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

**Lecture – 34**
**Process Synchronization (Contd.)**

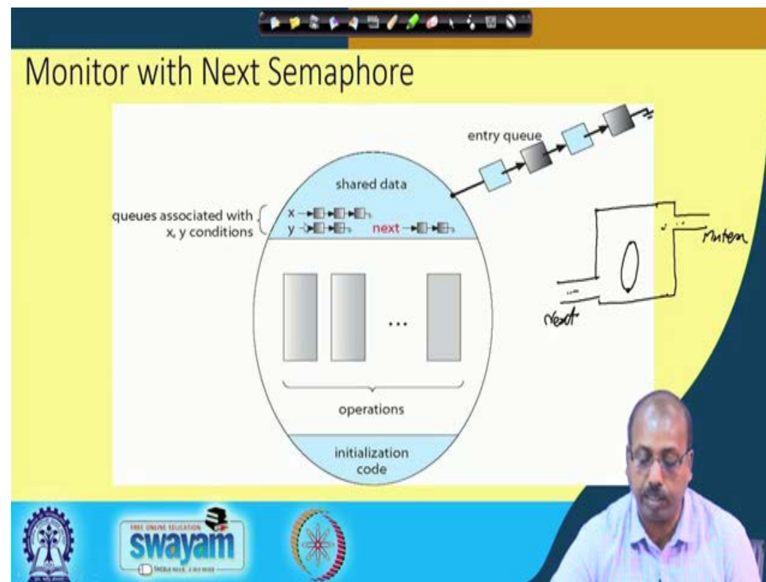
Those the monitor implementation that we were looking into.

(Refer Slide Time: 00:27)

So, we have seen that there are two gates one gate was that mutex gate ok. So, one gate was that mutex gate and the other gate was that next gate ok. So, this is the mutex gate and this is the next gate. And apart from that what we have? So, this part is fine like when a process is finishing; when a process is finishing a routine here. So, this at the end of this routine it is checking whether somebody is waiting at in this line. So, then that is made to enter into this monitor or if nobody is there then it is making somebody from the here to enter into the monitor. So, that part is fine. So, what is left now is the condition variable.

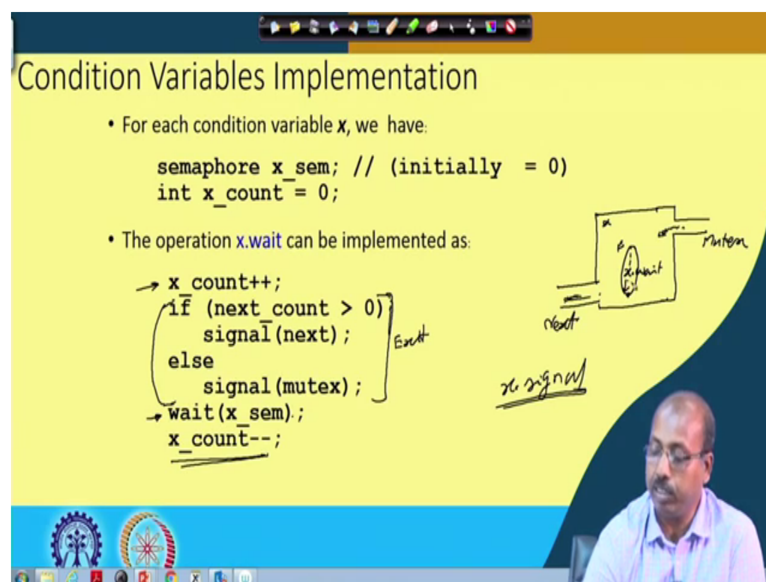
So, next we will see how this condition variables are implemented in this particular structure. So, for that we can implement this condition variable like this.

(Refer Slide Time: 01:32)




So, we have got a number of condition variables. So, this next becomes a queue like this and we have got a number of condition variables two condition variable x and y and there are some queues with that.

(Refer Slide Time: 01:44)




So, how this condition variables are implemented? So, for each condition variable x we have got 2 semaphore; we have got a semaphore x sem which is initialized to 1 and we have got an x count which counts the number of processes, which are waiting on this

particular semaphore. So, any process which is trying to call an wait operation on this condition variable x that is the x dot wait function.

So, it is implemented like this. So we have got this x count x semaphore the condition variable x which has got an associated semaphore x sem and this count. So, first is count is implemented So I have a process in its execution. So, somewhere here so, in some code here it has given a call to x dot wait. So, this is the procedure f and in that there is a call to x dot wait. So, how this x dot wait is implemented?

So, it is implemented like this first of all we increment the number of processes waiting on this particular condition variable x by this x count plus plus and then after implementing what it has to do is that, it has to give a call on this semaphore x sem it has to wait for it on this semaphore x sem. But so basically it has to go out of the monitor it and it has to wait on the some other process for some other process to give it a signal ok.

So, but before going out so, as we have discussed previously before going out it must check whether somebody is waiting in this next queue. So, that process is putting inside. So, that is done by this pair of statement if next count greater than 0 then signal next, otherwise it has to it can signal somebody from this mutex entry signal mutex. So, this is basically the exit part I can say.

So, the process has implemented the account after that it will give a call to wait on x sem. So, then; that means, it is its job is done, but in between the process wants to go out of the monitor and for that before going out it has to execute this piece of code for allowing other process to enter into this monitor. Either some signaling process that is waiting here, or a new process from this side.

Now the processes waiting on this, now after some time what will happen is that some other process will execute a signal statement ok. So some other process which has entered from here or from here so that will execute a signal on this x condition variable; so, that it will execute some x dot signal statement. We will look into the implementation of x dot signal slightly later, but suppose this process wakes up ok. So, when this process wakes up so, somehow this count variable has to be decremented because this x counts. So now this process will enter into the monitors. So, it was waiting at this point. So, it is again entering into the monitor and then it will be continuing from here. So, for that purpose it decrements x count and proceeds from there.

(Refer Slide Time: 05:09)

Condition Variables Implementation (Cont.)

- The operation `x.signal` can be implemented as:

```
if (x_count > 0) {
  next_count++;
  signal(x_sem);
  wait(next);
  next_count--;
}
```

Now, if we see the signal part; so, you see the signal part is implemented like this. So, if count is greater than 0, then it has to go out of this semaphore and it go out of the monitor and it has to signal some process which is waiting on this condition variable x and that is done by this.

So, this next count plus. So, this is actually the process will be going out from this. So, the signaling process will be going out through this next line. So, that was that is why it has implemented this next count then signal x sem. So signal x sem. So this will be signaling the waiting process ok. So, these on this condition variable x. So, this will signal that process and then it will wait for next. So, wait for next will take the process out from here and it will be waiting in the next queue.

And then again after some time some other process when it finishes so, it will check whether next count is greater than 0 or not and then it will be again taking the process inside, and at that time this process of wait will be over. So it will come to this point and it will be executing this next count minus minus so next count will be decremented.

So, it is a collaborative implementation of number of processes they are execution sequence they how they will be done, the semaphore, the monitor boundary and this individual functions that we have individual access routines that we have and the wait and signal calls. So, everything are used together so, that we can get to this whole thing implemented and a proper synchronization is done.

So, you just think over the whole issue and take all of them together then only you will be able to understand the bits and pieces they will fall into the their proper places. So, this is how this monitor can be implemented using semaphore it is a bit complex concept, but I think you just think it in a cool mind and then you will be able to get it.

(Refer Slide Time: 07:17)

Resuming Processes within a Monitor

- If several processes are queued on condition x, and x.signal() is executed, which one should be resumed?
- FCFS frequently not adequate
- conditional-wait construct of the form x.wait(c)
  - Where c is priority number
  - Process with lowest number (low number → highest priority) is scheduled next
- Some languages provide a mechanism to find out the PID of the executing process.
  - In C we have getpid(), which retunes the PID of the calling process

Now if you want to resume processes within a monitor if several processes are queued on the condition variable x and x dot signal is executed, which one is resumed? So, this is a problem like I have on this condition variable x we have shown that there are a number of processes waiting on this condition variable. So, which process will be resumed? The strategy that we have discussed so far it does not tell anything about which process will be resumed. So, naturally there is a question of starvation.

So, FCFS is frequently were not adequate, first come first serve may not be adequate because of the reason that there may be priority. Conditional wait construct in the form of x dot wait c is a priority number so, that can be used. So, if the processes have got some priorities. So, we can put that priority into this x dot wait call so, that this priority is taken care of. So, process with lowest number is scheduled next. So, if the number is low we mean that it is a priority is higher so, the process is the lowest number so, that is given the highest priority.

Some languages provide a mechanism to find out the PID of the executing process in C we have got get PID type of function which returns the PID of the calling process. So,

this returns the PID of the calling process so, that way we can issue the priority in number of that get the priority value of that. So, this is possible.

(Refer Slide Time: 08:52)

Next, can have resource allocator based monitor. So, monitor to allocate a single resource among competing processes. So, this is a situation where we have got a monitor that is used to allocate some resource and this is the way this is implemented. So, we have got a monitor resource allocator and in this we have got this Boolean variable busy and this condition variable x. So they are internal to the monitor they are not feasible from outside. This acquire and release so these are two functions that are available from outside the thing; from outside the monitor.

So, you can have a look at this now this outside routine so, it can give you a call to this in acquire. So, any process when it is requesting an allocation of the resource, it specifies the maximum time it plans to use the resource. And for that time, the monitor will allocate the resource to the process that has the shortest time allocation request.

So, it is assumed that we have got this type of strategy that whenever a process say P i wants to access a resource. So, it tells for how much time it needs to access the resource. So, in the code of P i maybe it is giving a call to acquire 10. So, what is happening in the acquire routine? So, if the resources not busy then it is immediately allocated, but if it is busy then the process is put into the corresponding condition variable wait states so, x dot wait.

And in the condition variable for x so, it is put into a queue like this and since if some other process P j also comes and that also makes a call to this acquire and that requires say so, for 20 time unit. Then this process P i is placed first then process P j like that. So, whichever process has got a less a resource utilization time. So, that will be given priorities another process P k comes. So, P k says that I want to acquire the resource for 5 time units. So, this needs to be modified. So, that P k comes here P i goes there and P j goes at the next place.

So, that way the queue has to be manipulated and that can be done very easily by checking into the corresponding values of this semaphore; of this condition variable. So, we can check where it should be put. So, if a process is acquiring the resource it is doing like this similarly you release the resource. So, this is made by making busy equals to false and calling this x dot signal. So, we assume that x dot signal it returns the first process from this queue and this queue is already sorted based on the timing value of this resource utilization.

So, we can say that this will be taking out the process which requires the resource for the shortest amount of time. So, it is trying to follow that shortest job first type of policy, but with the assumption that the process themselves tells like how much time it needs the resource. So, and there is an initialization code which is busy equal to false. So, this particular statement these busy equals to false so, this is the initialization code for the monitor.

So, when this monitor resource allocator is created so, these variables are created and this busy is initialized to false. After that any process asking for the resource so, it will be calling this acquire routine, and similarly for releasing the resource so, it will be calling the release routine.

(Refer Slide Time: 12:45)

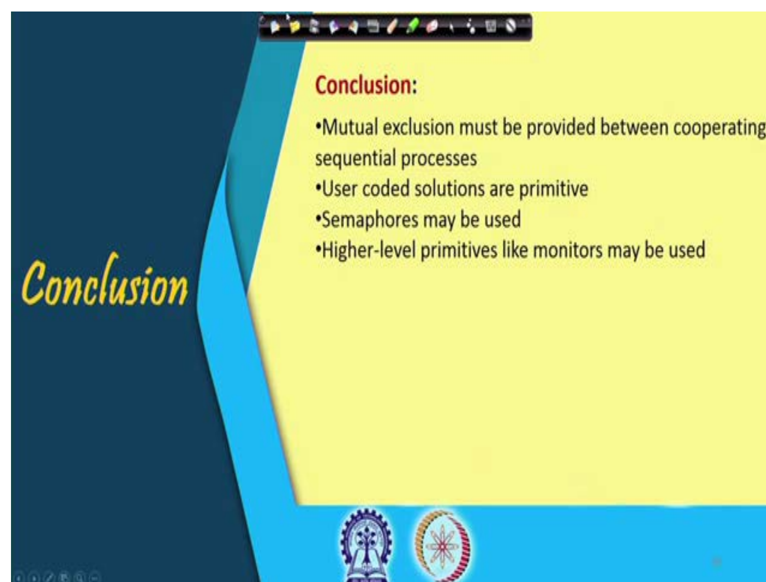
So, this is how this monitor can be implemented. Say, process that needs to access the resource in question must observe the sequence like this that R dot acquire t. So, it will be trying to acquire the resource then access the resource then R dot release so, it will try to release the resource. So, that way it can call that acquire and release functions from the resource allocator monitor and accordingly it will be executing them. So, we can have this resource allocated implemented by means of monitor.

(Refer Slide Time: 13:18)

So, observations like incorrect use of operations like again the type of operation that we can have the incorrect sequence maybe first a process gives a call to release and then it gives a call to acquire. So, these acquire release call sequence so, that is just reversed. So, unlike this thing so, this is this acquire release call. So, instead these r acquire first and release next so, maybe it is just these are just reversed. So, that way we can have a problem that release an acquire reversed, similarly the user make a mistake that instead of calling release may call acquire again. So, that is acquire acquire type of sequencing,

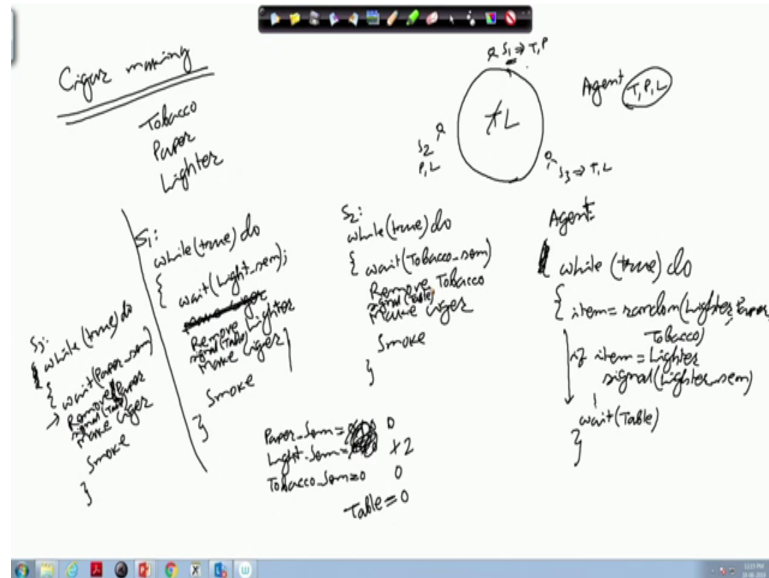
The another possibilities omitting acquire and or release or both maybe for forgetting to put this acquire or forgetting to put this release. So, these type of things can happen, but they are not that much severe compared to that semaphore based implementation. So, monitor we can always check whether it is there or not. So, solutions can be there, but it is going beyond the scope of this course so, we will not go into that discussion.

(Refer Slide Time: 14:34)




So, to conclude this discussion that we have made so, far was the mutual exclusion must be provided between cooperating sequential processes, user coded solutions are primitive. So, semaphores may be used and higher level primitives like monitors may be used for this purpose. So, these are the things that we have learned so, far. So, next we will be looking into some synchronization examples and we will see how they can be solved using this semaphore based solutions.

(Refer Slide Time: 15:18)




So, first we will take an example, which is the problem statement is like this that we have got say there is a round table. So, this table so, this problem that we are talking about. So, this is in popular this is known as cigar making example and my objective is not to promote smoking and all, but I am just taking it this as a as an example problem ok.

So, to make a cigarette and smoke so, what is required is that we need some tobacco, need paper and lighter; we need tobacco paper and lighter. Now we have got three smokers sitting here smoker 1 smoker 2 and smoker 3. Smoker 1 has got infinite supply of tobacco and paper, smoker 2 has got in finite supply of paper and lighter and smoker 3 has got infinite supply of tobacco and lighter. So, none of them can individually make a cigar and smoke. So, because of the third item is not present fine.

And there is another agent; so, this agent does not smoke. So, agent has got infinite supply you have all the three items like tobacco, paper and lighter. So, what the agent does is that randomly picks up one of these three items and puts it onto the table suppose the agent puts tobacco here.

So, as a result this smoker 2 will be able to get it make a cigar cigarette and smoke. Then after that agent takes off this tobacco and places again randomly one of the three items tobacco, paper, lighter onto the square on the table suppose puts a lighter here then this tobacco. So, now, the smoker 1 can make a cigar and smoke. So, this way you have to give a solution, we have to model this particular problem in terms of a synchronization

problem. So, how do you do this thing? So, for doing this so, we can write down the code for smoker 1. So, I assume that this is an infinite loop so, while true do.

So, this smoker 1 so, they has to wait for the lighter. So, wait for I assume that there is a semaphore light sem on which this smoker has to this anybody waiting for this lighter will be waiting on this light semaphore, then it will be calling make cigar and then let us make it like this. So, whenever this is there, then remove lighter then mix cigar then smoke and that is in an infinite loop fine.

So, similarly smoker 2 that particular code is like this. So, this will be while true wait on tobacco semaphore. Now remove tobacco mix together rest of the thing is similar, make cigar smoke similarly you can write the code for S 3. So, S 3 code will be like this while true waiting for this paper paper semaphore, then remove paper, remove paper then make cigar then smoke. So, these processes are more or less similar. So, 3 smoker processes are quite similar what about the agent? Ok agent; so, agent is also in infinite loop this is while.

So, it has to randomly pick up one such item. So, item it has to pick up randomly from lighter paper and tobacco. Now if item is equal to lighter, then it has to give that it has to signal lighter semaphore ok. Similarly, you can write other two ifs. So, if item equal to paper then signal paper semaphore the item equal to tobacco signal to tobacco semaphore like that.

Now, after that so, it has put it there then after some time either it is taken up by some this smokers or it has to repeat. So, this whole thing must go on. So, again this is in an infinite loop so, it has to go back there ok. So, how it operates? So, these 3 semaphores that I have that is this paper semaphore. So, this paper semaphore; so, this is initialized to 0, then this lighter semaphore light semaphore. So, that is initialized to 0 and this tobacco semaphore. So, that is also initialized to 0 all of them are initialized to 0.

So, initially if any of these S 1, S 2, S 3 they are scheduled. So, they will get stuck at the corresponding wait statements because none of them will be getting anything. So, now, once the agent process is scheduled so, agent process it will be getting, it will be can picking up randomly one of the item and based on that it will be making one of the semaphore variable to be equal to 1.

So, maybe it will make the picks up the lighter the lighter is made equal to 1. So, the S 1 was waiting at this points say S 1 will be signaled and it will remove the lighter and it will take it ok. So, that similarly next time so, once it executes this wait so, this will become 0 again and it will go on like that. So, the next time this agent comes; so, agent will be again picking up 1 item that way and it will be maybe this time puts paper onto the table.

So, makes this semaphore to be equal to 1 and then this S 3 will get a chance for execution. So, S 3 will make this paper it will execute this wait. So, it will make it back to 0 and then it will do the cigarette and smoke ok. Now, the question is this a correct solution? So, apparently it seems that it does well, but there is a catch here like suppose it may it happens like this agent process.

So, agent process it executes 1 loop of this statement accordingly it has picked up a lighter. So, it has made this lighter semaphore value equal to 1 fine. Now before this lighter is; so, before this process S 1 is scheduled suppose, this agent is scheduled once more. So, one thing you remember that we said that there is no assumption regarding relative speed of execution of the processes. Now it may so, happen that before S 1 is executed this agent executes once more.

So, as a result what happens is that this agent will be putting another item on to the table so, and how will it put it; so, it will make this one of the semaphore to be equal to one. So, next time it may be making this paper semaphore to be equal to 1 or it may so, happen that in this random choice process it picks up lighter again and as a result this lighter semaphore becomes equal to 2 whereas, the rest of them remains 0. So, this can happen.

So, this semaphore values; so, due to this relative speed problem, so, it may so, happen that after some time there may be a number of items accumulated on the table because the corresponding smoker processes they did not execute in between ok. So, if the agent is executing at a high speed and as a result this agent has piled up lots of items onto the table and then this smoker processes come and they find the items are available on the table by means of this semaphore variables having some large value more than 0 and that way all the processes will go into their critical section which is basically making the cigar and smoke.

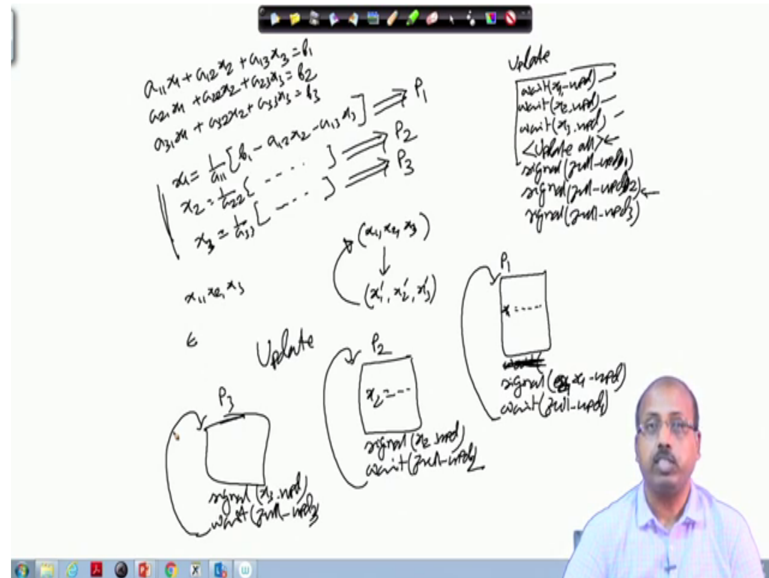
So, it will be the critical section so, they will be going into their critical section simultaneously. So, to avoid this situation one possibility one possible solution is that this agent before starting the next iteration it must be told that the item that it put on the table has already been consumed by one of the smokers. So, this table access is an important issue.

So, we can say that we can have another semaphore variable called table, which is also; which is initialized to 0 and so, that this after putting this agent process so, at the end of it so, it should wait on this table. It should wait on this table and after making after removing the paper here so, this at this point there should be a signal, there should be a signal table here.

And obviously, there is a there is another issue like say if this removal of paper or that is when this agent process is putting something onto the table and at that time if this smokers they come so, they are trying to access that table. So, we should be able to protect that and of course, that will be ensured automatically because, now if the agent comes so, it executes this code and it waits for this table, table variables initially is 0.

So one of these smokers will be scheduled; so, they will be once that is done. So, they will be doing this signal operation signal on this table similarly here also I should have a signal on table. So, this will be executed and once the signal is executed; that means, the item has been taken by one of the smokers. Now, this agent will be through this wait statement and it can go back and execute this while loop again and it will be able to get the item from the table and put the next item on to the table. So, this way we can have this process synchronization type of problem modeled by means of the semaphores ok.

(Refer Slide Time: 29:58)




So, another typical example that we can think about is like this, suppose I have got a set of equations and suppose I have got set of simultaneous equations. Suppose there are three variables x 1 x 2 and x 3, a 1 1 x 1 plus a 1 2 x 3 equal to b 1 then a 2 1 x 1 plus a 22 x 2 plus a 2 3 x 3 equal to b 2 and then a 3 1 x 1 plus a 3 2 x 2 plus a 3 3 x 3 equal to b 3. Now you know that there are many numerical methods by which we can solve these equations like this x 1 can be obtained from this equation 1 by a 1 1 into b 1 minus a 1 2 x 2 minus a 1 3 x t3. So, from the first equation we get x 1 from the second equation we get 1 by a 2 2 into this x value similarly x 3 is 1 by a 3 into this equation.

And the way this numerical solution methods work is that they assume some initial values of x 1 x 2 x 3; x 1 x 2 x 3 and then these equations are used repetitively to get new values of x 1 x 2 x 3 until and unless we will get the values converts to some value. So, there is an epsilon the error margin so, in the previous iteration whatever were the values of x 1 x 2 x 3.I In the next iteration if the values are almost similar to that and the error margin is less than epsilon then the iteration stops otherwise it goes on doing this thing.

Now here you see that one thing is that suppose I have at present; I have got current pair of current people of these values x 1 x 2 and x 3. So, that triplet it will be used for getting the next one. So, x 1 dash x 2 x 3 dash. So, if I say that x 1 computation. So, this is done by process P 1, x 2 computation is done by process P 2 and this x 3 computation is done

by process P 3 and there is an update process which copies these x 1 dash x 2 dash x 3 dash into the x 1 x 2 x 3 so, that next iteration it can use this.

Now this process P 1 that if you try to look into this, process P 1s code so it will be having this computation for this equation 1 x 1 equals 2 etcetera ok. After that it should wait for sorry, it should do a signal on some semaphore x 1 updated and then it wait for another semaphore which is full update. Similarly P 2s code will be like this so, this is x 2 equal to something and then it should signal x 2 updated and then it should wait for full update and then this will continue, this will go in a loop. Similarly the P 3 code will also be similar to that signal x 3 update and then it should wait for full update then it will do the next operation.

Then this one this update routine so, update routine maybe like this. So, update routine so, it should wait for this x 1 update; on this semaphore x 1 update it should wait for x 2 update, it should wait for x 3 update and then it should do the update operation which is basically copying the x 1 dash x 2 dash x 3 dash into x 1 x 2 x 3 update all.

And after that it should give this a signal full update and this signal should be given thrice because there are three processes waiting for this. So, I have to give this full update thrice. So, that all 3 of them will be or better I should say this is I should say full update 1 full update 2 and full update 3 and accordingly it should give a signal full update 1, signal full update 2 and signal full update 3 that will be a better route.

So, what am doing? I am having a number of semaphore variables x 1 update x 2 update x 3 update. So, all of them are initialized to 0 so, that even if the update process comes first. So, it will be waiting on these and then when P 1 is over. So, it is giving signal x 1 update. So, this will be true, then this will be true, this will be true then update process will update updated everything then it will signal full update 1 as a result P 1 was waiting at this point. So, P 1 will know that update has been done. So, P 1 can start its next iteration.

Similarly, by this signal so, P 2 will know that this update is over. So, this update has been done. So, it will be doing this thing again it will go to the next iteration of this statement. Similarly full update 3; signal full update 3 so, it will be getting this full update P 3 will be getting the signal and it will start the next iteration.

So, in this way the synchronization problems we can solve using semaphores. So, we will be looking into more classical problems in the next class.