
            wakeup(P);
        }
  }
```

Now, using these we can implement the semaphore like this with no busy waiting. So, this semaphore you see that we have got the semaphore has got a value field. So, the wait operation is like this. It decrements the value of this the value field, so S value is decremented by 1. And if the value becomes less than 0, then add this process to S list. So, this process is added to the list of processes blocked on that particular semaphore. Then the process gives the call to the block system call. So, process is get will get block. And it will get block for an event that the semaphore value is greater than 0.

And on the signal operation, so it is like this. First the value of the semaphore is incremented by 1 and if the value becomes less or if the value was less or equal 0, that means, there are some processes waiting in the ready in the list corresponding to the semaphore. So, we will remove one such process from S list. So, nothing is said about which process is removed here.

So, you can have more complicated selection process here so based on priority or some other scheduling criteria. So, somehow a process P is selected from the list of waiting processes for the semaphore that is S list and that process a wakeup call is given to that process, so that process makes a transition from the block state to the ready state. So, in this way, if we do it, then of course, there is no busy waiting. So, this is the standard way of doing this semaphore implementation.

(Refer Slide Time: 05:54)

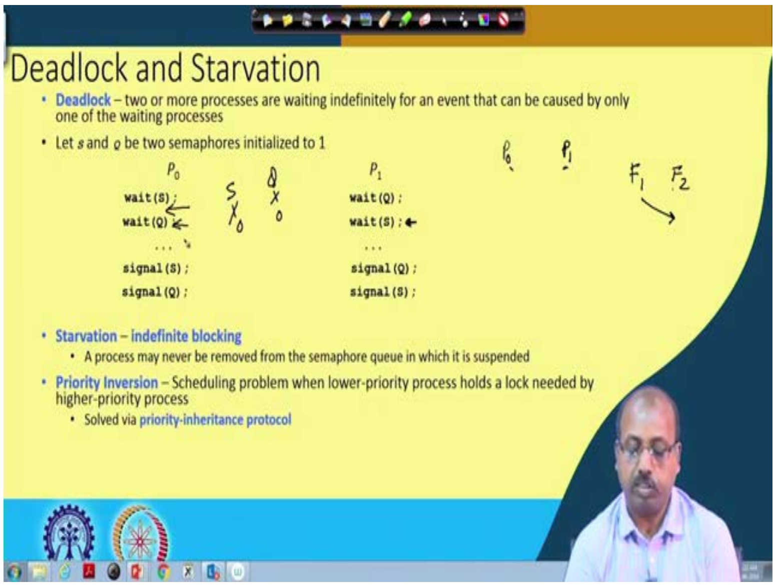


Implementation with no Busy waiting (Cont.)

- Does the implementation ensure the "progress" requirement?
- Does implementation ensure the "bounded waiting" requirement?

So, does this implementation ensure progress requirement? So and does this implementation ensure bound bounded wait requirement? So of course, bounded wait, it is not; it is not ensured, because which process will be picked up, so that is not like we have said that remove a process P from S list, but whether a process they are in the ready queue in the list of waiting processes for a semaphore, so whether it gets a chance to be wakeup, so that is not fixed. So the as a result this bounded wait is definitely not satisfied. And progress requirement is that how many time a process may be denied for entry, so that is also not ensured.

(Refer Slide Time: 06:39)



Deadlock and Starvation

- **Deadlock** – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let *S* and *Q* be two semaphores initialized to 1

```
P0                    P1
wait(S);              wait(Q);
wait(Q);              wait(S);
...                   ...
signal(S);            signal(Q);
signal(Q);            signal(S);
```

- **Starvation – indefinite blocking**
  - A process may never be removed from the semaphore queue in which it is suspended
- **Priority Inversion** – Scheduling problem when lower-priority process holds a lock needed by higher-priority process
  - Solved via **priority-inheritance protocol**

So, there may be other typical problem, one is the problem of deadlock. So two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes, so that is a situation of deadlock. So, we will have a more detailed discussion on deadlock later. So, what it means is that I have got two process say P 1, and P 0 and P 1. So, this P 0 and P 1, so they are waiting for some event to proceed, but P 0 is waiting for some event that can take place only P 1 execute properly and P 1 is waiting for some for some event that can happen only P 0 executes properly. So, there is a deadlock.

It is like say two people. So, they are both they are two friends F 1 and F 2. So F 1 telling F 2 that I will go if you go first and F 2 says that I will go, if you go first. So, that way so both of them are asking for the friend to go first. So, as a result there is a deadlock, none of them can proceed.

So, this is the deadlock situation and that can happen in the operating system also in the process execution. So, this S and Q suppose they are two semaphores both are initialized to one. So, process P 0 executes like this it executes wait on S. So semaphore, so this it executes is wait on S. So, as a result S becomes equal to S and Q. So S was equal to 1, Q was equal to 1. Now, S has become equal to 0.

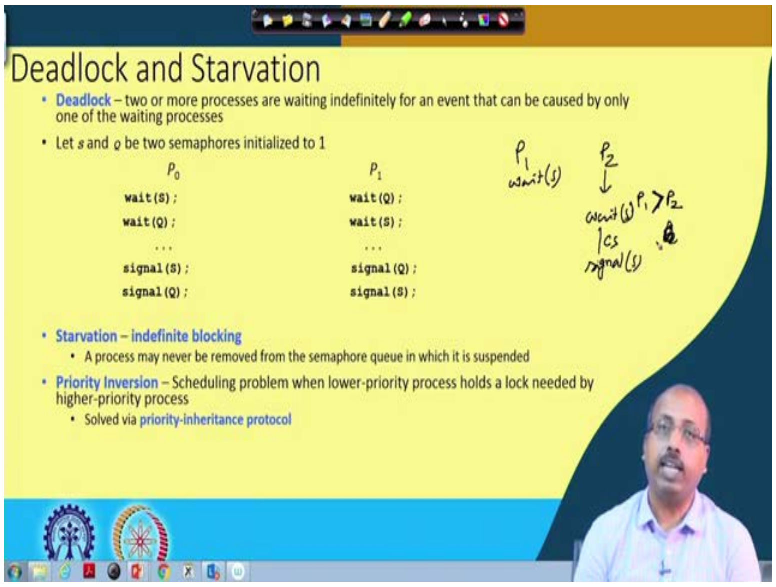
Now, suppose after executing this much, so this the P 0 gets rescheduled and P 1 comes into existence. P 1 executes this wait statement, it becomes Q becomes equal to 0. After that it executes this wait statement wait on S, so S is already 0, so P 1 is made to wait at this point. So, after some time P 0 gets scheduled. So, P 0 executes this wait statement, but it finds that Q is already equal to 0. So, it is made to wait at this point.

So, this way if the processes get scheduled in this fashion that is P 0 get scheduled, it executes the first wait statement then it is rescheduled. P 1 comes, P 1 executes the first wait statement, then what can happen is that this P 0 and P 1 both of them are put into an infinite wait, so that is the deadlock situation. So, none of the processes can proceed. So, this is one problem.

Another problem is the starvation. A process may never be removed from the semaphore queue in which it is suspended. So, as I was telling there is a queue of waiting processes, it may so happen that every time a process another process gives a signal. So, some, one process will be removed from this semaphore list of waiting processes, but one particular process from the list is never selected for to be removed, so that process always waits in the queue of the semaphore. So that is the starvation problem indefinite blocking for the process for this particular process. So this is the problem.

And there is another very interesting problem which is known as priority inversion. So, scheduling problem when lower priority process holds a lock needed by a higher priority process.

(Refer Slide Time: 09:59)



Deadlock and Starvation

- Deadlock – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let *S* and *Q* be two semaphores initialized to 1

```
        P0                    P1
    wait(S);              wait(Q);
    wait(Q);              wait(S);
      ...                   ...
    signal(S);            signal(Q);
    signal(Q);            signal(S);
```

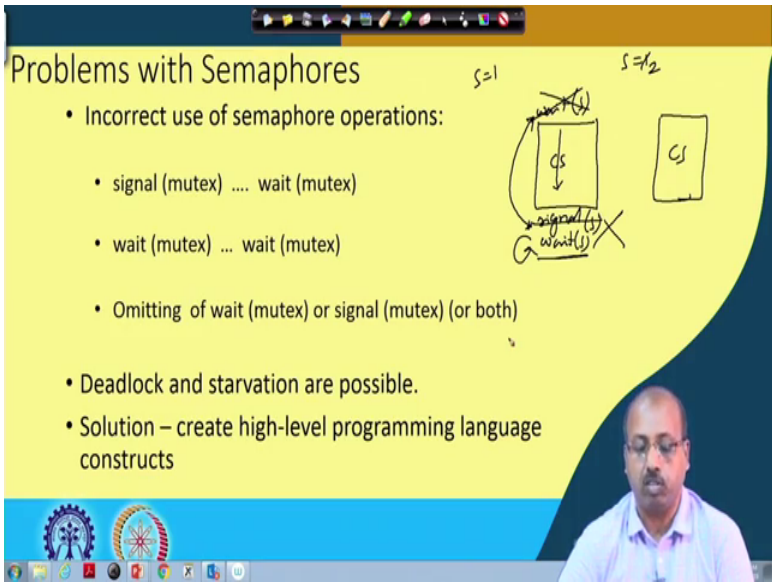
- Starvation – indefinite blocking
  - A process may never be removed from the semaphore queue in which it is suspended
- Priority Inversion – Scheduling problem when lower-priority process holds a lock needed by higher-priority process
  - Solved via priority-inheritance protocol

So, it is like this that suppose I have got two processes P 1 and P 2. And in a priority based scheduling policy maybe P 1 is of higher priority than P 2. But suppose in the system P 2 came first, so it executed a wait on some semaphore, wait on some semaphore S and then after that it was executing the critical section. And in between P 1 has come. So, P 1 is of higher priority, so P 1 should get the chance for execution, but P 1 also needs the semaphore S. So, P 1 also needs the semaphore S.

Now, what will happen since P 2 is holding the semaphore S. So, P 1 has to wait for P 2 to finish off the critical section and release S, so that it can proceed. So, it has to wait for that. So, there is an inversion in the priority of P 1 and P 2. So, though originally P 1 is of higher priority than P 2, but in reality what happens is that P 2 becomes of higher priority than P 1.

So, P 1 has to wait for P 2 to finish off. So, this is known as the problem of priority inversion. And it can be solved using something known as priority inheritance protocol. So, P 1, so P 2 inherits the priority of P 1, so that way we can make we can solve this priority inversion problem. So, not go into much detail of this, because it is more of some real time systems and also if we looking into that then we can have a look at that.

(Refer Slide Time: 11:29)


Problems with Semaphores

- Incorrect use of semaphore operations:
  - signal (mutex) .... wait (mutex)
  - wait (mutex) ... wait (mutex)
  - Omitting of wait (mutex) or signal (mutex) (or both)
- Deadlock and starvation are possible.
- Solution – create high-level programming language constructs

So, there are some problems with semaphores like incorrect use of semaphore operation, like the signal mutex and wait mutex. So, this is one particular situation. Like normally thus the situation is like this that if I have got a critical section; critical section code, then

I should put a wait statement at the beginning and the signal at the end, so that is the standard solution wait on S and signal S.

Now, if by mistake, we exchange these two this wait and signal are exchanged. So, the semaphore S was initially equal to 1. So, this is signal, so, this value becomes equal to 2 and then the critical section is executed. Now, another process when it comes, so this will also enter into its critical section because the semaphore variable is more than 1, more than 0, so that will also enter into the critical section.

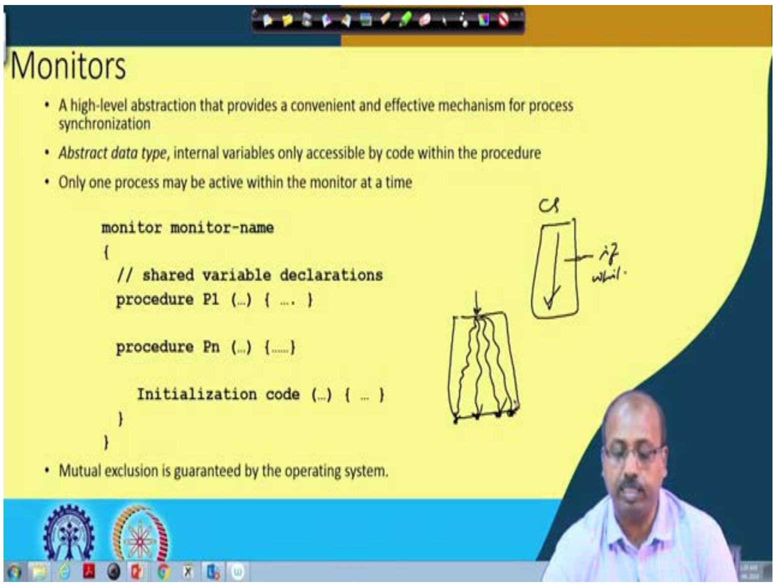
So, the critical mutual exclusion will not be satisfied. So, it is a very common mistake while writing the programs in our by an application programmer. So, this wait and signal statement positions, so they are interchanged. So, as a result the solution does not work properly.

Then sometimes what happens is that this instead of putting a signal, we put another wait statement here. So, wait followed by wait. So, this situation is more severe, because say the semaphore S was equal to 1. Now, this process has entered into the critical section. After that it makes wait S instead of signal S, the programmer has written wait S. So, this process is made to wait here, but more seriously no other process can enter into their critical section. So, nobody is there in their critical sections, no processes for no process is executing in its critical section at the same time, nobody can enter into the critical section also see if you do this type of mistake.

Another way another common very common mistake that we make is omitting the wait or signal or both. Maybe sometimes we forget to put this wait statement, somebody forgets to put the signal statement or maybe both, so that way all these leads to incorrect solution to the critical section problem or any other synchronization problem that is a semaphore. So, the semaphore provides a very high-level primitive, but at the same time we have to be very very careful while trying to implement or trying to use it in our programs.

Deadlock and starvation they are possible because of this thing that incorrect usage of this semaphores and all. And solution is to create some high-level programming language construct. So, we may go for even higher level programming language construct that can solve this type of mistakes. So, these type of mistakes cannot occur with them.

(Refer Slide Time: 14:22)



Monitors

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- *Abstract data type*, internal variables only accessible by code within the procedure
- Only one process may be active within the monitor at a time

```
monitor monitor-name
{
  // shared variable declarations
  procedure P1 (…) { …. }

  procedure Pn (…) {……}

    Initialization code (…) { … }
  }
}
```

- Mutual exclusion is guaranteed by the operating system.

So, we will see into some of them. And one such a high-level construct is the monitor. So, this actually hides many of the implementation details from the application programmer. So, application programmer if it uses the monitor as the high-level primitive, then the programmer is relieved of this wait signal sequence and also. Programmer need not to be bothered about whether I have put this wait signal and conditions. Apparently it seems it is very easy like I have got a block of critical sections statements, then I will put wait at the beginning and signal at the end.

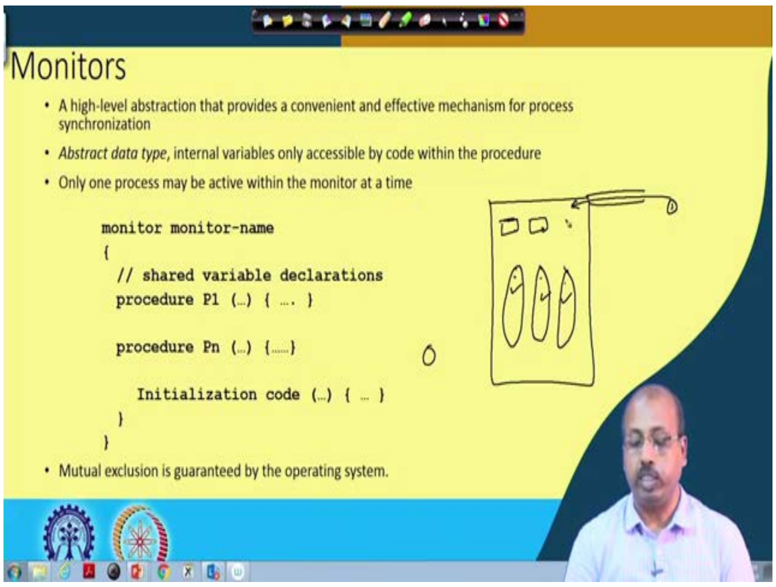
But many times what happens is that within this critical section within this critical section also the critical section code is this critical section code so instead of being a state line code, so there may be branching. So, maybe there is an if statement somewhere inside the critical section, there is a while statement. As a result, it may so happen that there are several execution traces through the critical section. So, there is only one entry point, but there may be many exit points. So, like that if you trace the route of the program or the path followed by the program, you may find that there are several exit ways for the critical section.

Now, at every point I have to ensure at each of these exit point I have to ensure that this wait signal sequencing has been done properly. Maybe due to the complex logic of the program it may so happen that for some path this balancing is not done properly. So, as a result, when the program executes and if it by chance follows that particular path, then this it can lead to an incorrect situation for the synchronization problem.

So, if I have a higher level primitive, then the programmer is relieved of this type of difficulties. So, this is a high-level monitor is one such high-level abstraction that will provide a convenient and effective mechanism for process synchronization.

So, it is an abstract data type, internal variables only accessible by code within the procedure. So, this is basically an object oriented structure you can say. The internal variables can be accessed only by the procedure which are inside the monitor. Only one process may be active within the monitor at a time. So, operating system itself will ensure that only one process will be inside the monitored.

(Refer Slide Time: 17:05)



Monitors

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- *Abstract data type*, internal variables only accessible by code within the procedure
- Only one process may be active within the monitor at a time

```
monitor monitor-name
{
  // shared variable declarations
  procedure P1 (…) { …. }

  procedure Pn (…) {……}

    Initialization code (…) { … }
  }
}
```

- Mutual exclusion is guaranteed by the operating system.

So, monitor you can as you can think of a monitor as if this is a so you can think of a monitor as if there is a bounding box and there is an entry gate. So, at through this entry gate, only one process can come into the monitor at a time. So, mutual exclusion is definitely satisfied, because since there is only one process in the inside the monitor, so there cannot be simultaneous access to more than one procedure.

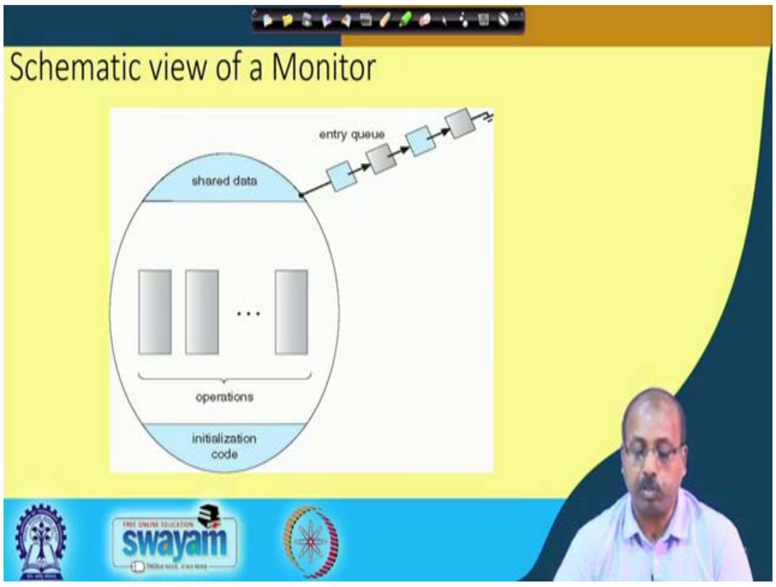
And inside this I have got several access routines. So, there are several access routines. And once the process enters into the critical section into the monitor, so it can use one of these axis routines and whatever variables that you have in the monitor, so they can be accessed only by these routines. So, routines which are residing outside the monitor, they cannot access these particular variables.

So, as a result, many of the issues in synchronization are resolved ok. So, we do not need to be bothered about these incorrect scheduling of these processes and all because of this thing. So, this type of object oriented framework has been provided in monitor. And using them we can solve many synchronization problems. So, this is an abstract data type, internal variables only accessible by code within the procedure. And only one process may be active within the monitor at a time.

So, some monitors, so this is a typical declaration of a monitor. So, the name of the monitor, so its type is monitor, the data type is monitor and the name of the monitor is given. And within that o can have shared variable declarations, I can have a number of procedures beyond P 1, P 2 up to P n and there is an initialization code. So, whenever this monitor is created, this initialization code is executed.

And then after that, so you from the from any outside process, it can call one of these procedures this access routines P 1 to P n for getting some job done. And mutual exclusion is guaranteed by the operating system. So, this monitor boundary this mutual exclusion is guaranteed automatically.

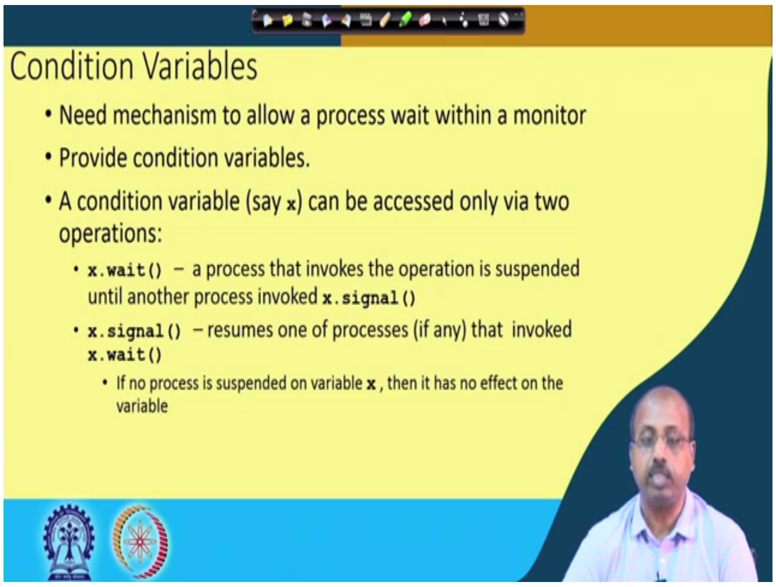
(Refer Slide Time: 19:05)



Schematic view of a Monitor

So, with this type of situation, so you can view it to be a structure like this. As if there is a queue of processes who are waiting to enter into the critical into the monitor. And once it enters into a monitor, so only one of the processes allowed is allowed into the monitor. So, it can call one of these operations; and whichever operation it calls, so it executes that. And then after finishing so it goes out, and then only another one will be allowed into the monitor. So, this way we can have a monitor conceptually it looks like this. So, as an OS designer it is the responsibility to have this monitor implemented by means semaphore or other hardware primitives, so that this facility is provided to the application programmer.

(Refer Slide Time: 19:53)



Condition Variables

- Need mechanism to allow a process wait within a monitor
- Provide condition variables.
- A condition variable (say **x**) can be accessed only via two operations:
  - **x.wait()** – a process that invokes the operation is suspended until another process invoked **x.signal()**
  - **x.signal()** – resumes one of processes (if any) that invoked **x.wait()**
    - If no process is suspended on variable **x**, then it has no effect on the variable

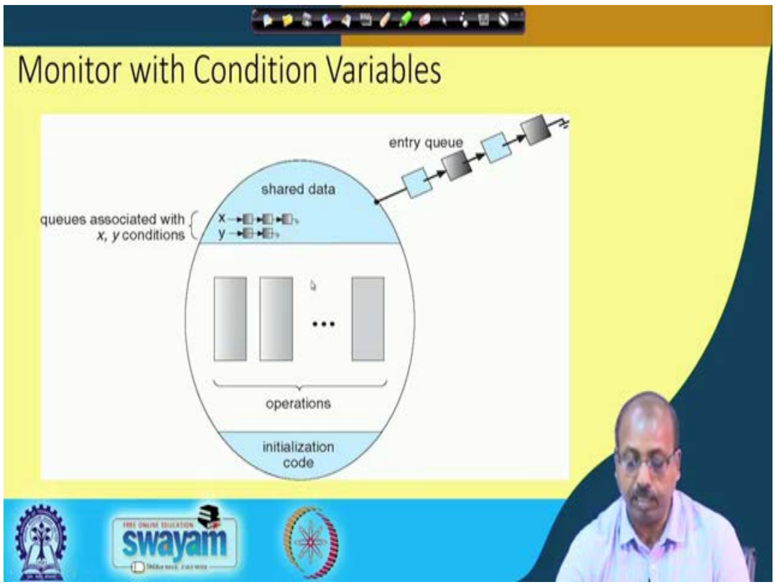
So, there are a number as we have seen in the last diagram, so we have got a queue of waiting processes. So, and this queue implementation is done by means of condition variables. So, condition we need a mechanism by which you can allow a process wait within a monitor. So, this provide, so by wait how to wait, so that is provided by means of this condition variable. So, these condition variables it provided.

So, a condition variable can be accessed only by two operations; one is wait operation another is signal operation. So, x dot wait – a process that invokes this wait operation is suspended until another process has invoked a x dot signal operation. So, one process waiting on a condition to be satisfied, so that is it calls an wait operation on that condition variable x. And that condition another until and unless some other process gives a call to signal so x dot signal, so this process is made to wait.

And x dot signal – it resumes one process if any that invoked x dot wait. So, x dot signal, its execution is like this, that if there are some waiting processes for this condition variable, so that will be invoked. And if no process is suspended on variable x, then it has no effect on the variable. So, if there is no nobody waiting, then there is no such thing.

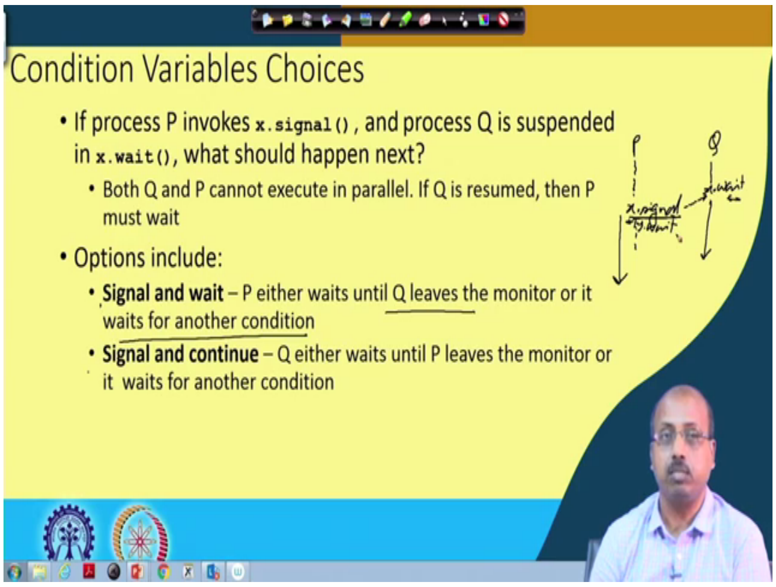
But one thing we must note that any process giving a call to this condition variable x wait on this condition variable x, so it is made to wait outside the monitor boundary. So, it is will be waiting outside. So, when this process another process will execute x dot signal like in this diagram you can see that this may be a condition variable it is waiting for some condition variables. Now, one process which is inside maybe executing this code, after finishing this code it when it goes out, so it may try to see that some condition has become true. So, it wants to invoke some process here which was waiting for that particular condition variable. So, it will be taking that process and that will be made to enter, so that way this whole thing is implemented.

(Refer Slide Time: 22:09)


Monitor with Condition Variables
entry queue
shared data
queues associated with x, y conditions
x
y
operations
initialization code
swayam

So, we have got this condition variables. So, queue is associated with the condition variables, so that is there and we have got other entry level queues. So, this is so sometimes say this is another possible implementation that we have got this wait waiting processes on the condition variable. So, they are made to it inside the monitor. And some implementations that they will make this queue outside the monitor, some implementations they will make it inside the monitor. So, this particular implementation it shows that the between queues associated with condition variables. So, they are put inside the monitor in the shared data area.

(Refer Slide Time: 22:45)



Condition Variables Choices

- If process P invokes `x.signal()`, and process Q is suspended in `x.wait()`, what should happen next?
  - Both Q and P cannot execute in parallel. If Q is resumed, then P must wait
- Options include:
  - **Signal and wait** – P either waits until Q leaves the monitor or it waits for another condition
  - **Signal and continue** – Q either waits until P leaves the monitor or it waits for another condition

Now, condition variable choice like if process P invokes x dot signal and process Q is suspended in x dot wait what should happen? So, the problem is that if both Q and P cannot execute in parallel ok, because see the what I am trying to mean is like this. So, this is the process P ok, this is process P. After some time, it executes one x dot signal statement.

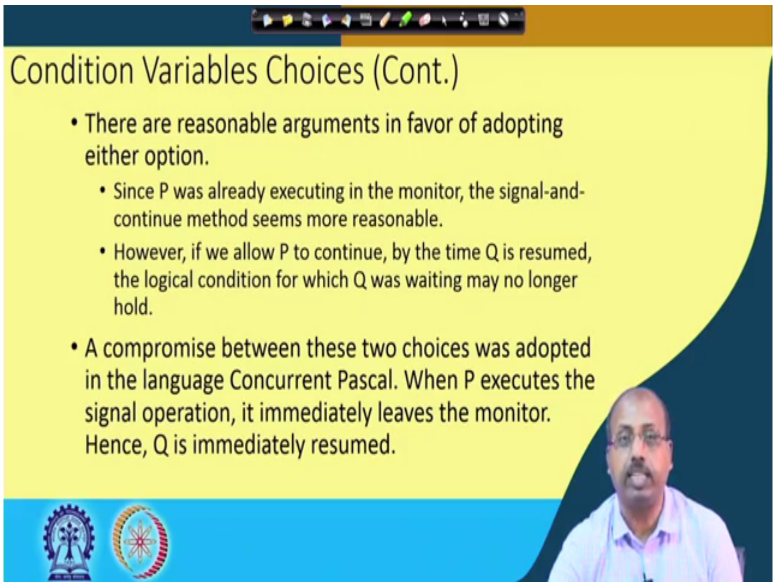
After that it has got some other codes some condition has become true for which it can invoke it wants to give the signal, but there is another process queue which was waiting at this point. So, it has executed in x dot wait and it is waiting for x some other process to give it a signal. Now, this signal has arrived at this point.

Now, the situation is that P is executing at this point, Q is executing at this point and both of them are inside the monitor, so that is actually the violating the monitor condition that at one point of time only one process can be inside the monitor. So, I cannot have both Q and P executed in parallel. So, if Q is resumed, then P must wait ok. So, this must be satisfied. So, options that we have is signal and wait. So, P either waits until Q leaves the monitor or it waits for another condition.

So, what can happen is that after P has executed a signal statement, it must wait at this point till Q finishes then only P will be entering into the monitor again. So, that is one possibility. So, either P waits as a P either waits until Q leaves the monitor or it waits for some other condition or it is waiting on some other variable y dot wait something like that. So, if it is waiting, then it is fine, but as far as executing is concerned, so it cannot execute normal instruction. So, in that case, it has to wait outside the monitor.

Another is that signal and continue. So, this is signal and wait option and this is signal and continue condition. It says that queue either waits until P leaves the monitor or it waits for another condition. So, it may so happen that Q has got the signal that it can proceed, but it will wait for P to be going out of the monitor, then only this Q will enter. So, these are the two options that are available. Signal and wait where the signalling process is made to wait and signal and continue where the signalling process continues, but the signalled process is made to quit. So, this is these are the two situations that can be there. So, both the these implementations are possible, so whereas, designer has to choose one such alternative and accordingly, so that has to be done.

(Refer Slide Time: 25:32)



Condition Variables Choices (Cont.)

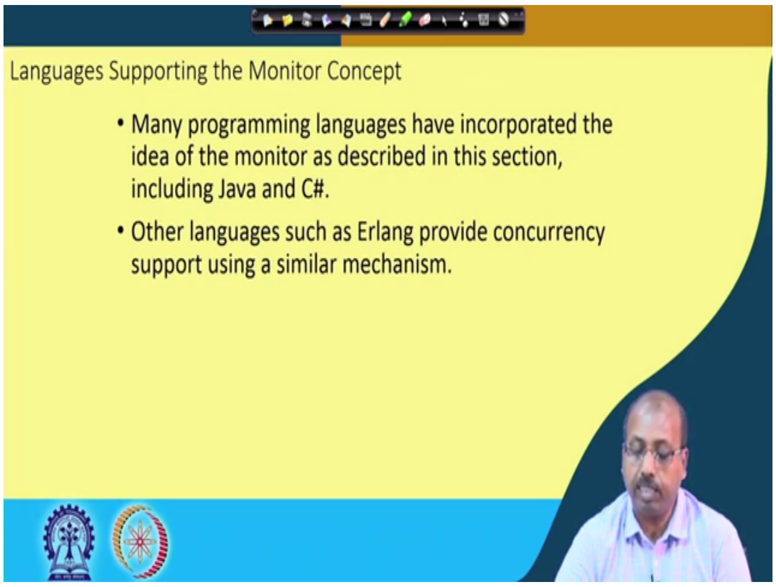
- There are reasonable arguments in favor of adopting either option.
  - Since P was already executing in the monitor, the signal-and-continue method seems more reasonable.
  - However, if we allow P to continue, by the time Q is resumed, the logical condition for which Q was waiting may no longer hold.
- A compromise between these two choices was adopted in the language Concurrent Pascal. When P executes the signal operation, it immediately leaves the monitor. Hence, Q is immediately resumed.

So, there are reasonable argument in favour of adopting either option. So, it is not that one of them is always better than the two, where better than the other. Since P was already executing in the monitor the signal and continue method seems more reasonable because P was executing. So, P has given a signal why should it be made to wait ok, so that way it may be the case that P continues. So, signal and continue appears to be a better method.

However, if we allow P to continue, by the time Q is resume the logical condition for which Q was waiting may no longer hold, so that is the other problem. So, if we allow P to continue, so at that time the condition was true at which Q was signalled; but by the time P finishes and Q is allowed maybe by the time that Q has to that the requirement has done. So, that way the waiting condition may no longer hold. So, this is the other issue. The both of them are there are pros and cons of both the approaches. So, either of them has to be followed and they may lead to some different different situations.

A compromise between these two choices as adopted in language called Concurrent Pascal. When P executes a signal operation, it immediately leaves the monitor, hence Q is immediately resumed ok, so that is the other thing that is adopted in this Concurrent Pascal statement. So, whenever P executes the signal operation, it immediately leaves the monitor. So, Q is immediately resumed. So, this is the thing that I was talking about. Whenever the process P signals, so it is it goes out of the same out of the monitor boundary and this process Q continues.

(Refer Slide Time: 27:13)

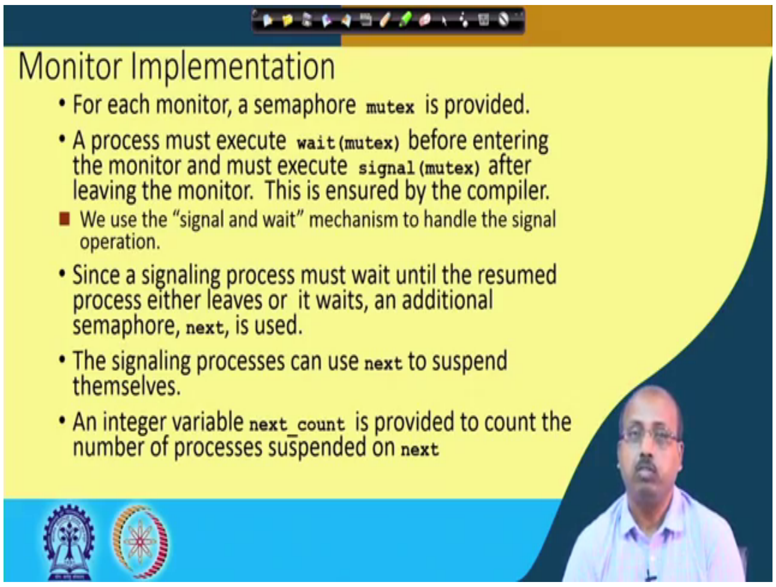


Languages Supporting the Monitor Concept

- Many programming languages have incorporated the idea of the monitor as described in this section, including Java and C#.
- Other languages such as Erlang provide concurrency support using a similar mechanism.

Many programming languages have incorporated the idea of the monitor has described in this section including Java and C sharp. Other languages like Erlang provide concurrency support using a similar mechanism. So, these are the different programming languages they are supported these mechanisms.

(Refer Slide Time: 27:30)



Monitor Implementation

- For each monitor, a semaphore `mutex` is provided.
- A process must execute `wait(mutex)` before entering the monitor and must execute `signal(mutex)` after leaving the monitor. This is ensured by the compiler.
  - We use the "signal and wait" mechanism to handle the signal operation.
- Since a signaling process must wait until the resumed process either leaves or it waits, an additional semaphore, `next`, is used.
- The signaling processes can use `next` to suspend themselves.
- An integer variable `next_count` is provided to count the number of processes suspended on `next`

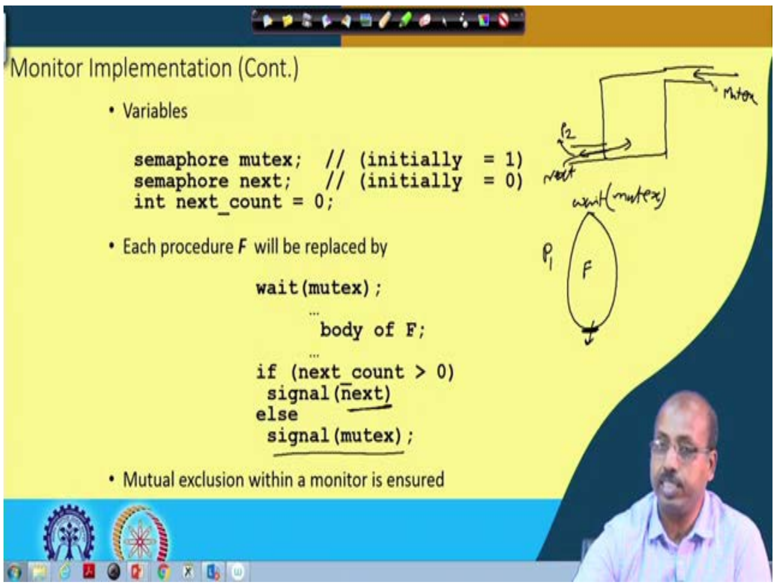
Now, how do we implement a monitor? So, this is very important issue like. So, monitor is not available as such. So, monitor has to be developed by means of some other primitives that are there. So, other primitives whatever we have seen so far is the mutex lock is one primitive, when semaphore we have seen that with the availability of this underlying hardware facility. So, we can implement semaphore. So, can we use them for implementing a monitor?

So, for each monitor we create a semaphore mutex. So, this is required because we want that at one point of time, only one process can be inside the monitor. So, mutual exclusion has to be provided. So, mutex lock is definitely required. So, one semaphore binary semaphore mutex is provided. A process must execute wait mutex before entering the monitor and must execute signal mutex after leaving the monitor. So, the compiler will ensure that this is done. So, compiler, while generating the code for a process giving a call to monitor, it will ensure that whenever a process is trying to access a monitor routine. So, before that routine it the compiler puts a wait mutex and after the call it puts a signal mutex.

And we use this signal and wait mechanism to handle the signal operation. Since a signalling process must wait until the resumed process either leaves or way or it waits, an additional semaphore, next, will be used. So, signalling process has to wait. So, for that it has to wait on a semaphore ok. So, naturally we have to use another semaphore next. And signalling process can use next to suspend themselves.

So, any signalling process, so of the idea that we are following is that any signalling processes will suspend itself, so that the signalled process can continue. So, the signalling process it calls a wait on this semaphore next to get it suspended. And in integer variable next count it counts like how many such processes signalling processes are waiting are getting suspended on the next variable.

(Refer Slide Time: 29:39)



Monitor Implementation (Cont.)

- Variables

```
semaphore mutex;  // (initially  = 1)
semaphore next;   // (initially  = 0)
int next_count = 0;
```

- Each procedure *F* will be replaced by

```
wait(mutex);
     ...
          body of F;
     ...
if (next_count > 0)
  signal(next)
else
  signal(mutex);
```

- Mutual exclusion within a monitor is ensured

So, this is the implementation. So, we have got the semaphore mutex which is initialized to 1; we have got the semaphore mutex next which is initialized to 0 and we have got a next count 0. So, each procedure F, so we know that inside the monitor we have got these access, we have got several functions F. So, on top of F will be will be putting something the it will be putting here at a wait on mutex. So, this wait on mutex is ensured, then the body of F comes.

So, when F finishes, then it has to see whether how to go out ok. So, when before going out, before going out of this F what it what the process does its is that it checks whether there is any process waiting on the next semaphore variable and that count is given by next count. So, in that case, it will be allowing some other process which is waiting on the next semaphore. So, you can look at it like this. So, if this is the monitor, so we have got one entry here; we have got one entry here which is the mutex entry. And there is another temporary exit here which we call the next.

Now, it may so happen that a process is executing this thing, this code F, but it was actually waiting on some condition variable. So, the so some other process has signalled it and some so this is the process P 1 say, some other process P 2 in its execution it has signalled P 1. So, P 2 when it call, so P 2 was made to wait on this next semaphore. So, before finishing this P 1 the this code of F, so P 1 actually checks whether somebody is waiting on the next semaphore or not. If it is waiting on this next semaphore the next count is greater than 0, so it will be signalling that process to enter into the monitor.

And if nobody is waiting here, then only if new process can come via this through this mutex line, so that is the signal mutex is done So, if next count is greater than 0, then it will allow some waiting process on the next Q to enter into the monitor. If it is not so, then it will it may allow some process waiting on the mutex lock here.

So, this way it will ensure that at one point of time either up when a process finishes, it is allowed to when a process finishes, so it is allowed to either go out of it is going out of this monitor, but it either allows one process from this side or allows one process from this side. So, that will ensure that only one processes inside the monitor always. So, that is the mutual exclusion will be satisfied. So, we will continue with this in the next class.