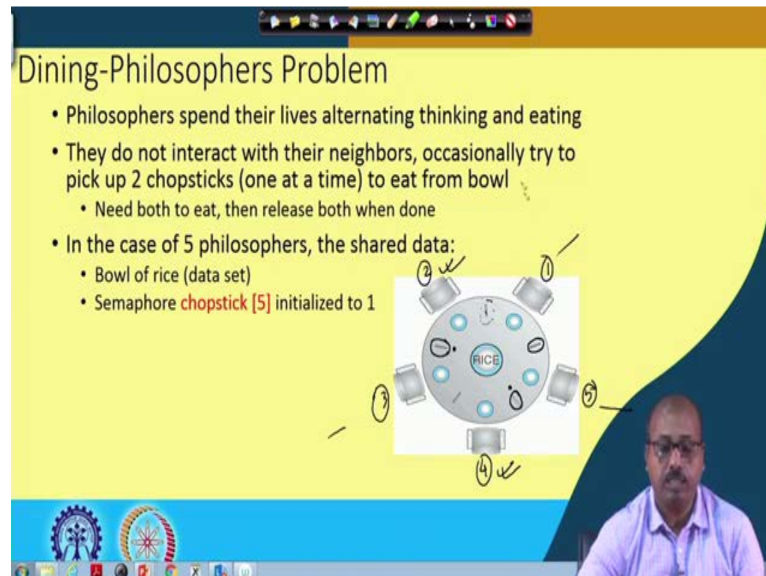
Operating System Fundamentals
Prof. Santanu Chattopadhyay
Department of Electronics and Electrical Communication Engineering
Indian Institute of Technology, Kharagpur

Lecture - 36
Synchronization Examples, Deadlock

(Refer Slide Time: 00:27)



So, next we will be looking into the dining philosophers problem so, the problem statement is like this. So, we have got a set of philosophers say they are sitting on a table and philosophers by nature they are in thinking state, but at some point of time they also feel hungry. So, they are sitting on a dining table and they are trying to eat some food and are simultaneously also continuing with the thinking process. So, all of a sudden some philosopher while thinking may feel hungry and try to eat some food and after that again goes into the thinking state.

So, this is basically modeling the situation where we have got processes and their behavior unpredictable, like a process may be doing some computation which is basically the thinking state of the philosopher and after some time the process may ask for some resource ok. So, it tries to grab some resource and that is basically the philosophers trying to eat and for eating the philosopher they need some spoons and it is assumed that the food that is there on the table. So here it is a model may be some

amount of rice and the rice is such that the philosopher or any philosopher will need 2 spoons to eat the rice so or 2 chopsticks to eat the rice.

So, that is the thing so, we need 2 chopsticks. So, one philosopher to be able to eat must grab 2 chopsticks, one on the left side, one on the right side and they are not snatching it from a distant chopsticks. So, philosophers are well behaved so, they do not snatch it from a for chopstick which is away from them. So, they just try to grab the chopstick to the left or right. So, in the dining table the situation is like this that the food is there in the middle and each philosopher has got a plate and between the plates we have got chopsticks, so these are the chopsticks.

So, for this philosopher to be able to eat so must catch hold of both the chopsticks, but at the same time they cannot take both the chopsticks simultaneously. So, they have to request for chopsticks one after the other, maybe they first request of the left chopsticks, then for the right chopstick. So, in terms of processes and resources so, we can think that each chopstick is a resource and if a process requires 2 resources to for doing the next phase of computation and the process has to request them separately. So, they cannot ask for the 2 resources simultaneously.

So, this is the situation the overall problem statement is like this, that philosophers spend their lives alternating, thinking and eating, they do not interact with their neighbors, occasionally try to pick up 2 chopsticks one at a time to eat from the bowl and need both to eat and then release both when done. So, that is the whole idea and in this particular case we have got 5 philosophers. So, that is the shared data that we have is the bowl of rice which is the data set and semaphore which is chopstick 5 so, these 5 chopsticks are there. So, that they are modeled by 5 semaphores an array of 5 semaphores, all of them initialize to 1 ok.

So, our so, bowl is not that much important for this from the sharing point of view, because if a philosopher has got chopsticks though he can access the bowl. So, bowl access is shared, so we do not need to do any protection there, but chopstick access is mutually exclusive. So, once a philosopher has grabbed the chopsticks to his left and right, the neighboring philosophers to the left and right of him, so cannot get any cannot get those chopsticks. So, this is how we are going to model this particular problem.

(Refer Slide Time: 04:22)

Dining-Philosophers Problem Algorithm

- The structure of Philosopher *i*:

```
do {
    wait (chopstick[i] );
    wait (chopStick[ (i + 1) % 5] );

            // eat

     signal (chopstick[i] );
     signal (chopstick[ (i + 1) % 5] );

             // think

} while (TRUE);
```

So, the structure of this philosopher i is like this. So, the philosopher will wait for chopstick i, then it will wait for chopstick i plus 1 percent modulo 5. So, chopstick as you as we; as I have said that chopstick 5 is an array of 5 semaphores all of them initialized to 1. So, for any the ith philosopher coming so, if nobody else is trying to grab the chopstick.

So, this philosopher will be able to grab the chopstick to the left and chopstick to the right. Then proceed to the eating and after eating is over so, it in the philosopher will release the chopsticks. So, it will signal chopstick i and signal chopstick i plus 1 modulo 5. Then the philosopher goes into the thinking state after some time the philosopher will again come up. So, the philosopher will feel hungry again at that time again try to grab the chopstick.

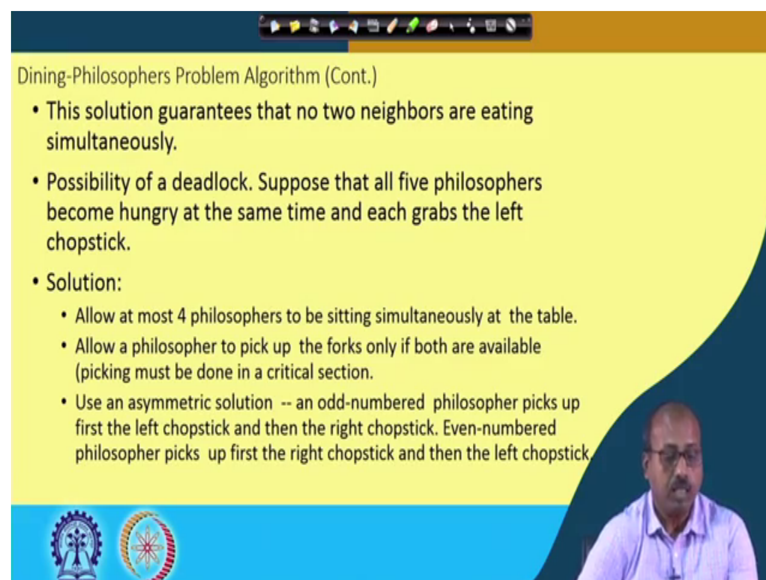
So, this is how this problem is modeled ok, so using semaphore so you can propose a solution such that this ensures mutual exclusion. So, at no point of time one chopstick is taken by more than one philosopher, so that the mutual exclusion is guaranteed. But of course, there are other problems like it may so, happen that the first philosopher comes up ok.

So, let us look into this diagram maybe this philosopher comes up grabs this particular chopstick by executing that wait i instruction, then a chopstick i. Then before this philosopher executes the next statement to grab this chopstick maybe this philosopher

has come up and has grabbed already grabbed this chopstick and before this philosopher has grabbed this chopstick maybe this philosopher process has come up so, it has grabbed this chopsticks.

So, it may so happen all philosophers they have all of they have grabbed their left chopsticks and entries nobody is able to release the chopstick because chopstick release comes only at this point. So, after grabbing the left chopstick all of them are stuck at the right chopsticks. So, all the philosopher processes they are waiting at chopstick i plus 1 modulo 5, so at this statement. So, this makes the solution deadlock prone like it misses a no philosopher can proceed. So, that is a situation of deadlock and that problem is there in this particular solution ok.

(Refer Slide Time: 06:43)



So, the solution guarantees that no two neighbors are eating simultaneously so that is guaranteed. There is a possibility of deadlock suppose that all philosophers become hungry at the same time and each grabs the left chopsticks. So, that is the situation I was talking about that there is a possibility of deadlock. So, what is the solution? So, there can be several solutions here first solution is allow at most 4 philosophers to be sitting simultaneously at the table. So, instead of 5 philosophers so, if we allow 4 philosophers. So, maybe these philosopher is; this philosopher is not there so, but I have got 5 chopsticks on the table.

So, everything else remain same, but this chair is empty nobody is sitting here. Then what will happen is that this philosopher will never face any problem in grabbing this chopstick, similarly this philosopher will never face any problem with grabbing these chopsticks. So, if I even if we take that all philosophers they have grabbed their left chopstick. So, this fellow has taken this one, this philosopher has taken this one, this philosopher has taken this one and this philosopher has taken this one, but at least this philosopher will all be able to grab the right chopstick also ok.

So, that way this philosopher will finish off and then release both the chopsticks. So, one then this chopstick will become available. So, then this philosopher will be having both the chopsticks available with him so, he can proceed. So, that way the deadlock situation can be broken. So, one solution to the deadlock problem here is to allow at most 4 philosophers to be sitting simultaneously.

Then second alternative is that allow a philosopher to pick up the forks only if both are available. So, in the solution that we have suggested here the problem is that we were trying to grab one chopstick at a time. So, if we say that no you cannot do like this so, you have to either grab both of them or none of them. So, if I make this part of code a critical section code. So, that if this philosopher is doing chopstick i pick up, so it though no other philosopher will be able to do this chopstick i plus 1 pick up. So, that way this philosopher we will be able to grab both the chopsticks so, that then the philosopher will be able to proceed to eat.

So, this is another solution where a philosopher is allowed to pick up forks only both are available and picking must be done in a critical section code. Also it is possible to have an asymmetric solution, so this is an odd numbered philosopher picks up first the left chopstick and then the right chopstick and the even numbered philosopher picks up the right picks up the first the right one and then the left one.

So, coming to this solution, so if I number this philosopher, so this is philosopher this is philosopher 1, 2, 3, 4 and 5. Now, this odd philosophers so, that is 1, 3 and 5 they pick up their left chopstick first. So, this chopstick is picked up sorry this is the left chopstick. So, this philosopher picks up this chopstick, similarly this philosopher picks up know this philosopher picks up the left chopstick and this philosopher picks up the left chopstick, they try to grab the left chopstick first.

Now, at that time if this, on the other hand this even numbered one that is 2 and 4 they will try to grab the right chopstick first. So, in the execution of 2 so, this 2 will try to grab this chopstick first and 4 will try to grab this chopstick first the right chopstick. As a result so, there is no contention for this chopstick ok. So, it cannot happen that this philosopher has grabbed this chopstick and asking for this, that is not possible.

So, naturally when this philosopher is asking for say this chopstick, so this one is free. So, I can; so this philosopher will be able to grab this chopstick and continue. So, this way we can have some asymmetric solution and that asymmetric solution will ensure that there will be no deadlock in the system.

So, this way we can have this variants of this dining philosophers problem using an asymmetric solution that odd numbered philosopher picks up the left chopstick and then the right chopstick and even numbered philosophers picks up the first the right chopstick and then the left chopstick. So, this is one possible solution.

(Refer Slide Time: 11:42)

So, in this particular section or particular set of lecture, so we have seen some examples on this synchronization problems. Semaphore and monitor based schemes we have discussed so, they will be monitor based solutions can also be there for this dining philosophers problem or reader-writers problem. So, in our previous classes we have taken the some monitor based solutions.

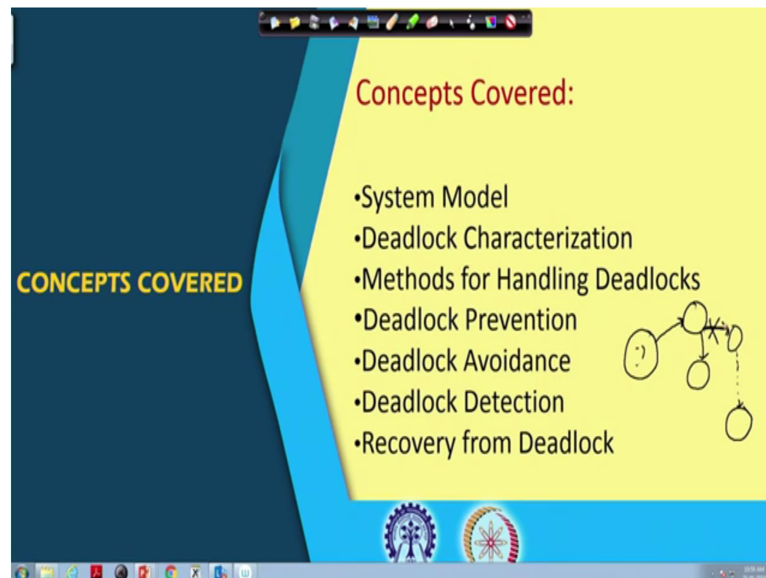
So, if you can look into any textbook for monitor based solution for this dining philosophers problem there. And so, with that we end this portion of code of this portion of this synchronization problem solutions, next will be taking up another very important discussion that is on the issue of deadlock.

So, deadlock happens to be a fundamental problem in a operating system, because now there are multiple processes and those multiple processes they are executing simultaneously, at they are executing concurrently and then they are requesting for some resource and they releasing some resource at some point of time.

Now, it can very well happen that the processes they are waiting for some resource which are already grabbed by some other process and those other processes they are again waiting for some resource which is grabbed by these processes. Overall the system in the system I have got a set of processes which are interdependent on each other in terms of resources and such that one process is requesting for a resource is grabbed by the other process and that other process is requesting for some resource which is grabbed by this process. So, that way none of the processes can continue.

So, this type of situation, so they are known as deadlock situation and if there is a deadlock in the system then the system throughput will be very low, even if you have got large number of processes in the system which we call the degree of multi programming, even if degree of multi programming is high you will find that the none of the processes are completing because of this deadlock situation. So, that will bring the system throughput to a low value and this waiting time and also they will be going up.

(Refer Slide Time: 13:57)



So, in this particular portion of our lecture, so we will be looking into this concept of deadlock and the concepts that we will covered we will start with the system model like how do you visualize the overall system from the perspective of this resource and deadlock these issues.

How do you characterize the deadlock, so how do you, what are the properties of this deadlock, then how can we handle deadlock, so that is very important. Then we will see some methods for prevention of deadlock, avoidance of deadlock and detection of deadlock like any other disease that we have.

So, in our in human being also we can treat it in three different ways, we can have some preventive measure, we can have some vaccination for example, so, that will prevent those diseases. We can do some avoidance like if we know that the certain part of the world is affected by some disease, so we do not go to that part of the world so, that is the avoidance policy.

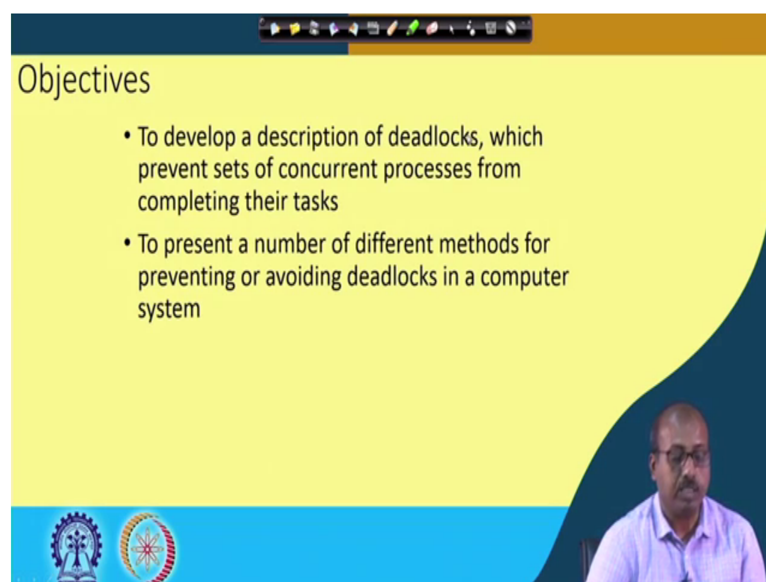
And the detection that is the final one; so you can have some test and see whether we have been infected by that disease and if there is a detection and then we have to cure that disease by taking medicines or whatever. So, similarly in a system also so, we can have these three case or policies for handling this deadlock situation, we can prevent the deadlock by ensuring the case that we it will never create any deadlock, we can avoid, we can adhere to some policy so, that deadlock will never occur.

Then we may try to avoid the deadlock so, we may try to take steps. So, that we do not take any allocation resource allocation, that may eventually lead to deadlock. So, those possibilities are cut down at the very beginning. So, we do not make the system to go from one state to another such that it finally, may go to deadlock state. So, what I mean is that, at any point of time you can take a system state to be the set of processes and resources that are there in the system so, that is the; that is one state.

And from this it can go to a new state when some process is allocated some resource or some process releases some resource. So, that way for the entire system you can think that it goes through via several transitions. Now, it may so happen that if I do this allocation then eventually it may lead to a deadlocked state for the whole system. So, we do not do this allocation at all, so this branch is totally cut off ok. So, apprehending that there that this will lead to a deadlock situation, so we do not go into that part at all so that is the avoidance policy.

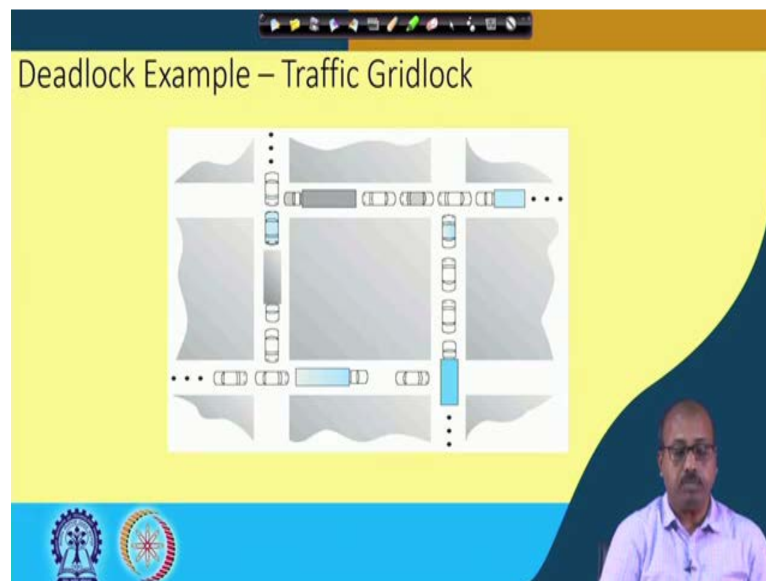
And detection is of course, there so, you can run some health checking algorithm that will see whether there is any deadlock in the system and if the deadlock is there, it will try to recover from the deadlock situation. So, the deadlock detection and followed by recovery from deadlock, so these are there. So, we will look into these concepts one by one.

(Refer Slide Time: 17:06)



So, objective is to develop a description of deadlocks, which prevent sets of concurrent processes from completing their tasks. So, we have to we will discuss about the deadlocks, to present a number of different methods for preventing or avoiding deadlocks in a computer system. So, how can we prevent deadlock, how can we avoid deadlock or how can we detect deadlock and recover. So, all these things will be discussed.

(Refer Slide Time: 17:31)



A typical example of deadlock is a traffic gridlock, suppose we have got a set of roads like this and the cars are going like this. Now you see this particular car it cannot proceed because there is a car standing here and it cannot go this way. So, all these; this road has got a number of cars. So, this dot means these there are cars in front of this also, so this cannot proceed.

Now, this fellow can proceed only if this car moves so, if this car moves then this one can proceed, but for this car to move, so I will need to have some previous car cleared. Similarly say this one what is happening is that say this car so, this car can move if this is free so, but if this is free then in that case I have to have; so this part this road available so, this is also stuck here, because this is not progressing.

Similarly, this is not progressing because this road is stuck and again this is so, for this road to be free. So, this car is blocking it and this car cannot proceed because there is a

blockage here. So, as a result there is a circular blockage in the grid, so none of the cars are progressing. So, this situation is a typical situation that occurs in the traffic line.

(Refer Slide Time: 18:55)

System Model

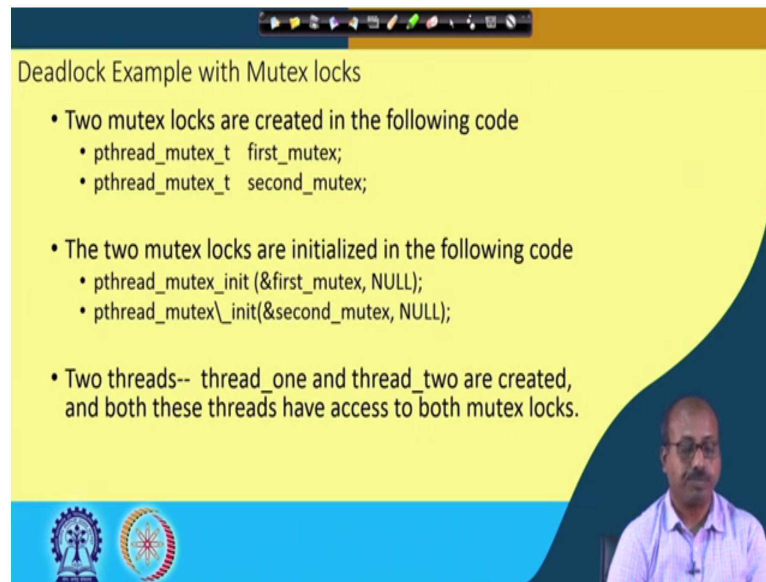
- System consists of resources
- Resource types $R_1, R_2, \ldots, R_m$
  CPU cycles, memory space, I/O devices
- Each resource type R_i has W_i instances.
- Each process utilizes a resource as follows:
  - request
  - use
  - release

So, similar case can occur in computer system also when the processes they are waiting for resources to be free, and the resources are currently held by some other processes. So, the system model that will consider it consist of resources so, we have got resource types R 1, R 2 up to R m, so we have got say m different types of resource. So, R 1 may be CPU cycle, R 2 may be memory, space R m maybe some I O device etcetera and each resource type R i has W i instances.

So, maybe in my system I have got three printers. So, if R i is equal to printer, then W i is equal to 3. So, how many instances of a particular resource type we have? So, that is W i, and each process utilizes resources in this sequence first it will request for the resource, if the resource is available then it will be told that yes the resource is available, then it will use the resource and after using the resource it will release the resource, so this is how this system model works.

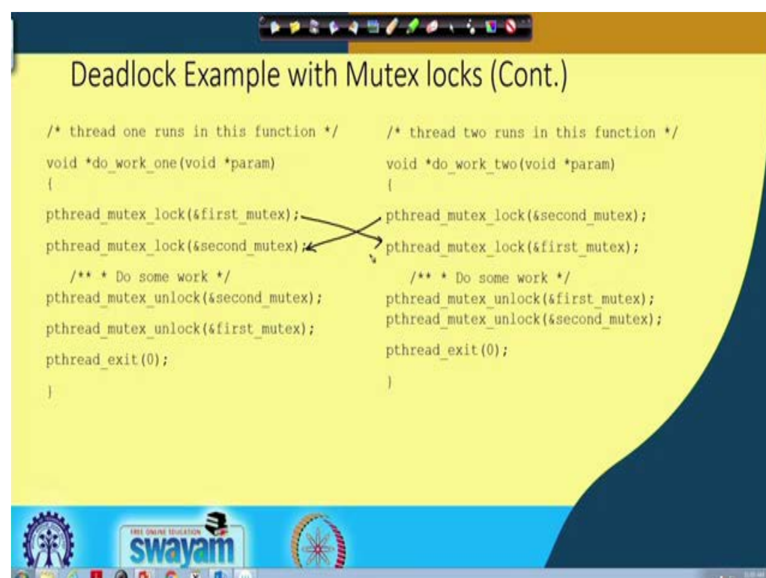
So, any process willing to get a resource so, it will send a request and this operating system will grant permission and then only it will use it and then it will release it by informing the operating system that I am done you take back the resource from me ok. So, this resource, request, use, release so, this will be followed for all the processes.

(Refer Slide Time: 20:19)



A typical example maybe even that we have got some mutex locks. So, two mutex locks are created here so, pthread mutex t first mutex and pthread mutex to second mutex. So, we have got two mutex locks and these two mutex locks are initialized here. So, they are both of them are initialized to NULL ok. Now, there are two threads thread one and thread two are created and both these threads have access to both the mutex locks, so they can access both the mutex locks.

(Refer Slide Time: 20:51)



Now, say the first one so, it does it like this, thread one runs in this function so, do work one. So, pthread mutex lock it is trying to lock the first mutex and then pthread mutex lock then it is trying to lock the second mutex, then it will do some work, then it will unlock the second mutex and then it will unlock the first mutex then it exits.

On the other hand this second thread so, it does it like this. So, it first puts a lock on the second mutex and then it puts a lock on the first mutex, then it will do some work and then it will be unlocking this first mutex and then unlock the second mutex. Now, you see the typical problem that can come is like this; the typical problem that can come is like this.

So, the first thread executes the first statement as a result it has locked the first mutex. And now suppose this thread swaps out because each thread is given some fixed amount of time for execution the time slice expires and this thread is taken out of CPU and the other thread executes; and other thread executes the first statement, so pthread mutex lock second mutex. So, it has set a lock on the second mutex.

Now, the first thread again comes and it tries to put a lock on the second mutex, but it will not be successful because this lock is already put by the second thread ok. So, this is not successful, so it is stuck here. On the other hand after some time if this thread is scheduled, so this thread will now try to put a lock on the first mutex and the first mutex is already locked by the first thread. So, as a result the second thread also gets blocked at this point. So, now, the situation is that the first thread it is waiting for a thread which is currently locked by the second thread and the second thread is currently waiting to put a waiting for a lock on the first mutex which is currently locked by the first thread.

So, we have got a mutual waiting. So, we can say that this is waiting for this one. So, basically this is waiting for this one and this is waiting for this one. So, this situation occurs so, both the threads will get stuck at this point. So, this is a deadlock situation so, because none of the threads can proceed and basically the only way out to solve this problem is to either kill one of the threads or we have to reset the system. So, that it starts from the beginning altogether. So, whatever be the case so that is that the situation is not desirable ok. So, we have to do something so that this deadlock can be handled.

(Refer Slide Time: 23:45)

And if you try to see what are the properties that a deadlock situation must satisfy is that, if it requires that the following four conditions they must be held simultaneously and we can argue that if any of these conditions is not satisfied at one point of time then deadlock cannot be there. So, violating any of them will ensure that there is no deadlock in the system.

So, I have to prove that a system is not deadlocked what you have to do is that we have to show that; so at least one of these properties is violated ok. The first condition and if you want to if you can show that all the four conditions can potentially hold then there is a potential chance of a deadlock occurrence. Though holding all this condition does not mean that there will always be a deadlock, but these are these conditions are necessary for deadlock to occur.

So, first one is the mutual exclusion, so only one process at a time can use a resource. So, if this mutual exclusion can be violated then of course, deadlock cannot occur, because if a resource is such that multiple processes can access the use the resource simultaneously like say the screen of the computer. So, this screen multiple processes can write simultaneously onto the screen, but think about the printer. So, printer if it is given to one process and that process is writing on to the printer, we cannot allow other process is to write onto the printer. So, that way it is a mutually exclusive access as far as the printer is concerned.

So, if the mutual exclusion is not there then no process has to wait for the other process. So, even if the other process is holding the screen resource, then the other this process need not wait, because it can also write onto the screen simultaneously, so mutual exclusion is not required. So, if mutual exclusion is not required to be satisfied, then deadlock cannot occurred. Second thing is the hold and wait a process holding at least one resource is waiting to acquire additional resources held by other processes.

So, if the process is not holding any resource and it is asking for some resource, then no other process can wait for this process to be over. So, what we mean is that, suppose I have got the process P 1 and a process P 2. So, process P 2 is currently holding the resources R 1 and R 2, but process P 1 is not holding any resource. Now, if process P 1 asks for a resource say R 3, so that cannot cause any harm because it is so, even if this process P 2 is also holding R 3.

So, I can say let P 2 finish off and when P 2 will finish off then this R 3 will be available and then it will be given to P 1. So, it is basically a scheduler decision policy, but the system is not deadlocked. So, it is not the case that P 1 and P 2 they are stuck simultaneously. So, this hold and wait condition if it is violated then also this deadlock cannot occur.

Then there is a third condition, which is said no preemption, a resource can be released only voluntarily by the process holding it after that process has completed it is task. So, it is like this some of the resources that we have so, they are preemptable in nature, some of them are non preemptable in nature for example, if you take the CPU time ok. So, CPU is given to a process for some time, so if you find that another process needs the CPU and you need to allocate the CPU to that process. So, you can preempt the currently running process from the CPU and give it to the second process.

Similarly, if it is a main memory, then what you can do? You can make the main memory free by writing that main memory content on to some disk space and later on when required. So, you can copy back from the disk to the main memory.

(Refer Slide Time: 28:11)

So, this no preemption so, if this condition is violated so, if you can preempt a source then also you need not wait like say process 1 is holding a resource R 1; so, it is holding a resource R 1 and P 2 is holding a resource R 2.

Now, if you find that process 1 is requesting for R 2 also and it should be, it has to be granted then if R 2 is a preemptive resource say CPU (Refer Time: 28:26), CPU or main memory etcetera, then you can take back R 2 form P 2 and give it to P 1, then P 1 will have both R 1 and R 2 available. So, P 1 can finish off and after P 1 finishes this R 2 may be given back to P 2 so, that P 2 will finish. So, as a result there cannot be any deadlock. So, this is the no preemption condition.

Next we have got circular wait, so circular wait is telling that I have got a number of waiting processes P 0, P 1 up to P n they are waiting the such that P 0 is waiting for a resource that is held by P 1; P 1 is waiting for a resource held by P 2, etcetera up to P n minus 1 is waiting for a resource that is held by P n and P n is waiting for a resource held by P 0.

(Refer Slide Time: 29:18)

So, it is like this that P 0 is waiting for so, P 0 is waiting for some resource. So, if this is the process P 0 it is asking for some resource R 0 it is the holding it is asking for some resource R 0 and R 0 is currently held by process P 1, then process P 1 is requesting for a resource R 1 and this R 1 is currently held by process P 2.

So, this way it goes on maybe process P 2 is requesting for some resource R n and this R n is currently held by P 0. So, this type of situation if it is there then you can understand that none of the processes P 0, P 1, P 2 they can proceed. So, they can, so none of them will be able to proceed because all of them are waiting in a circular fashion on some process to be over and get the corresponding resource and if proceed.

So, if the circular wait condition is there then also will have deadlock. So, if you can violate any of these conditions see the circular wait is not there then of course, there is no problem. For example, if this chain was not there then what we can understand is that P 2; so P 0 is requesting for R 0 or say this is requesting for R n.

So, I can give R n to P 2; so P 2 will finish off and once P 2 finish off this R 1 R n and R 1 and R n are free then R 1 can be given to P 1. So, P 1 will finish off and then R 0 can be given to P 0 for P 0 to complete. So, if there is no circular wait if it is only a chain of waits way of the processes then they can be satisfied very easily. So, then all that also help us in avoiding this deadlock condition, so deadlock will not occur in that situation

also. So, we will continue with this deadlock characterization and solution in the next class.