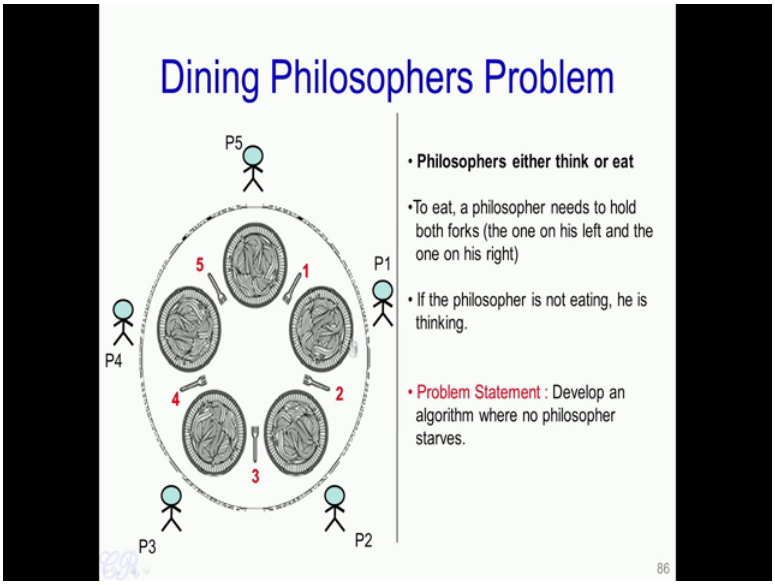
Introduction to Operating Systems
Prof. Chester Rebeiro
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Week – 07
Lecture – 30
Dining Philosophers Problem

Hello. In the previous video, we had talked about semaphores. We had seen how semaphores could be used to solve synchronization problems in the producer consumer example that we had taken.

In this video, we will look at an other problem, where semaphores are useful. So, this is the dining philosophers' problem and it is a classic example of the use of semaphores.

(Refer Slide Time: 00:54)

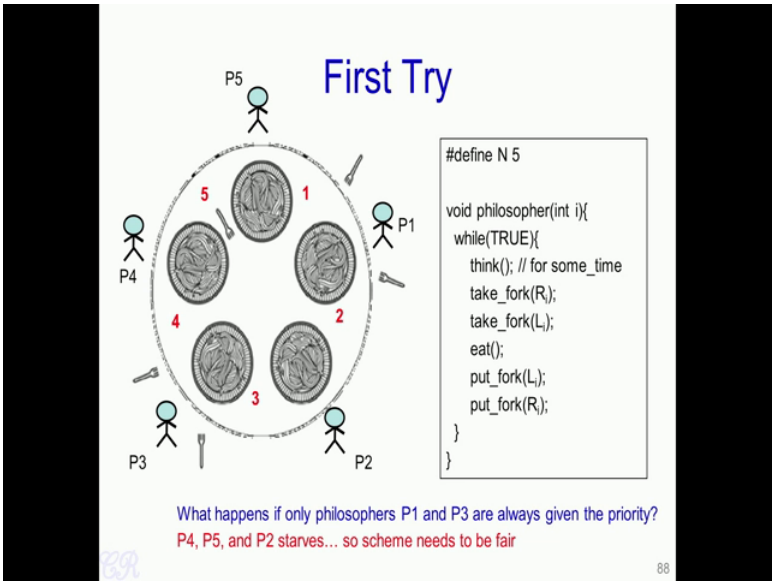


So let us start with the problem. So let us say we have five philosophers P 1, P 2, P 3, P 4, P 5 who are sitting around a table. Now in front of them, there are five plates; one for each of the philosophers and five forks 1, 2, 3, 4, 5 (mentioned in above slide image). Now each of these philosophers could do just one of two things. Each philosopher could either think or eat.

In other words, if the philosopher is not thinking then he is eating and vice versa. Now, in order to eat, a philosopher needs to hold both forks, that is the fork on his left and the

one on the right that is for instance if p 1 wants to eat then he needs to have the fork 1 and fork 2. These are the two forks, which are closes to him. Similarly, if P 3 wants to eat then forks 4 and forks 3 are required. Now the problem is or the problem what we are trying to solve is to develop an algorithm, where no philosopher starves that is every philosopher should eventually get a chance to eat.

(Refer Slide Time: 02:30)



So let us start with the very naive solution to this particular problem. So let us say we have a solution over here (mentioned above in a box), where we define N as 5 corresponding to each philosopher. And we have a function for philosopher i.e. void philosopher(int i). So, this function takes a integer value ‘i' and this ‘i' could be values of 1 to 5 corresponding to each philosopher that is P 1, P 2, P 3, P 4 or P 5.

Now in the function, we have an infinite loop (as mentioned above) where the i^{th} philosopher will think for some time and then after some time he begins to feel hungry. So, he will take the fork on his right, then take the fork on his left, then he is going to eat for some time, and after that he is going to put down the left fork, and then put down the right fork, and this continues in a loop infinitely.

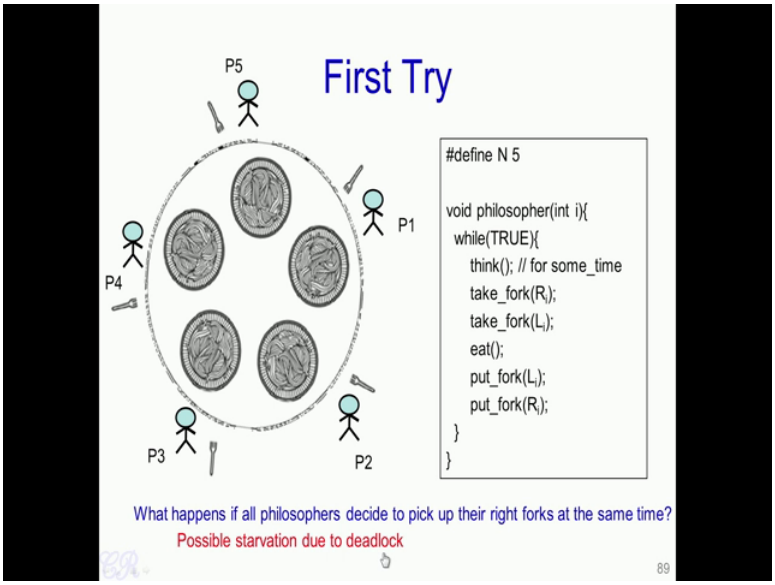
So, for instance philosopher P 1 will think for some time, then feel hungry, then he would pick up the right fork that is the fork number 1, then pick up the left fork that is fork 2, then eat for some time and put down the forks, first 2 and then 1. So, this seems like a very simple and easy solution to the problem. But as we will see that there are certain

issues that could crop up. The issues come because each of these philosophers which essentially executes this function independently, or thinking, and feeling hungry, and eating all independently.

So let us consider this particular scenario (based on above slide image). Let us say the philosophers P 1 and P 3 have a higher priority that is whenever the request for the fork to be picked up then the system will always ensure that they are given the forks. So, what would happen in such a case? So we will get a case where P 1 eats whenever he wants, and P 3 eats whenever he wants, while the other philosophers P 2, P 4 and P 5 which have the lower priority in picking the fork will not be able to eat. For instance, the philosopher P 2 neither could pick up the right fork or the left fork, and therefore, P 2 cannot eat.

In a similar manner, P 4 and P 5 have just one fork between them, which they could possibly pick up; the other fork in each case would with high probability be given to the philosophers P 1 and P 3. Thus P 2, P 4 and P 5 will starve; and this is not the ideal solution for our problem.

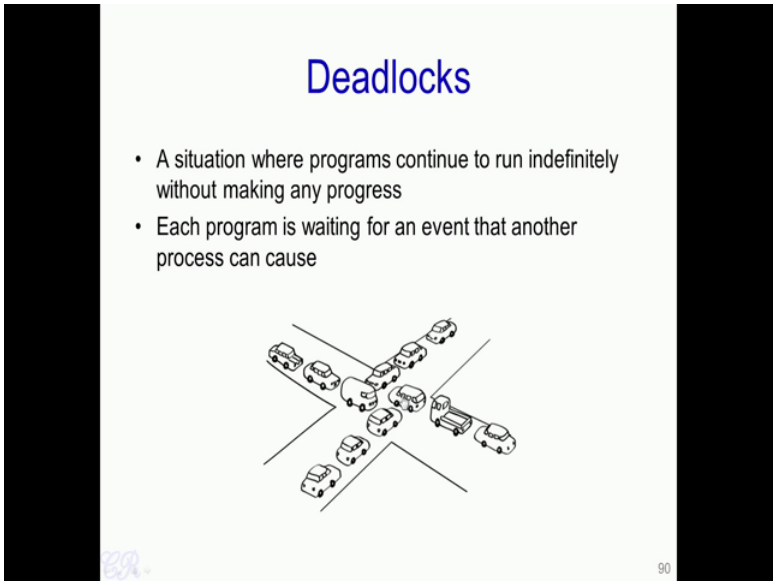
(Refer Slide Time: 05:56)



So let us see another possible issue that could take place. So let us say by some chance, all the philosophers' pickup their right fork simultaneously (mentioned in above slide image). So, we have philosopher P 1 picking up his right fork, philosopher P 2 picking the right fork, P 3, P 4, P 5 pickup the right fork respectively.

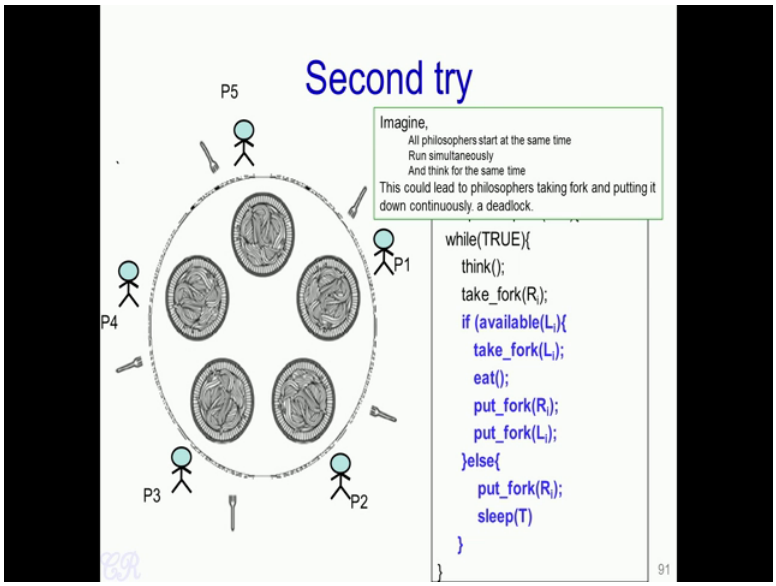
Now, in order to eat, each of the philosophers have to pick up their left fork and this could lead to a starvation. Essentially P 1 is waiting for P 2 to put down the fork, so that he could pick it up. Then P 2 is waiting for P 3 to put down the fork; P 3 is waiting for P 4 to put down the fork; P 4 is waiting for P 5; and P 5 is waiting for P 1. So, essentially we see that every philosopher is waiting for another philosopher, thus creating a chain. And this waiting will go on infinitely leading to starvation, which we often call as a deadlock.

(Refer Slide Time: 07:19)



So, to define a deadlock more formally; a deadlock is a situation, where programs continue to run indefinitely without making any progress. Each program is waiting for an event that another program or process can cause. So, you see in this case each of the philosophers is waiting for a particular event that is putting down the fork, which an other philosophers should do. So, there is a circular wait that is present and this leads to a deadlock, there by starvation.

(Refer Slide Time: 08:07)



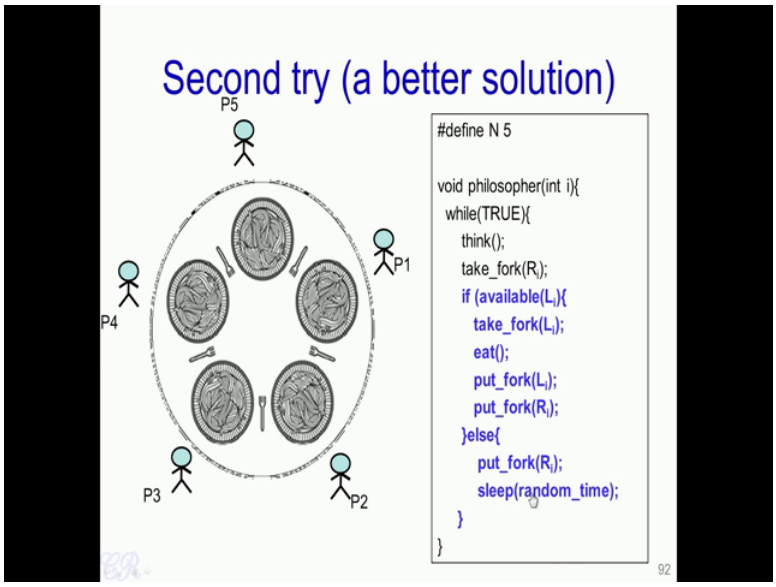
So let us look at another attempt to solve this particular problem. So let us say we have the same function over here (mentioned in above slide). And this time the philosopher takes the right fork i.e take_fork(R), then he would determine if the left fork is available (if condition mentioned above); if the left fork is available the philosopher would take the left fork i.e take_fork(L), eat for sometime i.e eat() then put down both the forks the right as well as the left fork (i.e put_fork(R) and put_fork(L), and the loop continues as usual.

However, if the left fork is not available, then we go to the else part (mentioned in above slide image) and the philosopher will put back the right fork i.e put_fork(R). Essentially, the fork which was picked up over here (initially), the right fork would be put back on to the table, if the philosopher finds out that the left fork is not available. So this will allow another philosopher to probably eat. And after this is done there is a sleep for some fixed interval T i.e sleep(T) before the philosopher tries again.

Let us see what is the issue with this particular case? (mentioned in above slide) So let us consider a particular scenario where all philosophers start at exactly the same time, they run simultaneously and think for exactly the same time. So, this could lead to a situation where the philosophers pick up their fork all simultaneously then, they find out that their left forks are not available, so they put down their forks simultaneously, then they sleep for some time and then they repeat the process. So, you see (mentioned in above slide

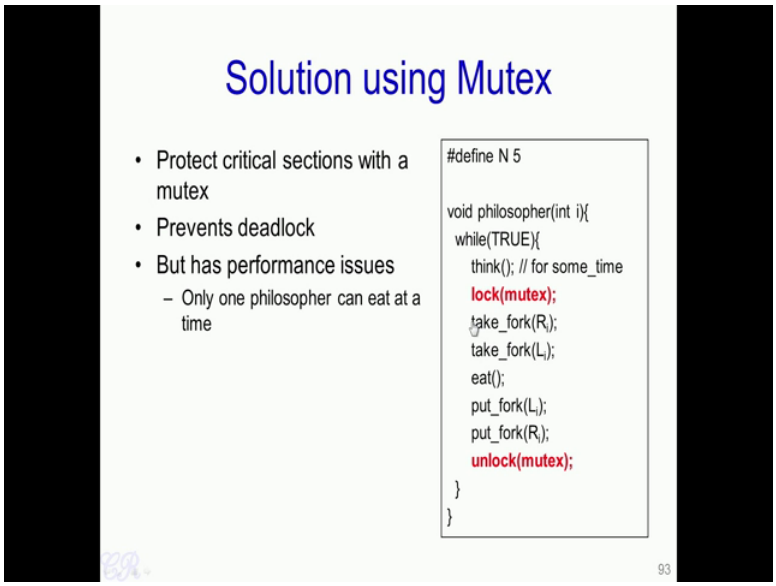
image) that the five philosophers are again starved. They will be continuously just picking up their right fork and putting it back onto the table. So, this solution is not also going to solve our purpose, since we have the philosophers starving again.

(Refer Slide Time: 10:26)



A slightly better solution to this case (as mentioned above) is where instead of sleeping for a fixed time, the philosopher would put down the right fork and sleep for some random amount of time (i.e sleep(random_time). While this does not guarantee that starvation will not occur, it reduces the possibility of starvation. Now such a solution is implemented in protocols like the Ethernet.

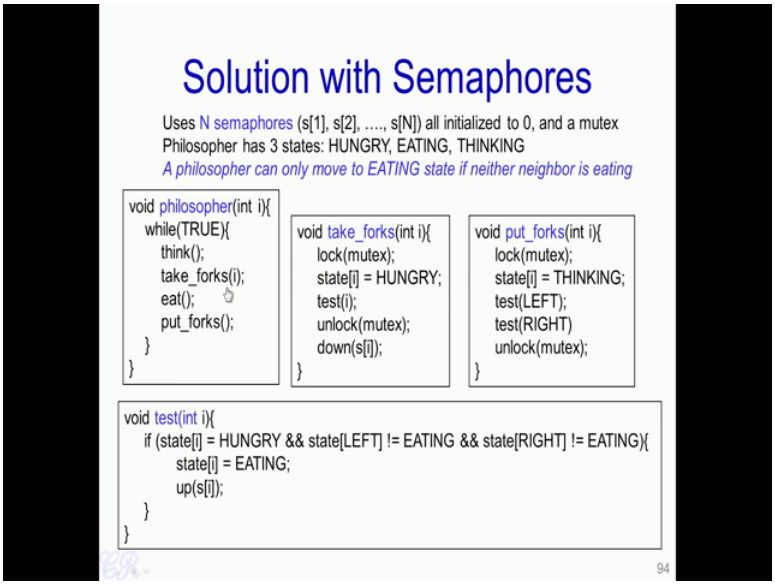
(Refer Slide Time: 10:57)



So let us look at a third attempt to solve this particular problem. So, this particular solution uses a mutex (as mentioned in above image). Essentially before taking the right or the left fork, the philosopher needs to lock a mutex. And the mutex is unlocked only after eating and the forks are put back on to the table. So, there is a lock mutex over here before picking the forks i.e lock(mutex) and an unlock mutex after the forks are put down onto the table i.e unlock(mutex). So, this solution essentially ensures that starvation will not occur, it prevents deadlocks.

However, the problem here (in above slide) is that because we are using a mutex, so at most one philosopher can enter into this critical section (i.e in between lock and unlock mutex). In other words, at most one philosopher could eat at any particular instant. So, while this solution works, it is not the most efficient solution. So, we would want something which does much better than this.

(Refer Slide Time: 12:20)



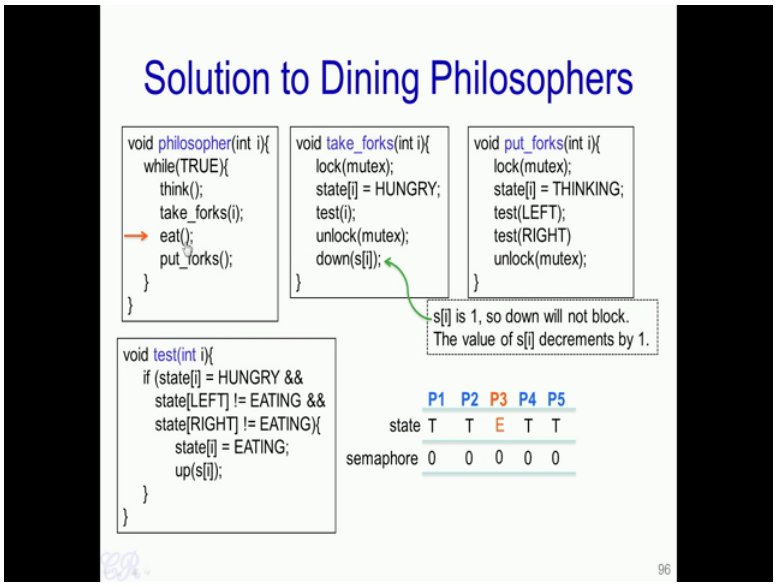
So let us look at our 4th attempt, and this one using Semaphores. So let us say that we have N semaphores; so the semaphores s[1] to s[n] and we have one semaphore per philosopher. So all these semaphores are initialized to 0; in addition the philosopher can be in one of 3 states – hungry, eating or thinking. So, over a period of time, each philosopher will move to one of these states. For instance when the philosopher is thinking, the state will be thinking; then the philosopher becomes hungry, so it goes to the hungry state then eating, and then back to thinking, and this process continuous till the eternity.

So, the general solution that we will be seeing here (as mentioned in above slide) is that a philosopher can only move to the eating state if neither neighbor is eating. That is a philosopher can eat only if its left neighbor as well as its right neighbor is not eating. So, in order to implement this particular solution, we have 4 functions. So, first is the philosopher i.e void philosopher(int i), which is the infinite loop and corresponds to the philosopher 'i'. So the philosopher will think, then it will take forks, then eat for some time and put down the forks and this repeats continuously.

Now, in the take forks function (mentioned in above slide), first we set that the philosopher is in a hungry state. The state of the philosopher is set to hungry then the function called test is invoked i.e test(i). So, what test will do is that it's going to check whether the state of the philosopher is hungry and as well as the state of the philosopher

to the left as well as to the right is not in the eating state (refer above slide image for test function). If this indeed is true then the philosopher can eat. And at the end, after eating, the forks are put down i.e put_forks() and the state of the philosopher goes to thinking.

(Refer Slide Time: 15:06)

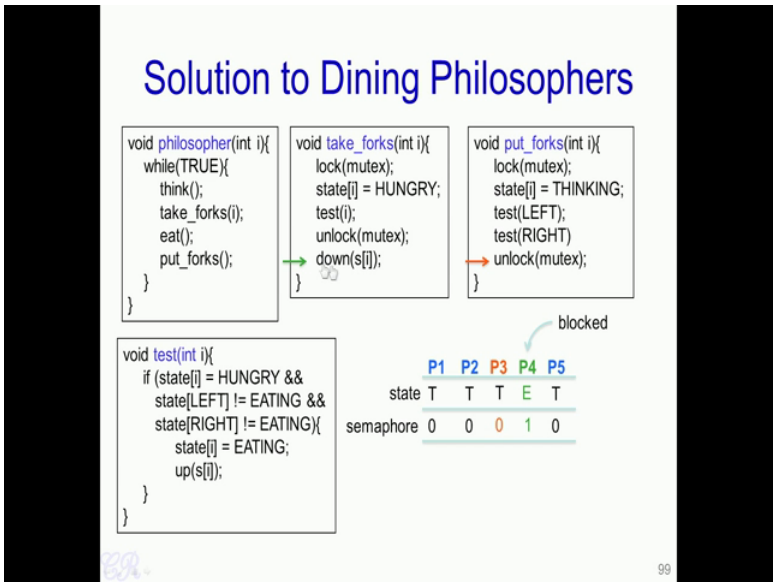


So let us look at this, particular functions more in detail (mentioned in above slide image). Let us say that these are the five philosophers P 1, P 2, P 3, P 4, and P 5. And let us say initially that all of them are in the thinking state so this is represented by the T over here (in the slide image). And as we know their initial value for the semaphores are all 0. So let us say all the philosophers are in the thinking state. And then philosopher 3 goes from the thinking state and it goes to the take fork; and as a result the take fork function i.e take_forks(3) for philosopher 3 gets invoked, and the state of the philosopher then changes from thinking to hungry i.e state[3] = HUNGRY.

Then the function test is invoked i.e test(3), and this condition is checked (if condition mentioned in test function); essentially the state of philosopher 3 is hungry. So this condition is true then the state of the left philosopher is not eating, because P 2 is in the thinking state as well as the right philosopher is not eating, because P 4 is also in the thinking state. So, as a result, this condition (if condition) goes to true and the state for philosopher 3 is set to eating, then the corresponding semaphore is incremented from 0 to 1. So the test function returns and thats the mutex which gets unlocked, and then there is the down.

So, down as we know would check the value of s[i], and if this value is less than or equal to 0 it is going to block or loop infinitely. And if the value is greater than 0, which is the case over here (as mentioned above), then it just decrements the value of s[i], so thus here s[i] had the value of 1, so the down will not block, but rather it is just going to decrement the value of s[i] to 0. So, s[i] or the corresponding semaphore has a value of 0. And then the philosopher 3 can go to the eating state and consume his food.

(Refer Slide Time: 17:35)



Now, let us see another situation, where philosopher 4 moves from the thinking state to the take forks state. Now take forks get invoked for philosopher 4 here i.e take_forks(4), and there is a lock mutex, the state is set to hungry for philosopher 4 i.e state[4] = HUNGRY and there is test(i) which is invoked. So, in the test(i) function, we see that the first condition is met for philosopher 4, because P 4 is indeed in a hungry state; however, the state left is in the eating state. Therefore, this entire if condition (mentioned above in test function) evaluates to false, and therefore, execution does not enter this if loop, rather it just skips the if part of it. Now we go back to the unlock and down.

So what we see now is that the semaphore value corresponding to P 4 has a value of 0. So, when down is invoked (mentioned inside take_form function) as we know, it would lead to the process getting blocked. So, thus philosopher P 4 will get blocked, so this P 4 will continue to be blocked as long as P 3 is eating. Then after a while P 3 decides to put down the forks (i.e put_forks function executed), and sets it state back to the thinking

state. Then it would invoke test with the left philosopher i.e test(LEFT) this is with respect to P 2, which in our case it is not really interesting so we will not look at that. But what is interesting is the test right i.e test(RIGHT) and the right here is corresponds to P 4 so this will invoke here.

And what we see is the state of P 4, so remember that test of right so this is invoked with i having the value of 4 (function test(4) invoked from test(RIGHT). Thus we see that the state of ‘i’ is hungry, because P 4 is hungry. The left and the right is not eating, therefore, this condition (if condition mentioned above in test function) will be set to true and execution comes into this if. Consequently, the value of state for the philosopher 4 would be set to eating and the semaphore value is set to 1 for P 4.

Now setting the value of 1 for the semaphore will cause the wakeup to occur. So P 4 which was blocked on the semaphore would wake up. And as a result of this (i.e down(s[i]) the value of the semaphore gets decremented to 0. Thus the P 4 philosopher would wake up and start to eat.

Thus, we see that the semaphores had efficiently ensured that the different philosophers could share the five forks, which they have in common. And it will ensure that every philosopher would get to eat eventually. So, there are other variance and solutions for the dining philosophers' problem which are very interesting to read, but we will not go into that for this particular course.

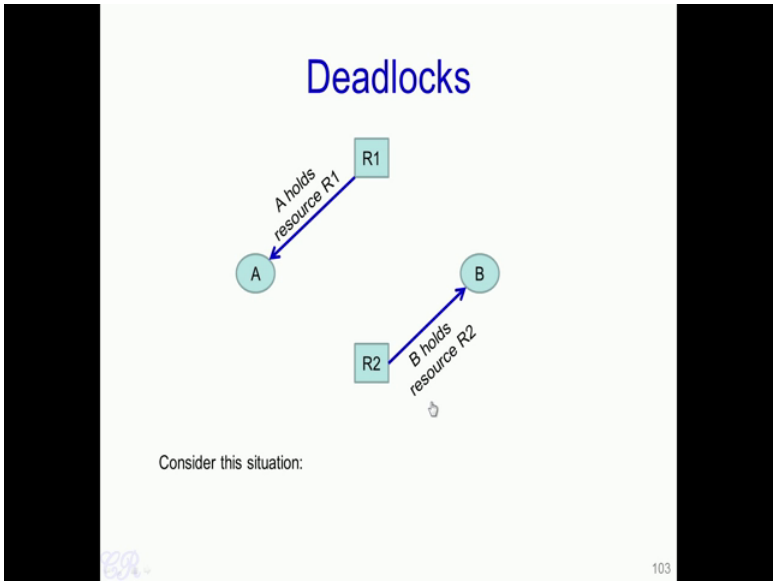
Thank you.

Introduction to Operating Systems
Prof. Chester Rebeiro
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Week – 07
Lecture – 31
Deadlocks

Hello. In the previous video, we had seen the dining philosophers' problem. And we had seen for the initial naive solutions that we had for the problem, it could result in something known as deadlocks. So, in this video, we will look at more in detail about deadlocks, and how they are handled in the system.

(Refer Slide Time: 00:39)

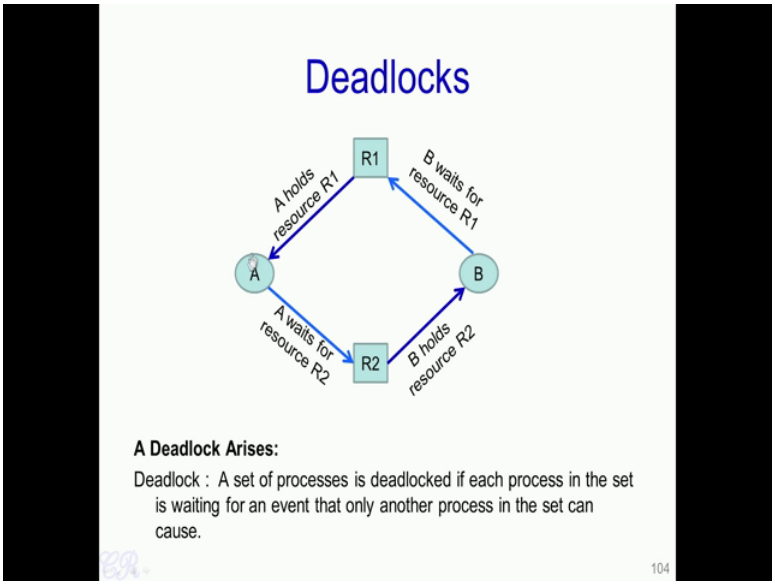


So let us say we have two processes A and B, and we have two resources R 1 and R 2 (as mentioned in above slide). So, these resources could be anything in the system which have a limited quantity. For example, the resources could be as something as small as a file or stored in the disc or it could be a printer which is used to print or a plotter, a scanner and so on.

So, essentially the arrow from R 1 to A indicates that A is currently holding the resource. For instance, if R 1 is the file; that means, A has currently open the file exclusively and is doing some operations onto the file. In a similar way, the resource R 2 is held by B. So if

it is a printer, for instance, if R 2 is a printer, it means that B is currently using the printer to print some particular document.

(Refer Slide Time: 01:52)



Now, consider this particular scenario (as mentioned in above slide) where the process A holds the resource R 1, and process B holds the resource R 2; but at the same time, process A is requesting to use R 2. So, essentially the process A is waiting for R 2 to be obtained; and process B is waiting for the resource R 1 to be obtained. So, to take an example, process A is opened the file and is using a particular file which is stored in the disk.

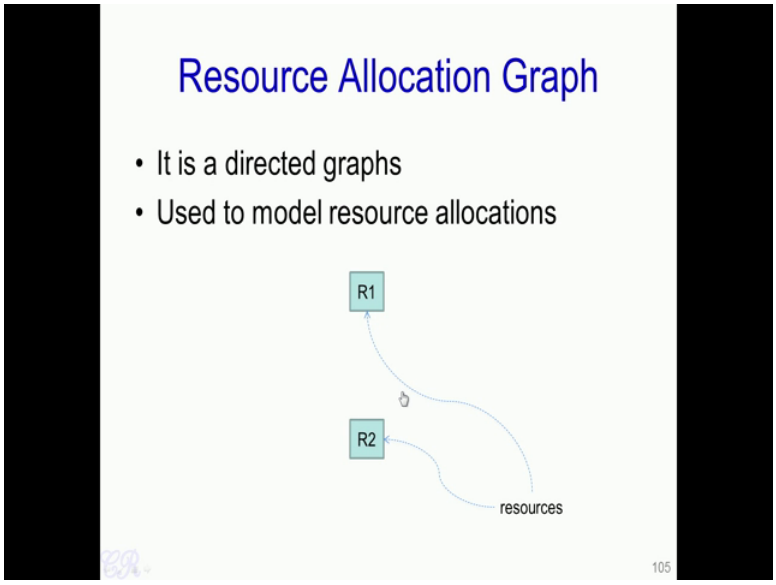
And at the same time, for instance, it wants to print the file to the resource R 2 which we assumed was a printer. Now in a similar way, process B is currently holding this resource (i.e printer) that is using this particular resource (R2) and it wants to open and utilize this particular resource R 1 (i.e file).

So, what we see that over here (as we discussed above), we have a scenario called a Deadlock. Essentially, a deadlock is a state in the system where each process in the deadlock is waiting for an event that an other process in that set can cause. For instance, over here (as mentioned in above slide), the process A is waiting for the resource R 2 which is held by B; B in turn is waiting for R 1 which is held by A. We have a set of two processes A and B; and each process in the set is waiting for the other process to do something and each process in the set is waiting for the other process to do a particular

thing. For example, A is waiting for B to release this particular resource R 2, while B is also waiting for A to release the resource R 1.

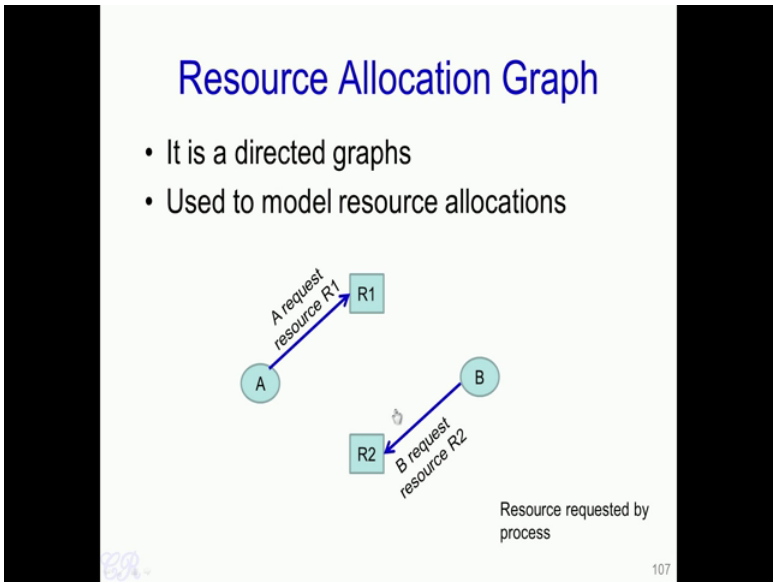
So, deadlock like this is a very critical situation that would occur in systems. And when this deadlocks occur, it could lead to process A and B in this case waiting for an infinite time continuously waiting without doing any useful work. So, such deadlocks should be analyzed thoroughly. So, in this particular video, we will see how such deadlocks are handled in systems. Now, in order to study deadlocks, we use graphs like this (mentioned in below slide), these are known as Resource Allocation Graphs.

(Refer Slide Time: 04:55)



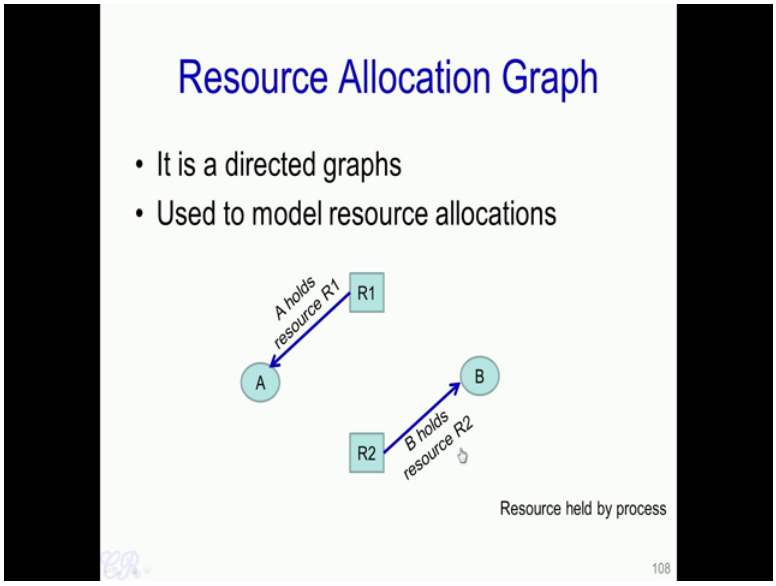
Now resource allocation graphs or directed graphs used to model the various resource allocations in the system. And there by determine whether a deadlock has occurred or a deadlock is potentially going to occur and so on. So, in this directed graph, we represent resources by a square. So, instance R 1 and R 2 are resources and they are represented by the square as shown over (mentioned in below slide). In a similar way, circles as shown over here (as mentioned in below slide) are used to represent processes.

(Refer Slide Time: 05:37)



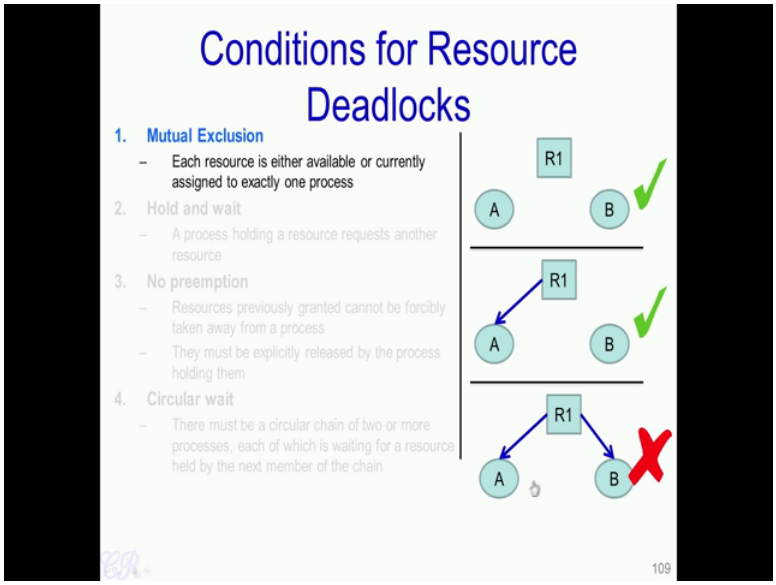
And as we have seen before, arrows from the process to the resource that is directed from the process to the resource would indicate that a request is made for that resource. For example, over here the arrow from A to R 1 indicates that A is requesting for resource R 1; similarly B, in this case is requesting for resource R 2 (mentioned in above slide). So, these requests are made to the operating system and if possible the operating system will then allocate that resource to the corresponding process.

(Refer Slide Time: 06:22)



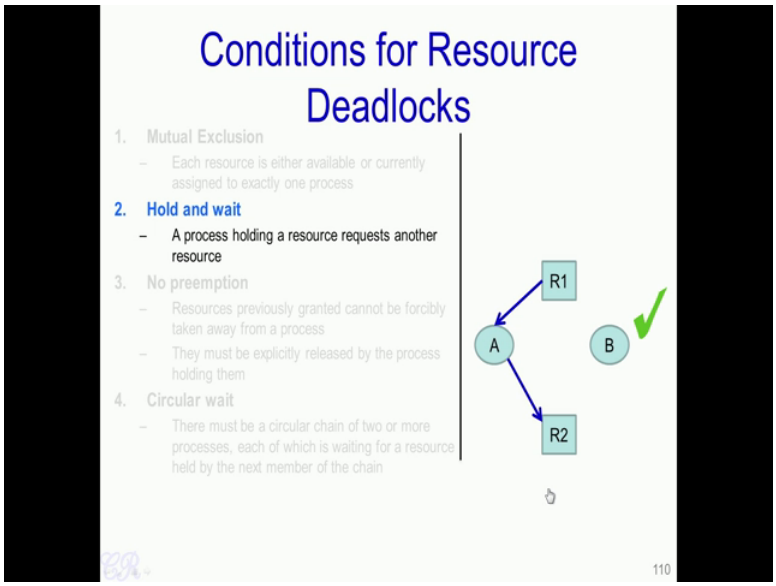
So, when that happens the graph will look like this (as mentioned in above slide), essentially the direction of the arrow has changed. Now the arrow moves from R 1 to A, indicating that A holds resource R 1. Similarly, the arrow from R 2 to B indicates that B holds resource R 2. There are four conditions in order that a deadlock occurs. So, we will now look at each of these conditions for a deadlock.

(Refer Slide Time: 06:59)



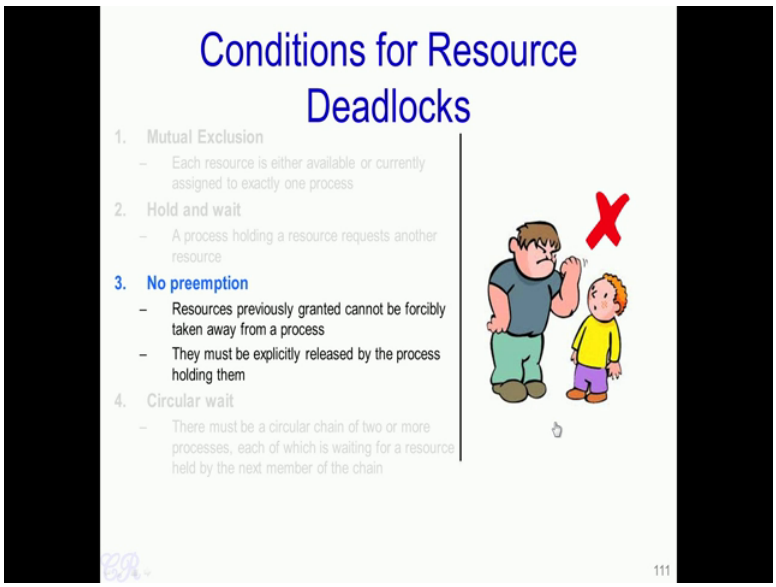
So, the first is Mutual Exclusion. So, what we mean by this is that each resource in the system is either available or currently assigned to exactly one process. So, for instance over here (refer to above slide image) we have resource R 1 which is free (1st figure). So, it is not assigned to any particular process, so this is fine. While this is also fine (2nd figure) where the resource is allocated to exactly one process, but in order that deadlocks happen this kind of scenario (3rd figure) should not be present that is the resource cannot be shared between two processes A and B.

(Refer Slide Time: 07:48)



The next condition for a deadlock is Hold and wait (refer above slide image) that is a process holding a resource can request another resource that is for example, in this case (figure mentioned above) the resource R 1 is held by process A. and while having R 1, A is also requesting for another resource R 2, so it essentially holding R 1 and waiting for R 2.

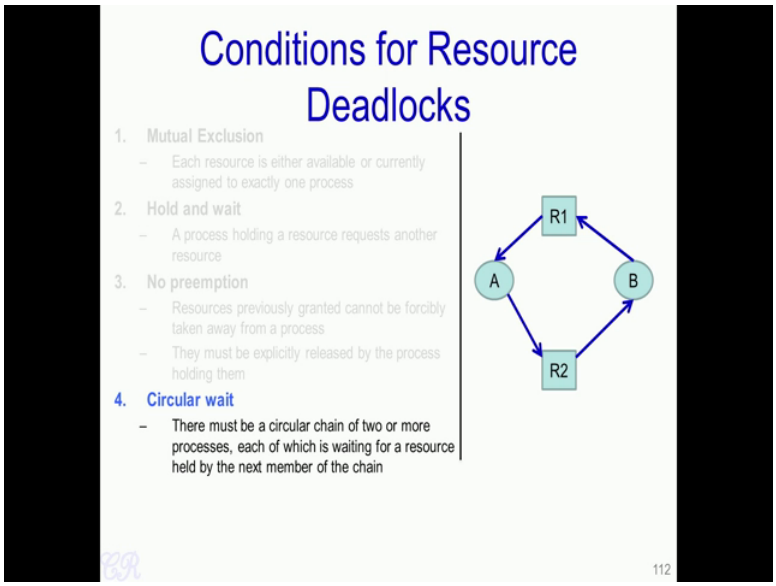
(Refer Slide Time: 08:22)



The third condition for a deadlock to happen is No preemption (refer above slide image). Essentially, it should not be the case that resources which an operating system previously

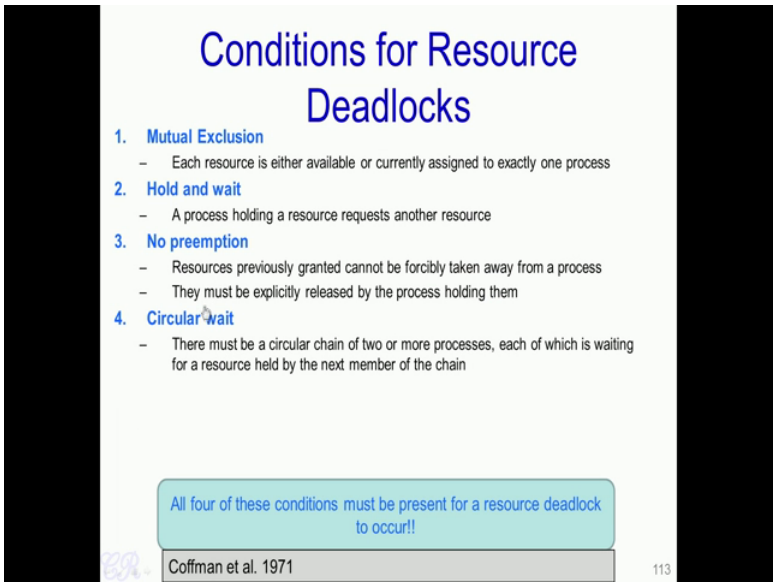
granted for a particular process is forcibly taken away from that process. That is the OS or another entity in the system cannot forcibly remove a resource which has been allocated to a particular process. Instead processes should explicitly release the resource by themselves that is whenever the process wants it should release the resource by itself.

(Refer Slide Time: 09:18)



So, a fourth requirement is the Circular wait (refer above slide image). What this means is that there is a circular chain of 2 or more processes, each of which is waiting for a resource held by the next member of the chain. So, we see over here (mentioned in above circular figure) with that we have a circular chain and there is a wait over here because process A is waiting for process B to release resource R 2; and process B is in turn waiting for process A to release the resource R 1. So, we have a circular wait condition over here.

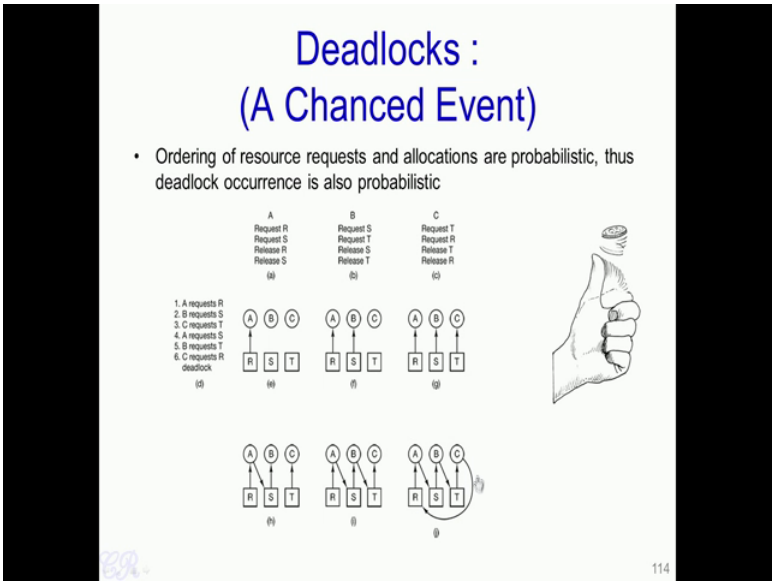
(Refer Slide Time: 10:00)



So, these 4 conditions: mutual exclusion, hold and wait, no preemption and circular wait must be present in the system in order that a deadlock could occur. So, if for instance, we were able to build a system where one of these conditions, were not present. For example, suppose we build a system where processes cannot hold a particular resource and wait for another resource at the same time. So, such system would never have any deadlocks.

On the other hand, suppose a system has been developed where all of these things are possible (as mentioned in above slide) that is there is a mutual exclusion when using resources, a process could hold a resource and wait for another one, once allocated they cannot be forcefully preempted from the resource and circular wait mechanisms are allowed then deadlocks could potentially occur. So, having all these conditions does not imply that a deadlock has occurred. It only implies that there is a probability of deadlock occurring in the future.

(Refer Slide Time: 11:33)



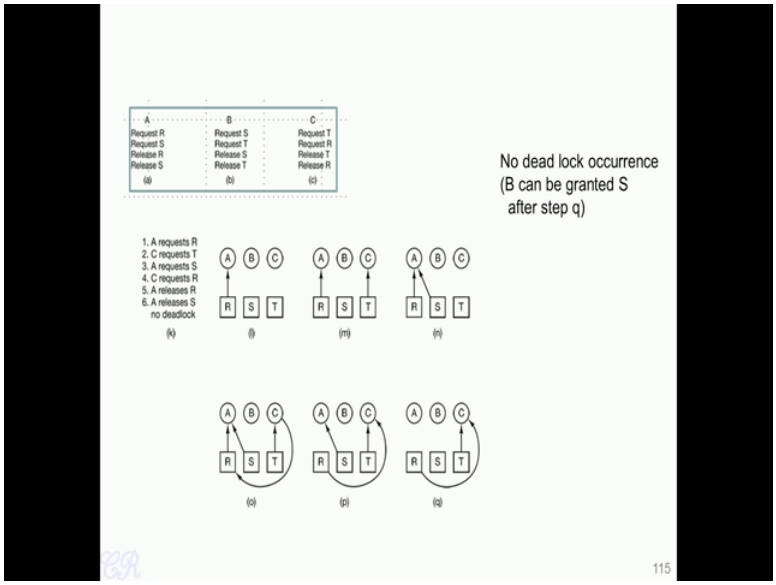
So, this being said "a deadlock in a system is a chanced event". Essentially, it depends on several factors such as the way resources are requested by processes, the way allocations are made for these resources, the way de-allocations are made by the operating system and so on. So, only if certain order of these requests an allocation happen and only then will a deadlock occur. So, a small variation in the request and allocations may cause the deadlock to not occur.

So let us see some examples of this (mentioned in slide time: 11:33). So let us say we have 3 processes in the system A, B and C, and there are 3 resources as well R, S and T. So, each process could request and release resources at sometime during its execution. So obviously, a release can be made by a process only after the request is made. So, these request and release of resources are given to the operating system at various time instance, depending on how A, B and C get scheduled and how they are executed.

So let us consider this particular sequence that A requests R, and then B requests S, C request T. Then A requests S, B requests T, and C requests R. So, this is one potential order for how request occur. So, we can use our resource graphs or resource allocation graphs to view this (mentioned in slide time 11:33). So, we see that A requests R and the operating system will then allocate the resource R to A, then corresponding to B requests S, the allocation will be of S will be to B. Then C requests T and the OS will allocate T to C.

Then A requests S, so there is a line like this (arrow pointing to S from A). B requests T there is a line over here (arrow pointing to B from T) and C requests R. So, we have the four conditions that we have seen in the previous slide, have all been met. For instance, the circular wait you see is achieved here. R is held by A, while A requests S; S is held by B and at the same time B request T. Now T is held by C, while simultaneously C is requesting for R. So, you see that each process is waiting for an other process in this set to release a particular resource. So, we have a deadlocked scenario over here.

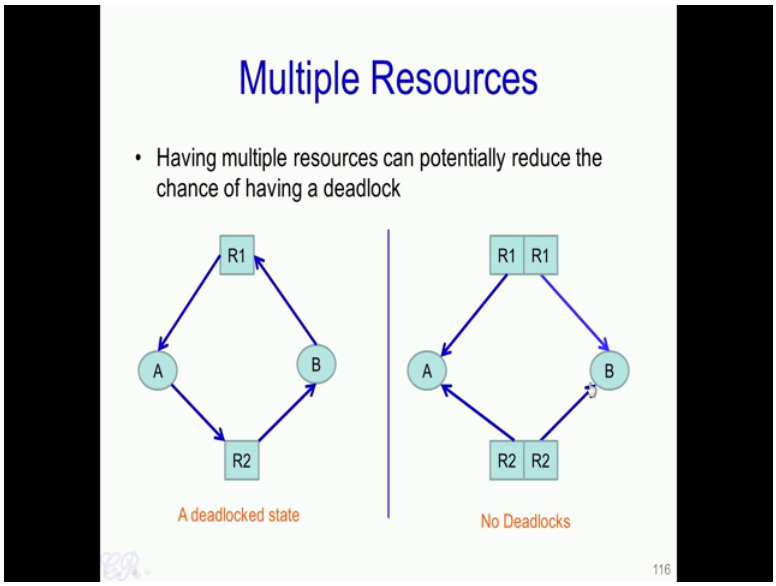
(Refer Slide Time: 14:55)



Now, we will see that if the same requests and allocations are done in a slightly different manner then the deadlock will not occur. For example, R is allocated to A (fig. (l) as above), and then T gets allocated to C (fig. (m) as above); then A requests S and gets allocated to A (fig. (n) as above). Then C requests R, so we have this over here (mentioned above in fig.(O)). Now, A releases R, so there is no more line over here (fig. (p) as above) or an edge over here then A releases S, and therefore R can be allocated to C (fig. (q) as above).

So, you see that depending on the requests and releases, we are able to achieve a situation where each and every request or release can be serviced by the operating system. So, in such a scenario, we have not obtained the deadlock.

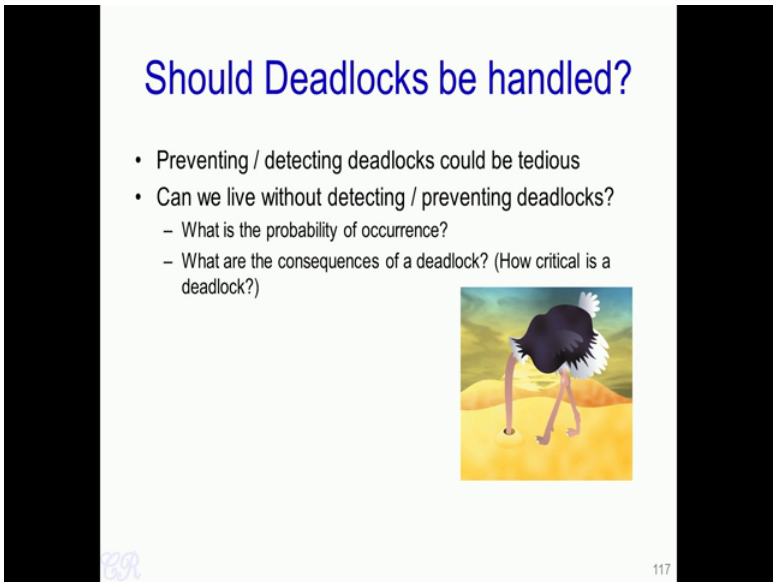
(Refer Slide Time: 16:07)



Now, we have seen that deadlock is indeed a probabilistic event and could occur with some probability. Now one way to reduce the probability is by having multiple resources present in the system. So, we had seen that this was a deadlock state (mentioned in above slide left side), essentially because A is waiting for resource R 2 to be released by B, and B in turn is waiting for resource R 1. Now one way this can be solved is by having multiple resources, while this particular solution will not always work, and essentially depends on the type of resources, it may help to some extent.

For instance, if we have 2 resources of exactly the same type then both A's request as well as B's request could be managed that is the resource if we have 2 types of R 1 or in other words if you have a duplicate of the resource R 1 then that can be given to A as well as B. Similarly, a duplicate of the resource R 2 can be given to A and B simultaneously (mentioned in above image right side). So, what this means is that for example, we could have 2 printers present and A can be allocated one printer while B could be allocated the other printer while this does not completely eradicate deadlocks, it may reduce the likelihood the deadlocks may occur.

(Refer Slide Time: 18:00)



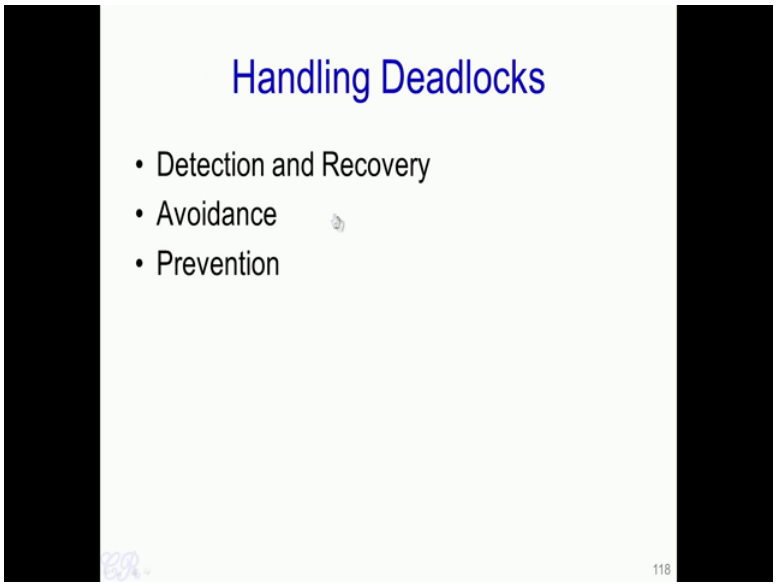
Now, the next question is in a system which could have deadlocks, should deadlocks be handled? So, this is a debatable question, essentially is not an easy thing to answer, because the cost of having a prevention mechanisms or to detect deadlocks is extremely high, and it will cost huge overheads in the operating system.

So, the other aspect was known as the ostrich algorithm is to completely ignore that deadlocks could occur and run the system without any prevention or any deadlock detection mechanisms. So, either having some deadlock prevention or detection mechanisms or just ignoring the entire aspects of deadlock would need to be made during the OS design time or rather the system design time. So, various things need to be discussed before a decision can be made, such as what is the probability that a deadlock occurs? Is it likely that a deadlock will occur every week, or every month or once in 5 years or so on?

Second what is the consequence of a deadlock? Essentially, how critical a deadlock could be? For instance, if a deadlock occurs on my desktop, I could simply reboot the system and it's not going to affect me much. On the other hand if a deadlock occurs, say in a safety critical application like a spacecraft or a rocket kind of scenario, then the consequence could be disastrous. Therefore, we need to argue about these two aspects, essentially if the probability that a deadlock occurs is very frequent then probably the OS would require some measures in order to handle the deadlock.

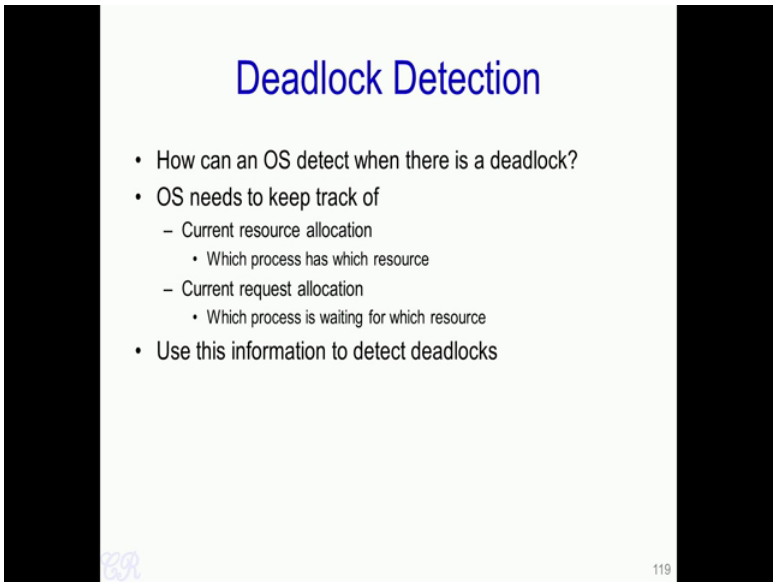
On the other hand, if the deadlock occurs very sporadic may be on average once in 5 years or so then you may not require to have or handle a deadlock in the operating system. So now let us assume that we need some mechanism in our operating system to handle deadlocks. So, what can we do about this?

(Refer Slide Time: 20:47)



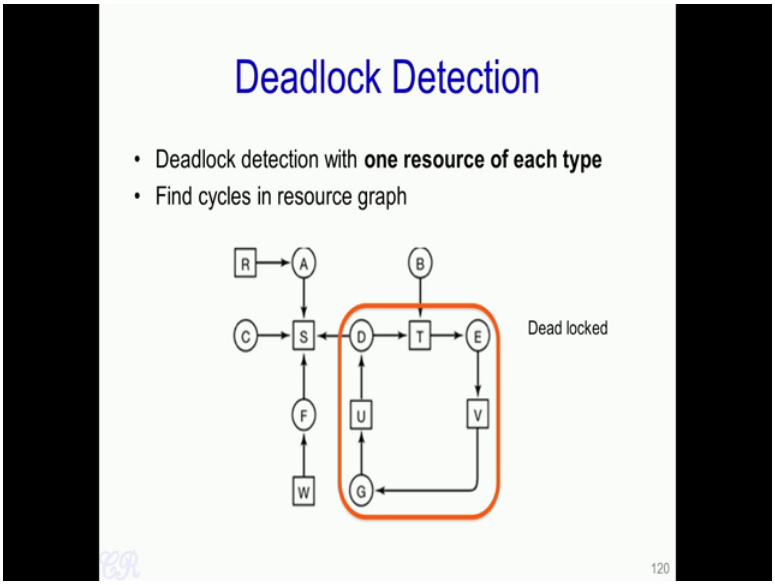
Essentially, there are three ways that deadlocks can be handled. One is by detection and recovery, second by avoidance and third by prevention.

(Refer Slide Time: 21:06)



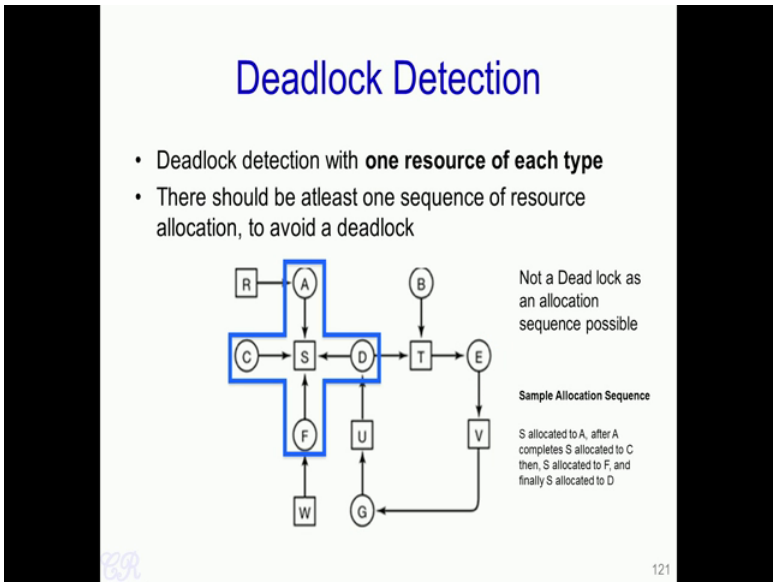
So let us look at the first case, that is detection and recovery and first how are deadlocks detected? Essentially for the operating system to detect deadlocks, it requires to know the current resource allocation in the system. Essentially which process holds which resources and it also requires to know the current request allocation by the processes. Essentially which process is waiting for which resources, and the OS will then use this information to detect if the system is in a deadlocked state.

(Refer Slide Time: 21:46)



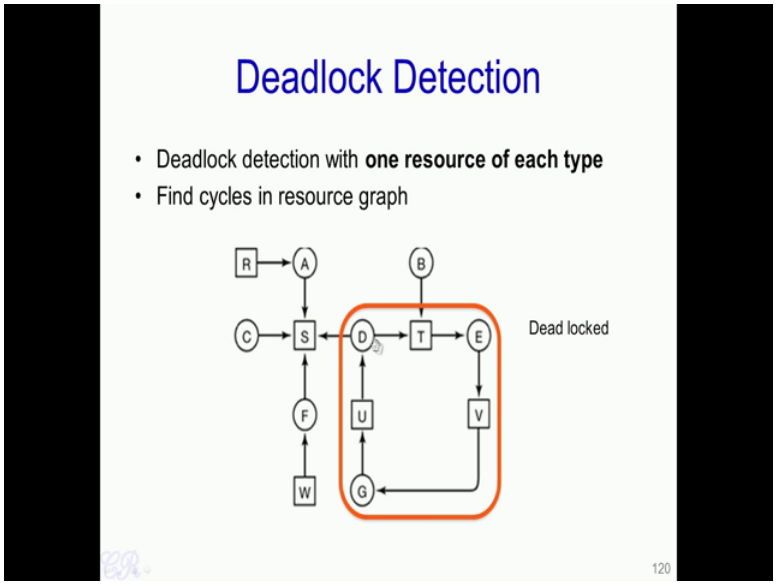
So, the way the detection could work is by finding cycles in the resource allocation graphs. For instance, if this is the resource allocation graph (mentioned in above slide image) for the various resources in the system, the OS will detect a cycle present. For example, over here (processes inside orange box), we have a cycle between the processes D, E and G then the OS will say that these 3 processes are indeed in the deadlock state.

(Refer Slide Time: 22:19)



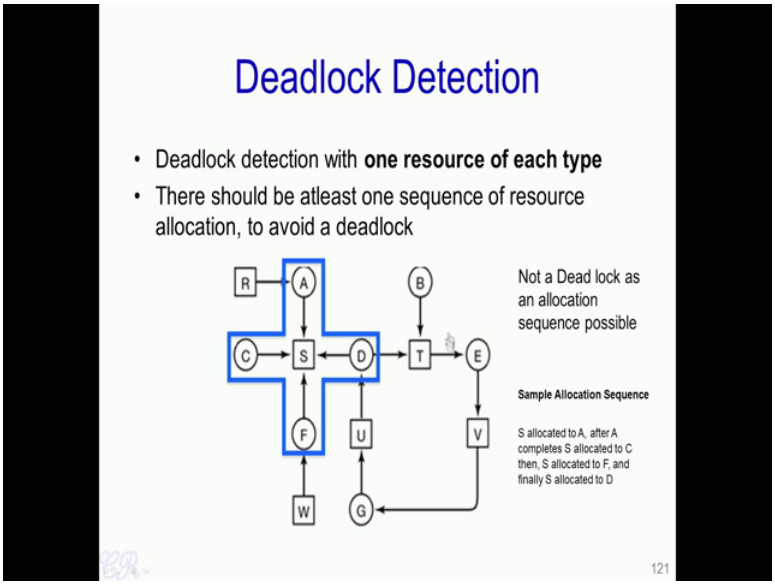
Now the other aspect is there could be request as shown over here (mentioned in above slide in blue cross), where there is resource S and it is requested simultaneously by A, D, F and C. So, we see that this is not in a deadlock state because there is some sequence of allocation of S to all these processes. For instance one possible allocation came for S is that first S be allocated to A then A will use the resource S for some time, and then after it complete using S, S can be then allocated to C, and after which S is allocated to F, and then D. So, essentially the allocation of S could be sequential among these 4 processes. So, this will not have a deadlock.

(Refer Slide Time: 23:18)



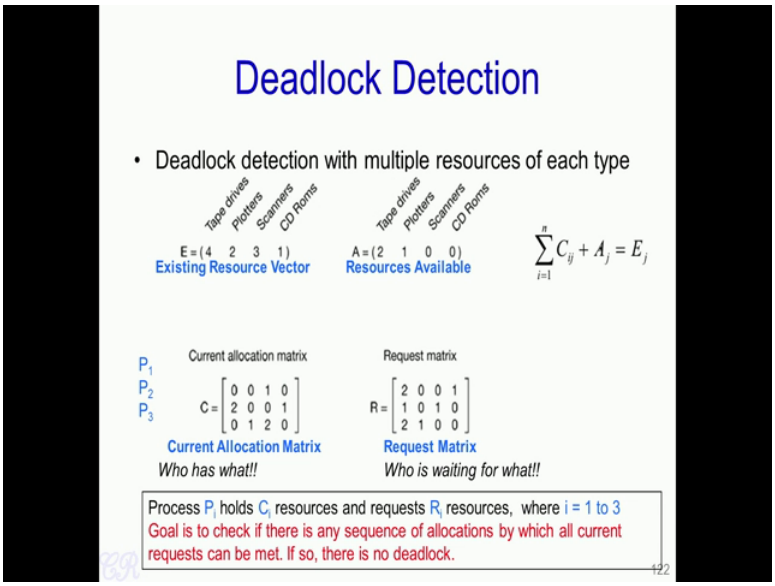
However, the presence of a cycle in the resource allocation graph will indicate that a deadlock is present in the system.

(Refer Slide Time: 23:28)



This technique of finding cycles in the resource allocation graph would work well with systems where there were one resource of each type. Now suppose we had systems where there were multiple resources of each type then another algorithm would be required.

(Refer Slide Time: 23:45)

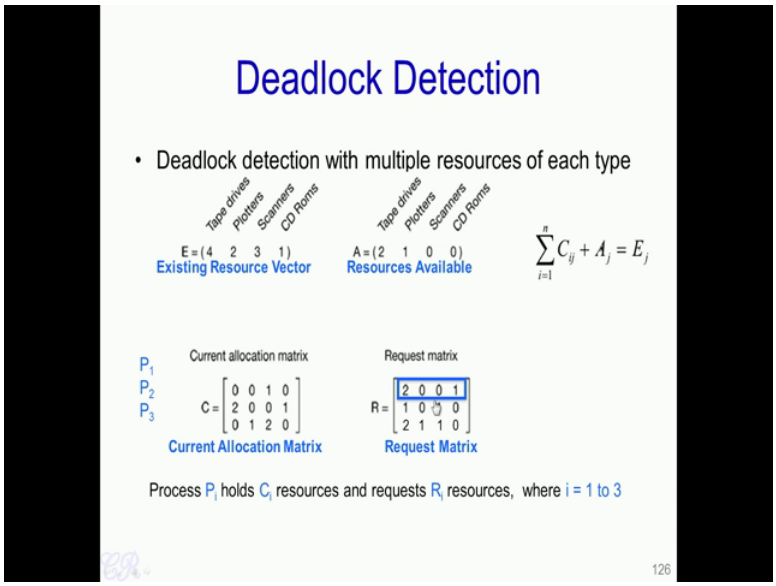


So let us give an example of how that would work (mentioned in slide time 23:45). Let us say in a system we have 4 resources: tape drives, plotters, scanners and CD ROMs. Further, there are 4 tape drives, 2 plotters, 3 scanners and 1 CD-ROM. Then let us also say that we have 3 processes executing in the system P 1, P 2, P 3, and this is the current allocation matrix (refer above slide). So, the row 1 is with respect to process P 1. This means that the process P 1 does not have any tape drives, does not have any plotters, is allocated 1 scanner and no CD-ROMs. Similarly, process 2 is allocated 2 tape drives and 1 C D ROM (2^{nd} row of matrix) and process 3 is allocated 1 plotter and 2 scanners (3^{rd} row of matrix).

Now, this particular set A determines the resource available (i.e A = (2 1 0 0); essentially out of the 4 tape drives that we have, if you look into this corresponding column over here in the current allocation matrix (refer above slide image) we have seen that 2 is used. So, what remain is 4 - 2 that is 2 tape drives are free and available to be allocated. So, similarly if you look at plotters, out of the 2 plotters 1 is allocated and 1 is free. Out of the 3 scanners, we see that all 3 scanners are allocated, so it is 0 over here (i.e in resource available set). Similarly, there are no CD-ROMs also available to be allocated.

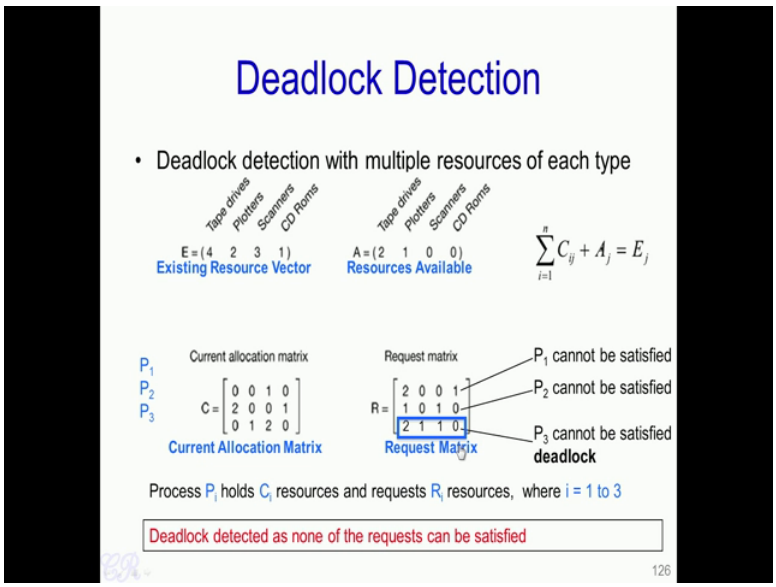
Now, in addition to this we have a request matrix (mentioned in above slide), essentially which process is waiting for what, is represented in this matrix. For example, process p 1 requests 2 more tape drives and 1 CD ROM (1^{st} row in request matrix). So, process P 2 requires 1 tape drive and 1 scanner (2^{nd} row of matrix), and process P 3 requires 2 tape drives and 1 plotter (3^{rd} row of matrix). Now the goal of having such definitions or such representations of the resources allocations and requests is to determine if there exists a sequence of allocations of these requests, so that all request can be met. If such a case is possible, then there is no deadlock present. Let us see if we can allocate these requests by the 3 processes.

(Refer Slide Time: 26:52)



So let us take a process P 1 and let us see (refer above slide request matrix) the request by process P 1 can be met. So, it requires 2 tape drives and we see that 2 tape drives are available. So, this is fine and 1 CD-ROM is requested, but there are no CD-ROMs is available. So, process P 1 cannot execute. So, the process P 1 cannot be allocated all its resources. So, it cannot continue to execute.

(Refer Slide Time: 27:17)

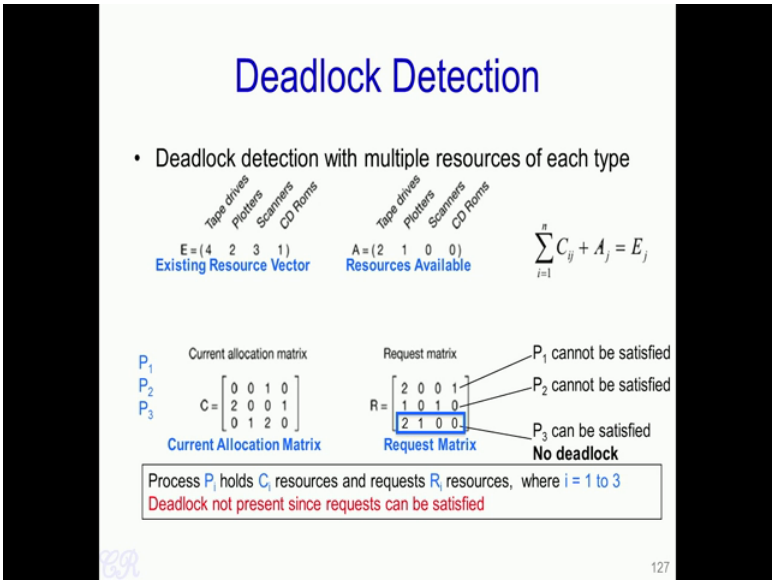


Let us see for about process P 2. So, process P 2 requires 1 tape drive which can be allocated to it because it is available; and it requires 1 scanner, but 0 are available. So,

similarly, process P 2 cannot be satisfied as well, and process P 2 will also need to wait. Now let see the 3rd case where for process 3, there are 2 tape drives which can be met, 1 request for a plotter which can be met and 1 request for a scanner which cannot be met (mentioned in above slide image).

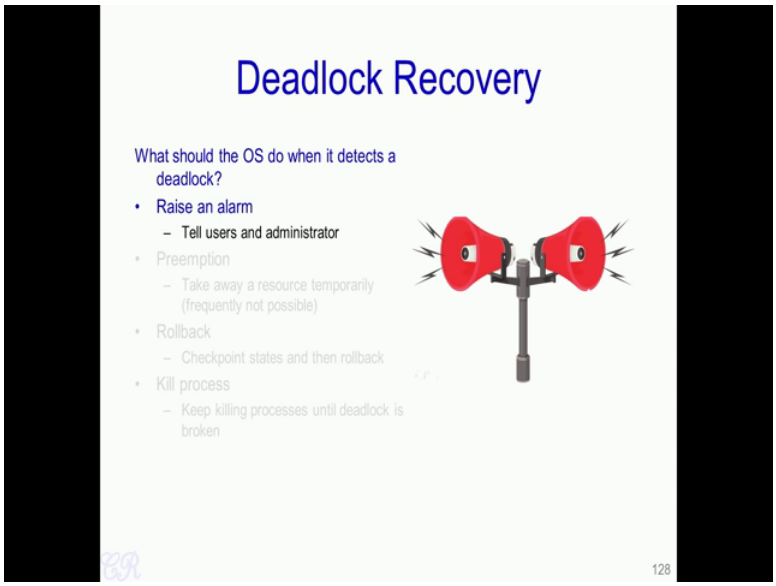
So, process P 3 also cannot be satisfied its request. So, P 3 also will wait. So, P, 1 P 2 and P 3 are all waiting for the request, and therefore the state is in a deadlock because none of the requests by any of these processes can be met and therefore, all the processes will need to be waiting.

(Refer Slide Time: 28:34)



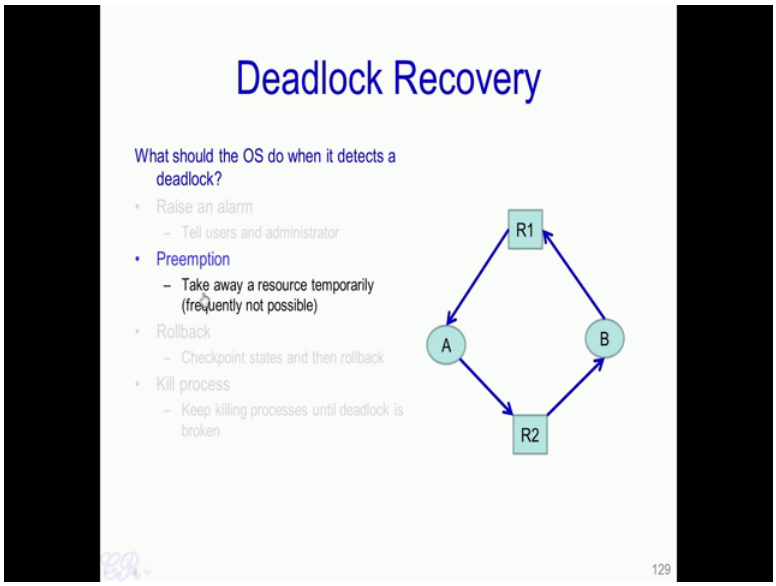
If you look at another example where there is a small change (mentioned in above slide image). We have just reduced the number of scanners required by a process P 3 (mentioned above in 3rd row of request matrix). So, in such a case, we see that both the tape drives requested by process P 3 can be met by the system, the single plotter can be also met and there are no scanners and no CD-ROMs that are required. And therefore, we see that all the request can be allocated to process P 3, therefore P 3 can be satisfied and we do not have a deadlock over here. So, in this way, deadlocks can be detected by the operating system, based on the current allocation matrix and the request matrix.

(Refer Slide Time: 29:32)



So, once the deadlocks is detected, what next should the operating system do? So, there could be various things that the OS could do. So, one thing is that it could raise an alarm, that is tell users and administrator that indeed deadlock has been detected.

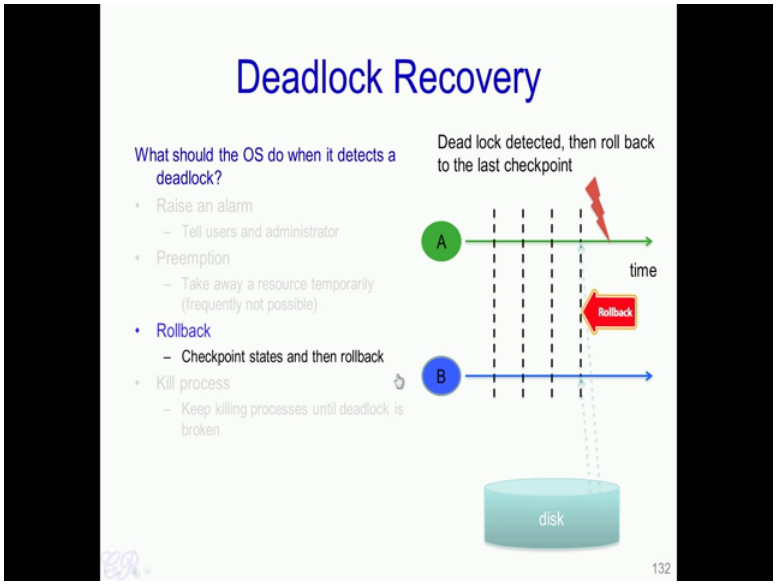
(Refer Slide Time: 29:51)



Then a second way is to force a preemption that is you force a particular resource to be taken away from a process and given to another process. For instance, it could be like R 2 was a printer and it is currently held by B (refer above slide image). So, what could be done was that the printer could be forced to be taken away from B, while it is allocated to

A for some time and thus the deadlock will be broken, this is as shown over here. So, B no longer has other resource R 2, but R 2 is given to A.

(Refer Slide Time: 30:42)



Then a third method is by using a technique known as rollback. So, with rollback, both processes A and B as they execute will be check pointed (mentioned in above slide image). So, by check point we mean that the state of the process gets stored on to the disk. The process executes for some time and then the entire state of the process gets stored on to the disk. Now storing the state of the process on the disk, will allow the processes to execute from the point where it has been check pointed, now the checkpoint state. So, as time progresses more check points are taken periodically as shown over here (dotted lines in above image).

Now, let us say a deadlock is detected after sometime (mentioned in above slide image) then what could happen is that the system could rollback to the last non deadlock state that is over here (last dotted lines). So, how the rollback occurs would be by loading the state of process A and B in this particular example to the last known state as shown over here (last dotted lines in above image).

Now process A and B will continue to execute from this state (last dotted lines in above image) and the deadlock may not occur again. Essentially we have seen that since deadlock is a probabilistic event by modifying the ordering in which the allocations are made we could prevent this particular deadlock.

A fourth way is to kill processes. Essentially, if process A and B are in a deadlock state, killing one of these processes would break the deadlock. So, typically the lower important or the less priority process would be killed.

Thank you.

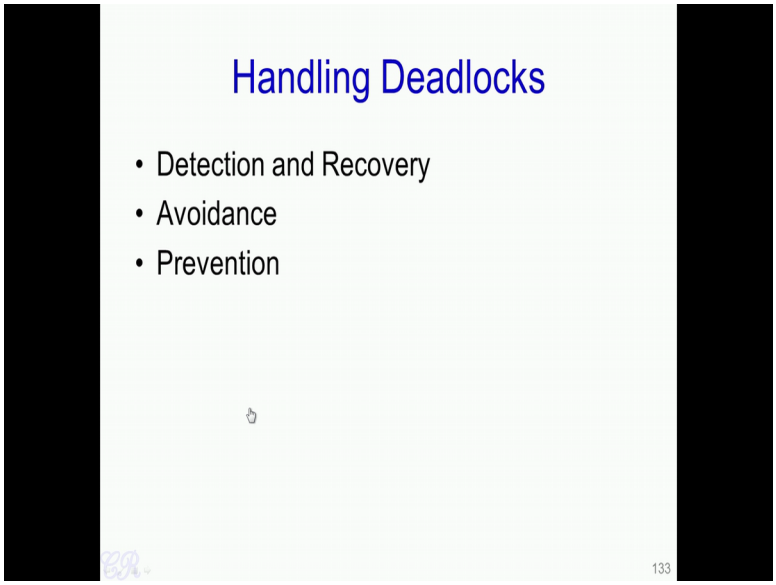
Introduction to Operating Systems
Prof. Chester Rebeiro
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Week - 07
Lecture - 32
Dealing with Deadlocks

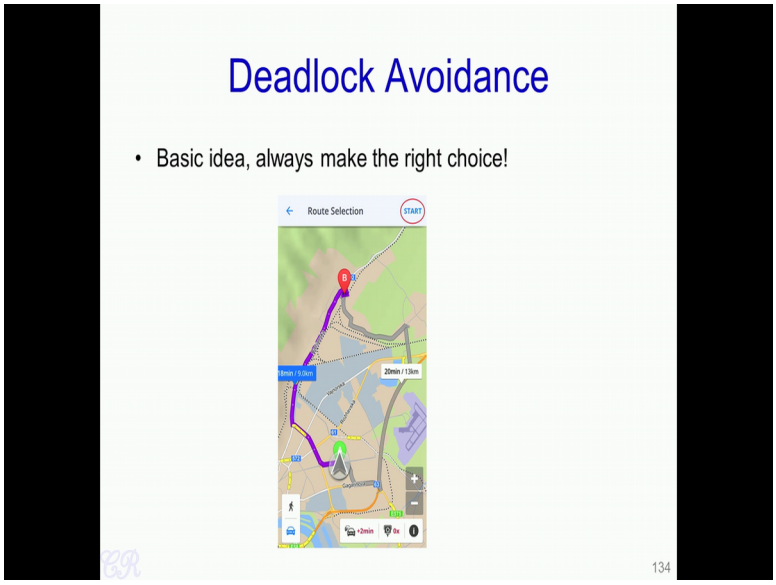
So there are 3 ways to handle Deadlocks. That is by Deduction and Recovery from Deadlocks, Deadlock Avoidance and Deadlock Prevention.

In this video, we will have a look at Deadlock Avoidance and Prevention. By avoidance we mean that, the system will never go into a state which could potentially create a deadlock situation. To have an example about how avoidance algorithms work, we will just take very simple example.

(Refer Slide Time: 00:53)



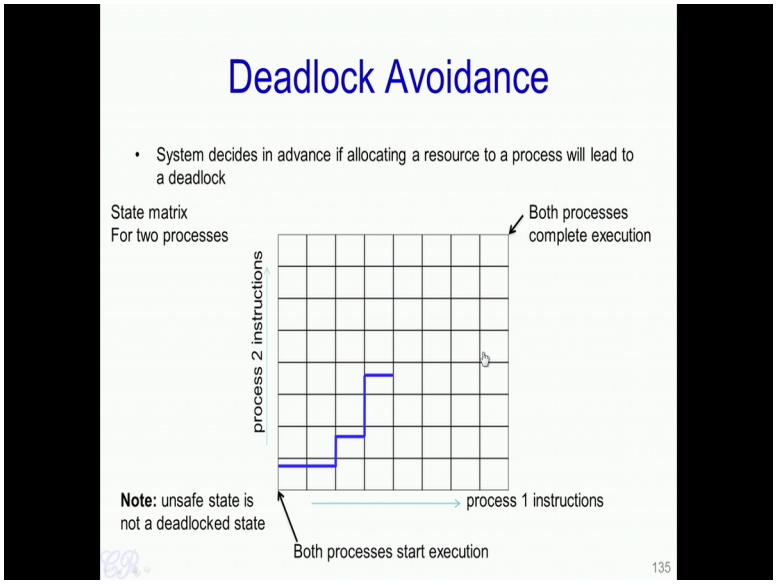
(Refer Slide Time: 00:56)



So, let us say we want to go from this point A (map's blue dot in above slide image) this is a place, to this point B (red mark) and as usual there are multiple routes which could take us there. So, this is one particular route (gray color route) and this is another route (purple color route). Now we need to make a choice or rather the right choice that will take us to this particular place B in a safe way. For example, if we may go through this route (using purple route) it may be dangerous or may have some particular blocks on the road.

On the other hand going through a third way may have a similar situation. So, out of the multiple ways which we can go from one point to another, we need to make a choice of which path we need to go. Deadlock avoidance algorithms works in very similar ways. Let us take a particular example with two processes.

(Refer Slide Time: 02:09)



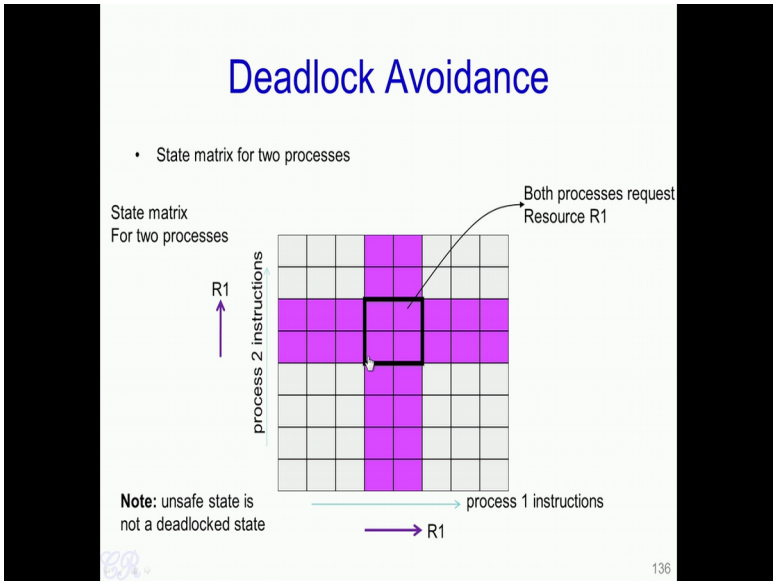
Process 1 and process 2, and we will represent the state of the system by this particular graph (refer above slide image). Now on the x axis, are the instructions executed by process 1 and the y axis corresponds to the instructions executed by process 2 with respect to time; so both are to with respect to time. Thus, process 1 and process 2 begin to execute from the origin that is this point over here (arrow mentioned in bottom of the image) and they complete execution at the other end that is the point over here (arrow mentioned on top). Now, as these process executes in the system the state of the system will change and the state changes depending on how these two processes execute with respect to each other.

So let us say this state is represented by each of these blocks over here. Now, this line over here indicates the execution path. For example, over here (blue line indicating in above graph) the line is horizontal indicating that instructions' corresponding to process 1 gets executed. So, as a result of process 1 executing while process 2 not executing, we get a state of the system over here which is shown by this (first square) small square, then the state is shifted to this as shown over here (second square) and then at this particular point (first vertical blue line in graph) there is a context switch and process 2 executes.

So as a result of process 2 being executed, the state of the system gets shifted to this point. Then process 1 executes again and the state changes, then process 2 get executed, process 1 and so on (refer blue graph line). And at the end we will have both process 1 as well as process 2 reaching this particular point (arrow on top in above slide); essentially both processes have completed execution.

Now, you see depending on how the operating system schedules, how various resources are allocated or de-allocated to process 1 and process 2, this particular trace or rather the execution trace of the process 1 with respect to process 2 would vary. For example, in the worse case it could be the process 1 executes continuously till it completes and then process 2 executes completely till it completes or another technique could be that process 1 executes a few instructions, then process 2 executes the few instructions, then process 1, process 2, process 1 and so on, till both of them complete.

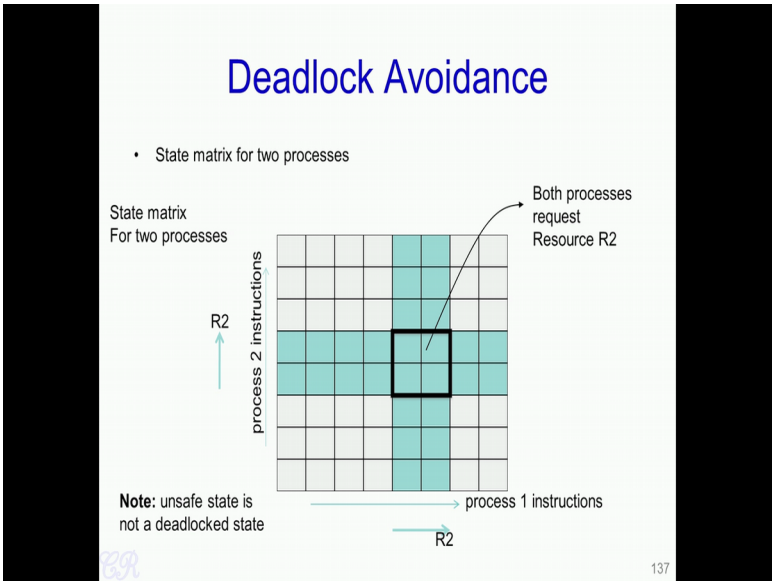
(Refer Slide Time: 05:08)



Now, let us also state that process 1 and process 2 requires a resource R1 during its execution. So, process 1 requires the resource R1 at this particular time. So, this particular time over here which is shown at this point (horizontal R1 arrow mentioned in above slide image), so, these instructions in process 1 requires the use of resource R1. So, this has been highlighted by this purple block over here (refer slide time 05:08).

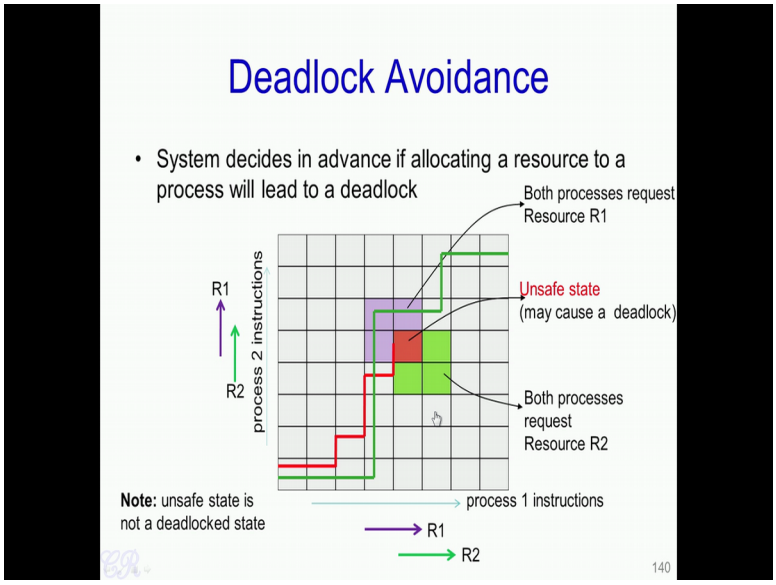
Now, in a similar way during this time from this point to this point (vertical R1 arrow in above slide), process 2 requires the use of resource 1, so this has been highlighted over here (black square in above slide graph). Now this intersecting square is the region where both process 1 as well as process 2, require resource 1. So, this intersecting square represented by the intersection of these two regions corresponds to the instructions in both processes that request resource R1.

(Refer Slide Time: 06:15)



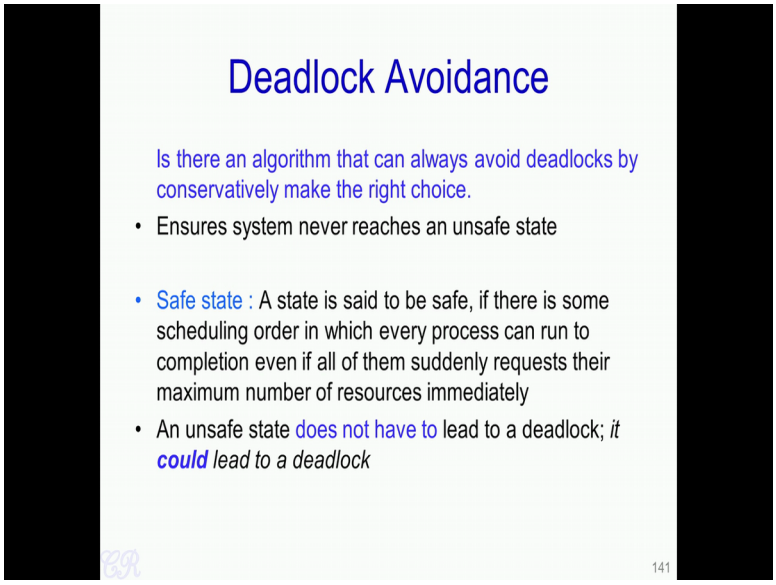
Similarly, let us say that there is another resource R2 which is required in these particular instructions by process 2 (vertical R2 arrow in above slide) and these particular instructions in process 1 (horizontal R2 arrow in above slide), and correspondingly obtained a intersecting square as shown here (blue square in above slide graph) and this square represents the part where both processes p 1 and p 2 request resource R2. Now, what we will see is the intersecting part of R1 and R2 resources. So, if we actually intersect this graph (refer slide time 05:08) with this graph (refer slide time 06:15), we get the points where both processes request R1 that is this purple part here (refer below slide image), and this part (3 blue blocks) is where the both processes require R2, and the intersecting area is where both processes require both resources R1 and R2 (red block in below slide image).

(Refer Slide Time: 06:58)



Now, this area is which may potentially cause a deadlock (red block in above image). So, we call this as an unsafe state. So, an unsafe state is not a deadlock state, but rather it is a state which may potentially cause a deadlock. Thus, if the OS was to schedule the process 1 and process 2 in such a way that the execution path ends up in this unsafe state then, we may potentially have a deadlock. On the other hand, if the OS schedules process 1 and process 2 execution in such a way that this unsafe state is avoided then the deadlock is avoided. So, this is the essential technique used to avoid deadlocks in systems.

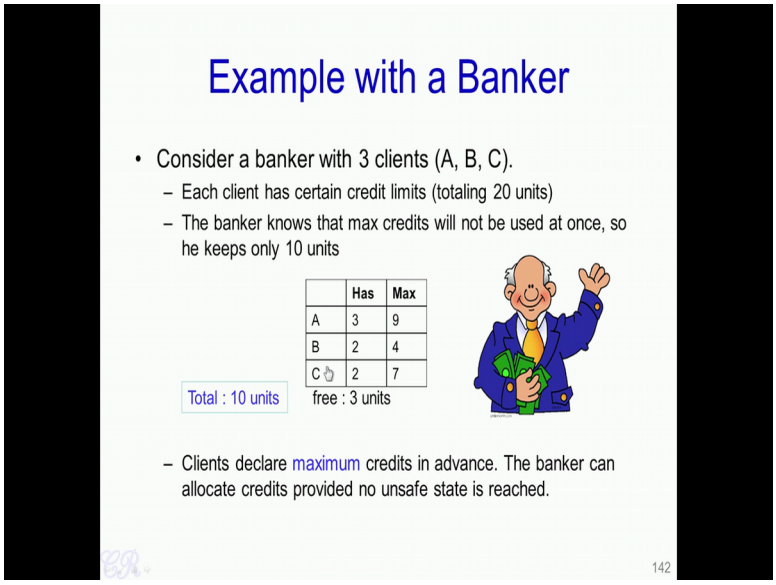
(Refer Slide Time: 08:33)



So the next question is - is there an algorithm that can always avoid deadlocks by conservatively making the right choice? That is can we ensure that the system never reaches an unsafe state by making some choices in the way processes execute or the way resources are allocated and so on.

So, one way that we can build such an algorithm is by using what is known as a Banker's algorithm. So, let us see an example with this.

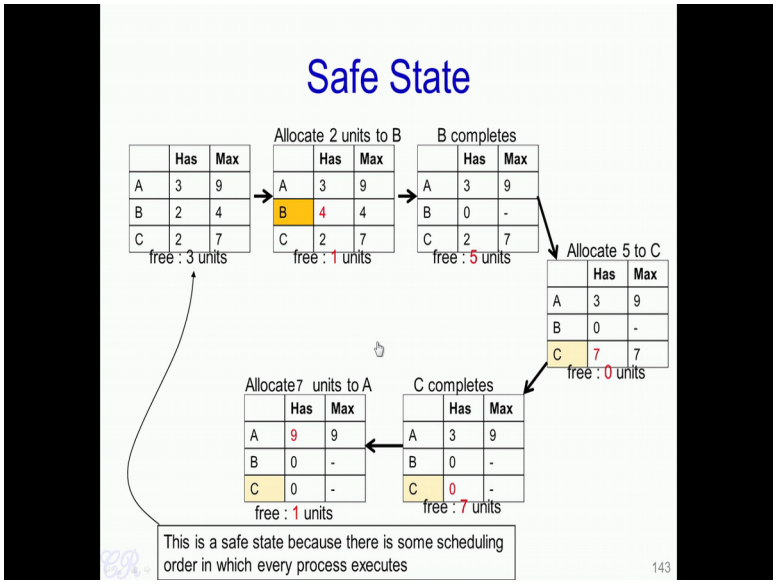
(Refer Slide Time: 09:12)



Now, consider a banker with 3 clients A, B and C (refer above slide image). Each client has certain credit limits and the sum of all these credit limits totals to 20 units. Now the banker knows that the maximum credits of 20 units will not be used all at once so he keeps only 10 units. To see what this means, let us say we consider this table (mentioned in above slide image) with the 3 clients A, B and C and each client has in his account some units. So, for instance A has got 3 units, B has got 2 units and C has got 2 units, so the total of 7. And since, the banker only keeps 10 units so the remaining 3 units are free and not allocated to any of the clients.

On the other hand, the maximum credits for each client is 9, 4 and 7 respectively (refer slide time 09:12). So, you note that 9 + 4 +7 is the maximum credit limits which totals to 20 units. And this maximum credit limits is declared by the client in advance. Now, given this particular table the banker should be able to allocate the maximum credits to users in such a way that no unsafe state is reached. So, let us see this with examples.

(Refer Slide Time: 11:07)

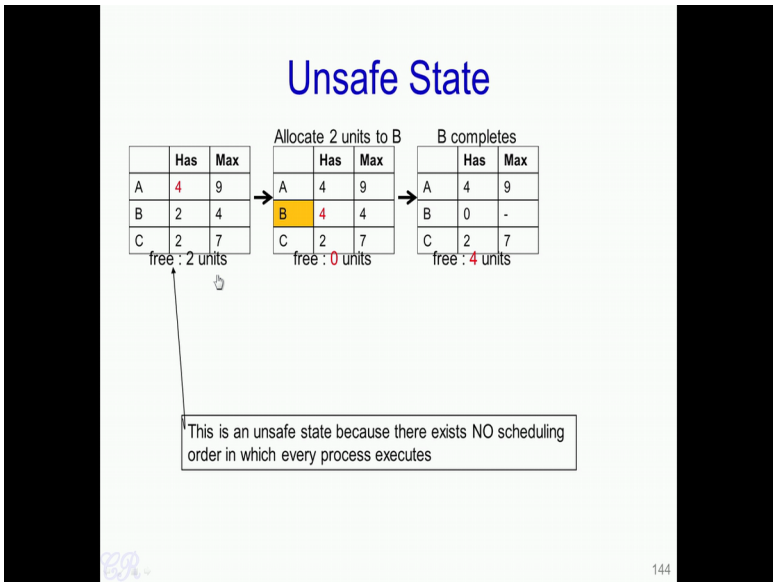


So, the problem we are trying to solve here is whether these allocations maximum of 9, 4 and 7 can be made to A, B and C in such a way that the state is safe or rather such an allocation can be made. So, let say we start with 2 units being allocated to B (mentioned in above slide image). So, we have 3 units in the free store, out of these 3 units - 2 are allocated to B. Therefore B has now 2 + 2 that is 4 units, and the free store now has 3 - 2 that is 1 unit free (refer table "Allocate 2 units to B" in above image). Now, after B completes all the 4 units are returned to the free store (refer table "B completes"), so 4 + 1 will be 5 units present in the free store and these 5 units can then be allocated to C. So, C will get its maximum credit limit of 5 + 2 that is 7 units as seen over here (refer table "Allocate 5 to C"). So, the amount present in the free store is 0 units.

Now, after C completes all the 7 units of C get returned back to the free store (refer table "C completes"). Now the banker could allocate the resources required by A that is the maximum resources required by A that is of 7 more units to get its maximum credit limit of 9 units. So, we have 9 units allocated to A and 1 unit which is free (refer table "Allocate 7 units to A"). Thus we see with this particular start point or this particular state of the system, where there are 3 units free and there is maximum of 9, 4 and 7 units as the credit limit of each client can have (refer first table in above slide). We see that with this particular state there is a schedule which is possible so that all requests and all

maximum requests by the clients can be fulfilled. So, we call such a state as a Safe State and the banker could go ahead and make such allocations.

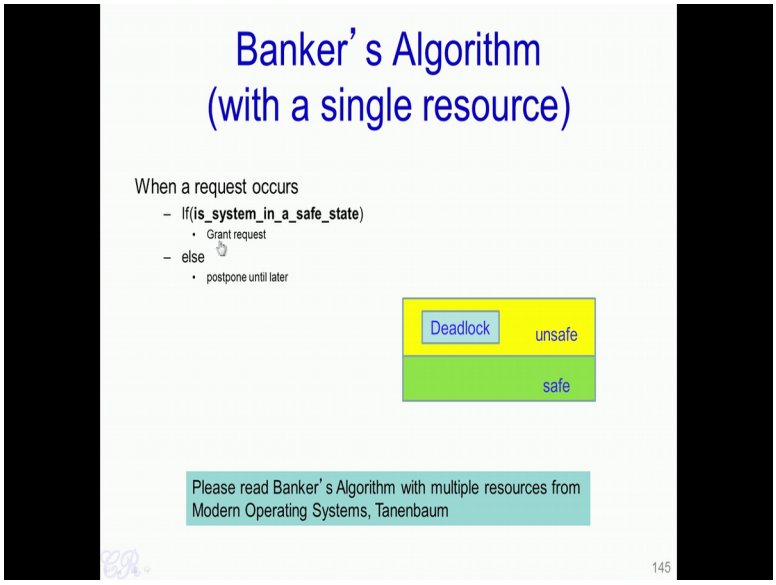
(Refer Slide Time: 13:35)



Now, let us look at another example where the state we start with, is that A has 4 units with it, B has 2 and C has 2, so a total of 8 units are present with A, B and C respectively (refer above slide image). And the number of free units is 2 and the maximum credit limit of each client is as before 9, 4 and 7 respectively. Now, we see that irrespective of how the banker tries to allocate these remaining units to each of its clients, there is no schedule which is feasible that could fulfill these requirements.

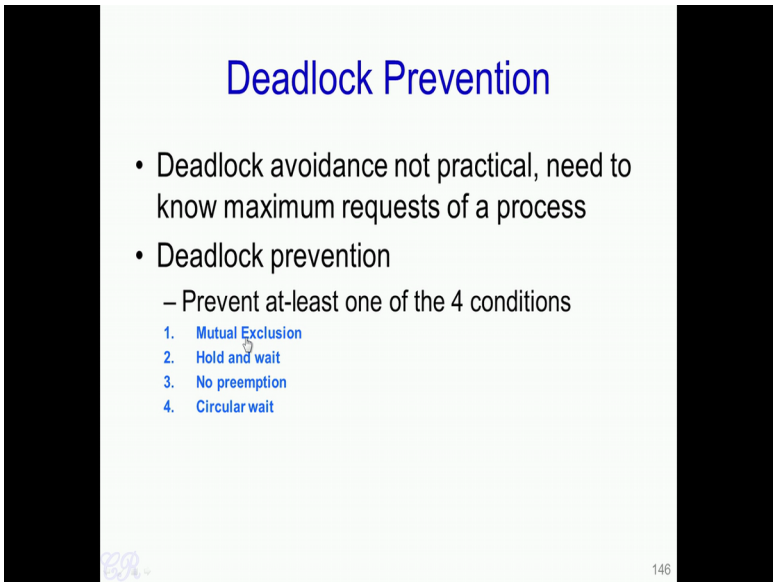
For example suppose B were allocated 2 more units and you see that there are only 2 units free, so B is the only client which can be serviced. Therefore, B gets 4 units while there are 0 free units present (refer table “Allocate 2 units to B”). Now after B completes we have 4 units free (refer table “B completes”), but you see that neither A which requires 5 more units, nor C which requires 5 more units as well can be serviced with 4 units which is present in the free store. Therefore, we see that this state is an Unsafe State because there is no scheduling order which can cater to these requirements (process requirements 9, 4 and 7). So, this is an unsafe state and the banker should ensure that such a state is not reached at any point.

(Refer Slide Time: 15:33)



So a similar mechanism can be also implemented in the operating system, where corresponding to each request the operating system will determine if the system is in a safe state. And if indeed the system is in a safe state indicating that there is a schedule of the resources so that all request can be granted, then the requests are allocated. Otherwise the request is postponed until later.

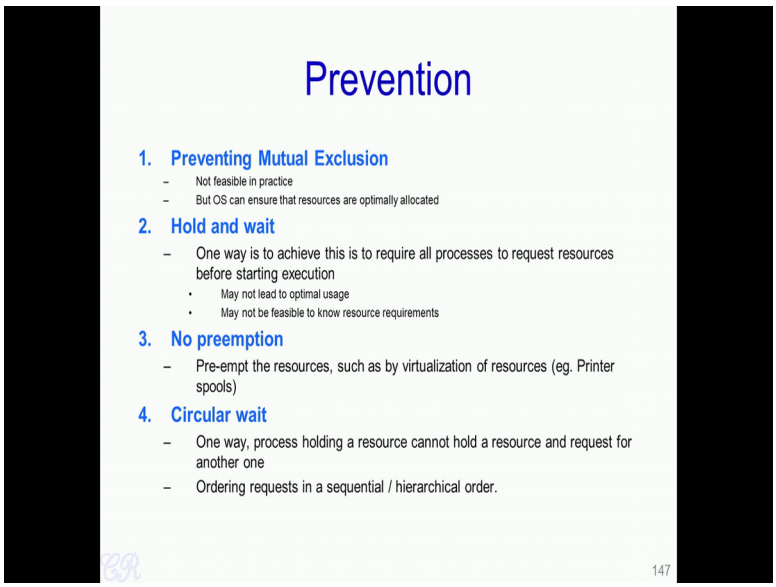
(Refer Slide Time: 16:08)



The third way to prevent deadlocks is what is known as Deadlock Prevention. Essentially we prevent deadlocks by designing systems where one of the 4 conditions is not satisfied (conditions mentioned in above slide image). So, we have seen these 4 conditions and these are the conditions which are essential for a deadlock to occur. By preventing one of these conditions from holding in the system, we can therefore prevent deadlocks. For example, if we design a system where hold and wait condition cannot be satisfied, that is a process cannot hold a resource while waiting for another resource then such a system will not have a deadlock.

So, let us see various ways in which we can actually prevent one of these conditions from happening (refer below slide for all prevention techniques).

(Refer Slide Time: 17:10)



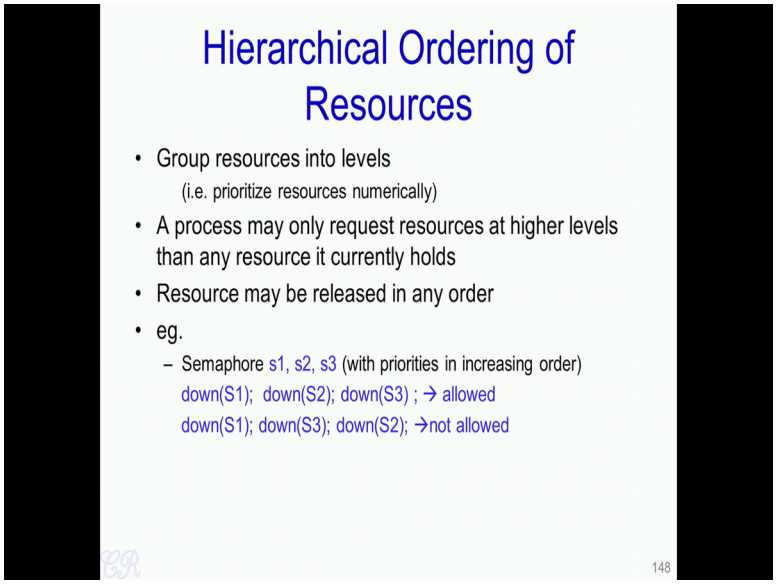
So Preventing Mutual Exclusion, so in practice this is not feasible. We will not be able to always prevent mutual exclusion. For example, take the case of a printer resource so; it always needs to work or print corresponding to a particular process. So, it cannot be simultaneously shared with two processes unless it has a spool present in it. However, this being said, the operating system can ensure that the resources are optimally allocated.

So let us see the Hold and wait. So, one way to achieve this is to require all processes to request their resources before they start executing. So, this obviously is not an optimal usage of the resources and also may not be feasible to know the resource requirements during the start of execution.

Now, let us look at the third way there is no preemption. So, Pre-empt the resources such as by virtualization of resources (example by printer spools). So, in a printer spool we could have several processes sending documents to be printed at exactly the same time. So, all these documents are spooled or buffered in the printer and eventually each document is then printed one after the other.

The final technique we can target is the Circular wait. So one way to prevent this is that process holding a resource cannot hold a resource and request for another one at the same time. Second way to prevent the circular wait is by ordering requests in a particular order - either sequential or hierarchical order.

(Refer Slide Time: 19:20)



So let us look at the hierarchical order of resources. Here resources are grouped into levels that is they are prioritized by some numeric value. A process may only request resources at a higher level than any resource it currently holds, and resources may be

released in any order. For example, let us say we have semaphores s1, s2 and s3 with priorities in increasing order that is s3 is the highest priority, while s1 is the lowest priority (mentioned in above slide image). So, let us say a process makes this particular sequence of semaphore requests S1, S2 and S3. So, we see that an ordering is followed that is S2 is greater than S1, and S3 is greater than S2 (i.e down(S1); down(S2); down(S3)). So, this kind of allocation should be allowed.

However, in this case where we have the first S1 then S3 and then S2 (i.e down(S1); down(S3); down(S2)), so in this particular case the allocation is not allowed because the ordering is not maintained.

Thank you.

Introduction to Operating Systems
Prof. Chester Rebeiro
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Week – 07
Lecture – 33
Threads (Light Weight Processes) Part 1

Consider this particular problem. If it takes one man, 10 days to do a job then how many days will it take 10 men to do the same job? So, we have all done such problems during our school days, and the answer is quite simple to compute. Essentially it would take 10 men 1, day to do the job.

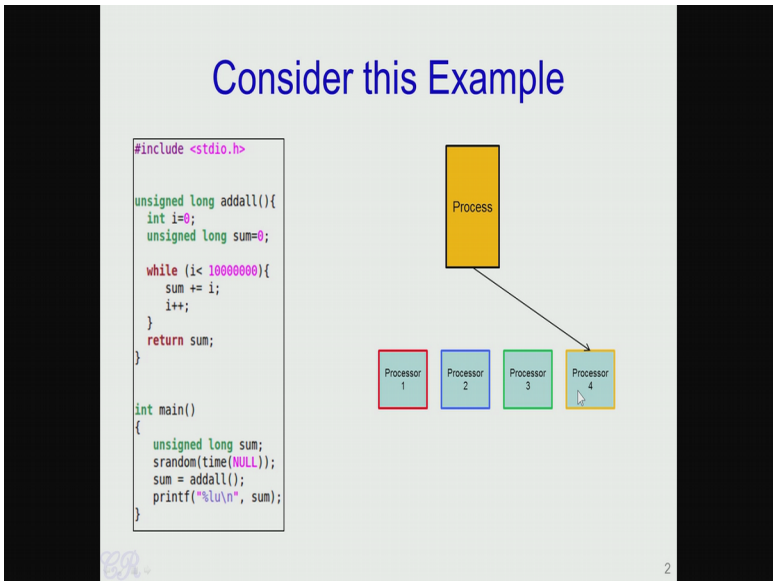
Now the job is done much more quickly because the 10 men are working in parallel. So, each man just needs to do one-tenth of the entire job and the entire job is just completed within a single day. So, this very same concept of parallelization is applied extensively in computer systems. Essentially, parallelization in computer systems are used quite often to improve performance of applications. So, what is done is that if there is a large job to do, we will parallelize it in the system. So that different computing entities, these are computing entities which are very small and do a small portion of the larger job and these computing entities execute in parallel and all together the job will complete much more quickly.

So, these days parallelization is extensively supported by several computer hardware. In fact, there are some hardware that is present which are called GPUs or Graphics processing units which are capable of performing thousands of tasks in parallel. So, each of these tasks are a very small part of a larger job, but the fact that all these tasks are done in parallel will achieve a much more lesser time to complete the larger job.

Now one important construct when doing parallelization is threading or threads, threads are essentially execution entity, very much like processes which we have studied in the previous videos. However, threads are extremely light weight processes, and they are essentially used to make parallelization much more easy.

In this video we will look about threads and essentially provide an introduction to threads. So, the video will cover how threads are created and destroyed, how it differs from processes, and how different operating systems support threads in different ways.

(Refer Slide Time: 03:12)



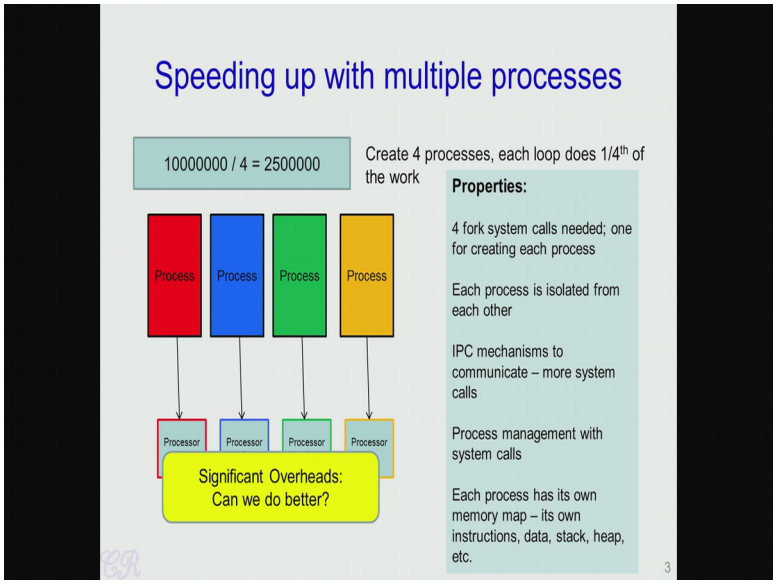
So let us start this particular video with an example. So let us consider this particular program (refer above slide image program), essentially we have a function here called addall() which is invoked from main() and returns unsigned long. So, this function addall(), just sums up or adds up the first 10 million positive integers and the sum is returned over here (inside main() function to variable sum), and printed in the main function. So obviously, this particular program could be written in a much more different and much more easier way, without having to iterate through every number from 0 to 10 million. However, for stay sake of understanding threads, let us go with this particular program.

So, what we have seen in the earlier videos is that after we write this program, we use a compiler such as gcc which creates an executable and then that executable is run to create a process. So, we will eventually have a process which has its own memory space and part of it may be in the RAM (refer above slide image) and now let us see what would happen if this process is executed on a system with say 4 different processors. So, what happens is that the scheduler in the operating system is going to pick up this process and assign it to one of these processors. So, always once assigned this process will be executed in one of these processors, and a rarely this process may migrate to other processors, but let us not get concerned with that.

Now, what we observe here is that even though we have 4 processors present in the

system, this entire program (refer slide time 03:12) which we take considerably the long to execute on a system. Just runs on a single processor leaving the other 3 processors ideal or doing some other tasks. Thus the execution time of this entire program will be quite large because this sum is found sequentially by iterating through all numbers from 1 to 10 million in a sequential manner as done in this particular while loop. So, given that we have hardware like this with 4 processors, this is not a very good way or efficient way to write this particular program. So, what can we do better to make this program parallelizable and make it to execute on all these 4 processors.

(Refer Slide Time: 06:07)



So, what we will do is that we will take 10 million numbers and divide it by 4 to get 4 quarters of 2.5 million each. So, then we would create 4 processes and each process will loop through a quarter of these numbers; that means, the first process would loop through numbers from 0 to 2.5 million and find the sum of all these numbers, process 2 will find the sum of the numbers from 2.5 million to 5 million, a process 3 would find the sum of numbers from 5 million to 7.5 million and process 4 will find the sum of numbers from 7.5 million to 10 million.

So if we are able to create 4 processes in such a way, then when we execute these 4 processes it will be something like this (mentioned in above slide image). So, each of these 4 processes execute on a different processor in the system and an each of these processes just do a smaller job of finding the sum of 2.5 million numbers instead of

finding the sum of the entire range of 10 million numbers.

Further, since each process is scheduled in its own processor therefore, we get parallelization. Thus what one would expect is that the entire program of finding the sum of the 10 million numbers can be completed 4 times faster. So, this seems to solve our problem of parallelization and it will more effectively use the processors present in the system. And some of the properties that we have of this particular approach are as follows. So, in order to create this particular 4 processes (mentioned in slide time: 06:07) we would require 4 fork system calls. So, each of the fork system calls that we invoke would create one of these processes.

Further what we notice another property of this particular model of speeding up or achieving parallelization, is that each of these processes are isolated from each other. That is each process has its own set of instructions, data, heap and the stack and these segments in this process are isolated from the other processes among these 4.

Next due to this isolation we require inter process communication techniques, as we have seen in a previous video in order to communicate with one process with another. So, we have seen that IPCs can be achieved in several ways. So, one way is by having a send and receive system calls and the operating system will help in creating a channel through which IPCs are achievable. The other ways are by shared memories, in which case a shared memory is created in one of these processes (4 processes mentioned in above slide) again by use of system calls and this shared memory can then be used for IPCs between the processes.

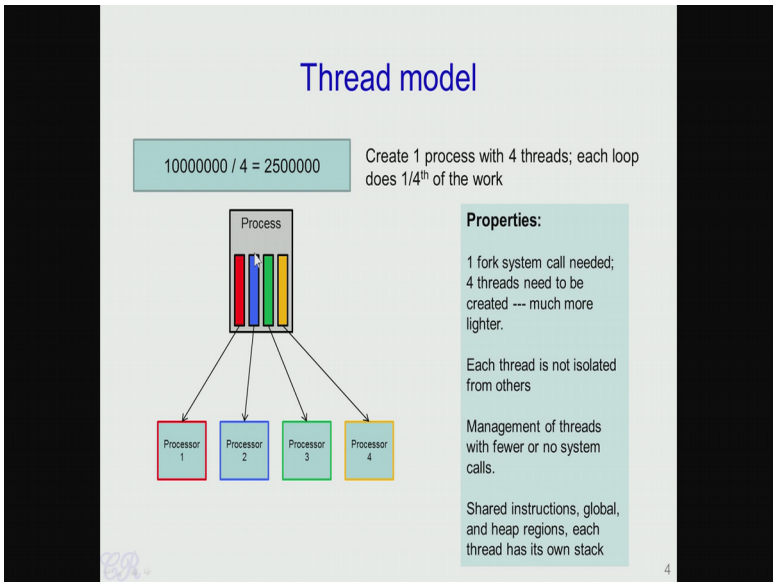
Further, since there are various stages in the process, such as the creation, destroying or file operations and so on. So, all these process management activities are done through system calls. And also the last point as we have mentioned, each process has its own isolated memory map comprising of instructions, data, stack and heap. Now, what we notice among all this from these properties is that there are considerable amount of system calls involved in solving this problem in this particular way that is by having multiple processes to perform parallelization.

And as we have seen in an earlier video these system calls have considerable overheads. So, every time one of these processes invokes a system call it results in the operating system being executed, being triggered due to a software trap and the operating system

then determines what system call has invoked and the corresponding system call handler will be executed. So, as we have seen in an earlier video this entire process could be quite considerable.

An other overhead comes from the fact that a large portion of these 4 processes are similar. For example, all these 4 processes would have the same set of instructions which they operate upon and also there could be the same array which they are accessing, same global data which they are accessing and therefore, what we see is that there are a lot of duplication of instructions and data which is happening due to having multiple processes which execute a job in parallel. So, is there a way we can do better than this? Is there a way we can have a solution, where the amount of overheads present in parallelization can be reduced? So, in order to achieve this, what is used is the concept of Threading.

(Refer Slide Time: 12:02)



So, with threading like before, we divide the entire space of 10 million into 4 parts of 2.5 million each. But instead of creating 4 processes as we have done earlier, we will only create one process and within this process we create execution entity which we call a thread. So, each of these threads (mentioned in above slide, one process with 4 threads) quite like the previous idea where of using parallelized processes, would have a loop which does a quarter of the work that is each of these threads would find the sum of 2.5 million numbers.

So, pictorially this is what it's going to look like (refer slide time 12:02). So, we have 1

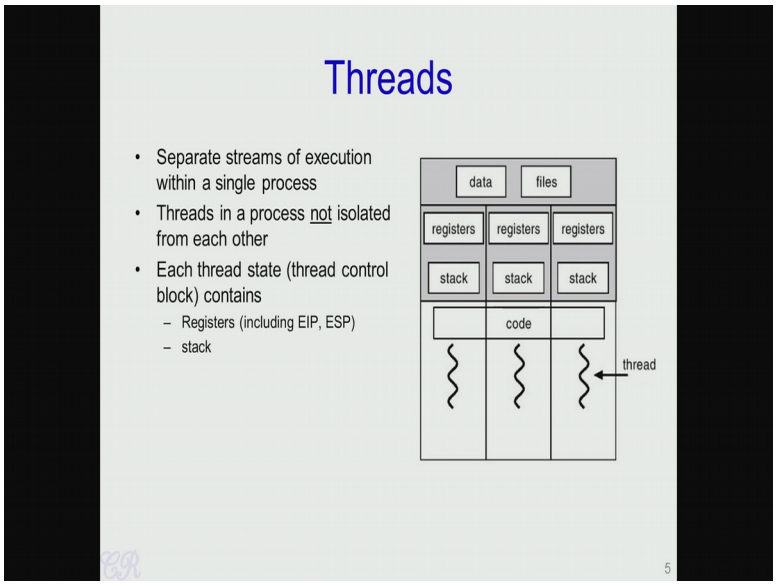
process, so, this 1 process would mean that it has one set of instructions, a global data, heap and other segments, and within this process we have 4 executing context. So, these 4 executing context are the 4 threads and they execute within the particular process. And each of these threads are executed in a different processor or typically executed in a different processor. For example, the red thread goes into processor 1 and its executing here (executes inside processor 1), the blue thread goes into processor 2, green and yellow go into processor 3 and 4 respectively.

Thus we are able to achieve parallelization by using threads. So, this parallelization is quite similar with respect to the process parallelization what we seen in the previous slide. However, it has lower overheads. So, the properties of this particular thread model of a parallelization, is as follows. Since we have created only one process over here, so, what we require is just one fork system call to create this process. Now within this particular process in order to create the 4 threads, we may use a library call such as Pthread create which may or may not require system calls.

Further, a major difference with respect to the process parallelization from the previous slide (refer slide time 06:07) is that these threads are not isolated from each other. So, what this means is that since all these 4 threads execute in the same process space, so these threads could access the processes segment such as the heap segment and the global data. Also it is possible that the instructions or the text segment in this process is also shared among the various threads. Now what is different or what distinguishes the different threads is that each of these threads will have its own stack and having its own stack will allow it to have its own execution context. So, each of these threads will be able to use the stack to create local variables as well as to store information about function calls and so on.

Further management of these threads that is for example, waiting for another thread or exiting a thread, may not require to have system calls. So, what we will see now is these threads in more detail.

(Refer Slide Time: 15:53)



So let us look at it with this particular figure (refer above slide). So, we have this single process over here (complete box mentioned above) and within this particular process we have multiple threads (3 threads as given in above slide). So, these are the execution context of this process. So, these execution contexts all share the same code, and have access to the same global data and heap of the process, and also they will have access to the same files which are opened by the process (given on top of the box). So, what distinguishes one thread from the other is that each thread has its own stack which it uses for local variables and for function calls.

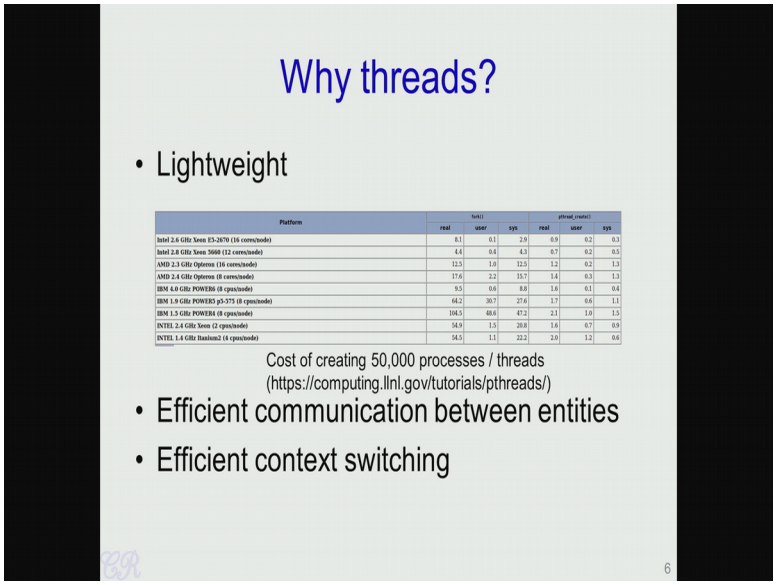
So these threads are as mentioned execution context and each of these threads could be schedule to run on one of the processors present in the system. Or it could be also possible that these threads are scheduled on to the same processor will be at different time instance. Now each thread is executing almost independently of each other therefore, this registers over here (mentioned in above slide in process box) signifies the execution context of a thread. So, as we have seen, each thread has its own execution context.

So the execution context of this thread (first thread in above process box) comprises of a set of registers like the EIP and the ESP register. Similarly the execution context of this thread (second thread) comprises of the stack and registers values. So, these register values would help in the stopping and restarting of threads after or during a context

switch.

Now, what is the major advantage of having threads over processes?

(Refer Slide Time: 17:52)



So this is actually seen from this particular slide (refer above slide); essentially the most important difference comes on the fact that threads are extremely light weight compare to processes. To take an example let us have a look at this particular table (mentioned in above slide image), which shows the time required to create 50,000 processes using the fork system call and it compares this with creating 50,000 threads on exactly the same machines. So, let us take any of these machines as an example. Let us say the first one (first row of table) and what we see and if you look in this particular column (second column), which shows the time taken is 8.1 in order to create 50,000 processes using the fork system call. On the other hand, creating 50,000 threads on exactly the same machine only requires a time of 0.9.

So we see that creating threads are much, much more efficient than creating processes. In a similar way if we actually analyze other operations as well, we would understand that creating and managing threads are much more lighter than managing processes. Another important advantage of threads is that they allow efficient communication between the execution entities. For example, we have seen in processes since each process is isolated from each other therefore, the only way that processes can communicate is by IPCs, and as we have seen IPCs involve quite a bit of system calls. So, for example, in the message

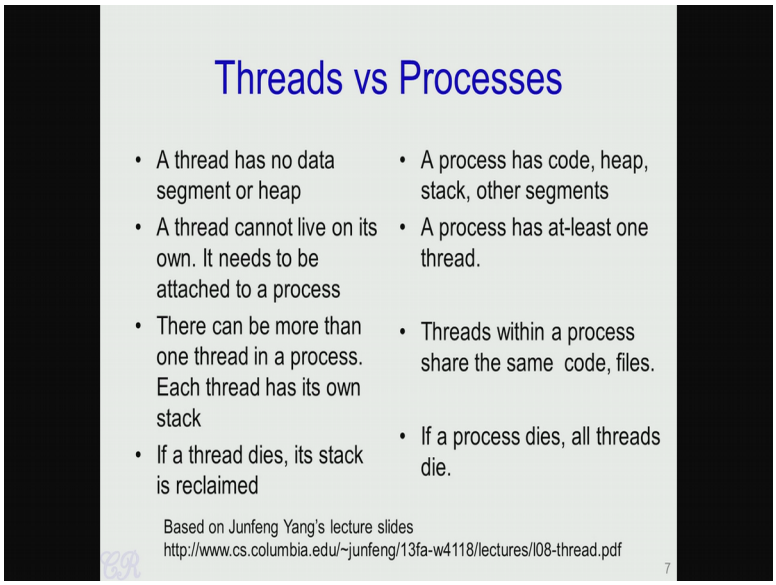
passing IPC where we use send and receive; each send and receive invocation is a system call and has significant overheads.

Similarly for shared memory IPCs, creating and managing the shared memories is again done by system calls, again has overheads. On the other hand communication between two threads in a single process can be done extremely easily, the reason for this is that the global data of the process is shared among the various threads and this means that each thread has access to the global data in the process. Therefore, communication between threads in a single process can be easily achieved via the shared global data or similarly via the shared heap of the process.

Another big advantage of threads is that it results in efficient context switching. So, when we switch from one process to another, the context switching time is considerable because the process state is quite large. Also we require the TLB, we flushed and the page tables for the new process to become active and this new page table would then loaded into the TLB and so on. On the other hand, switching from one thread to another first of all requires a smaller saving of the context, because the context of the thread is much more smaller than that of a process. Therefore, lesser amount of context needs to be saved when switching between threads. Therefore, threads context switching would be faster.

And other reason why context switching is faster in threads is that we are not changing page tables. Essentially since the threads execute from the same process and each process is associated with a set of page directly and page tables. When switching from one thread to another, this page tables and page directories will remain the same. Therefore, there are no overheads of making an other page table the active and therefore, the locality is maintained and cached entries in the TLB will still be valid.

(Refer Slide Time: 22:27)



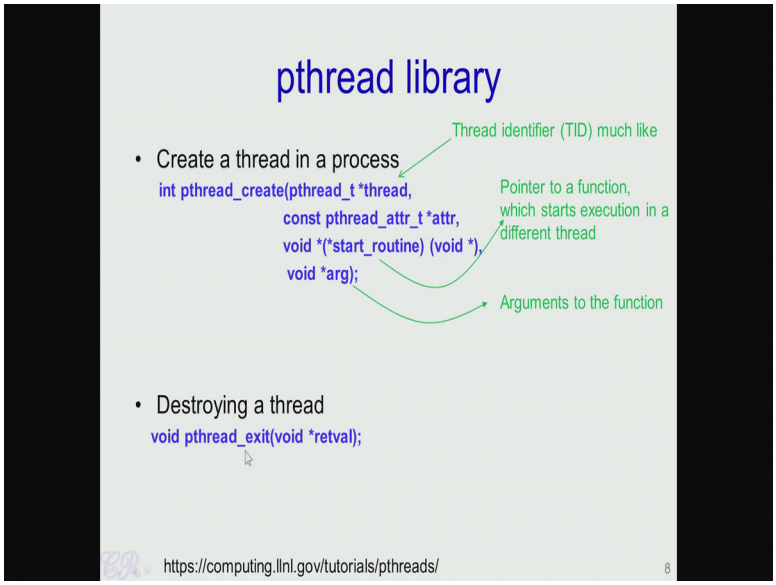
So this particular slide (mentioned in above slide image) re-iterates a lot of differences between processes and threads. So, essentially a process is an isolated execution entity which has its own code, heap, stack and other segments. On the other hand, a thread does not have its own data segment and heap and essentially it adopts or it uses the processes data segment and heap. So, all threads within a particular process would use the same data segment and heap. Another difference is that a process as itself is a complete entity.

So a process you could consider it as having a single thread that is a single execution context and could survive on its own. On the other hand; a thread cannot live on its own. So, it needs to be attached to a process. So, when a process terminates then all the threads that are present in that process will also terminate. So, in other words without a process a thread cannot execute. On the other hand, a process would just have a single thread and that is sufficient for the process to execute.

So, why do we say a single thread? This means any execution context of a process forms the single thread. So, threads within a process share the same code and files as we have seen in the earlier slide (refer slide time 15:53), but each thread has its own stack and this stack is reclaimed, when the thread dies. So, it is possible that when a thread terminates the remaining of the process will continue to execute. For example, if a process has say 10 threads and one of the threads terminates then the remaining nine threads will continue to execute.

On the other hand if a process terminates, then all threads within that process will also die or will also terminate.

(Refer Slide Time: 24:50)



So, how do we create and manage threads? Now there are several libraries to do so, but what we will look in this particular video is a very popular library known as the pthread library. So, we will see some of the very critical or very important functions in the pthread library which are used to create and manage threads. So, let us see how we could create a thread in a process. So, in the pthread library a thread is created by using this function pthread_create(), which takes several arguments. So, it takes like 4 arguments and these 4 arguments are as shown over here (mentioned in above slide image).

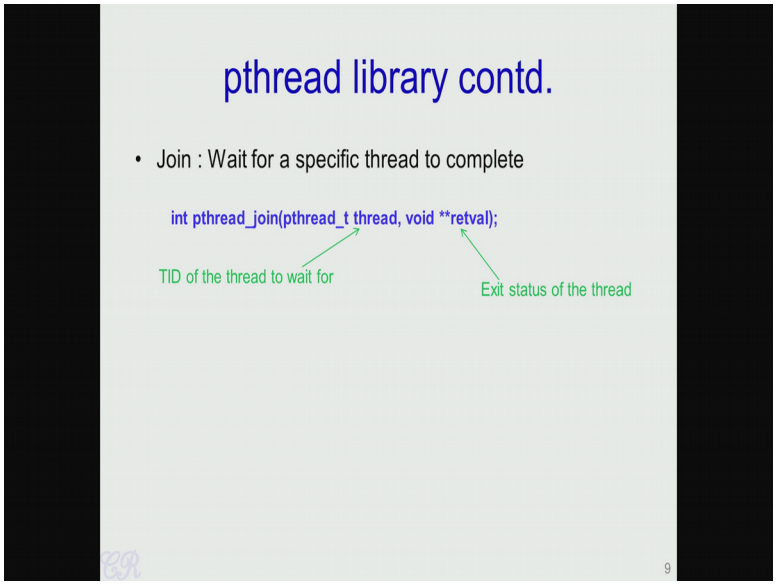
So the 1st argument which is a pointer to pthread_t is the thread identifier or TID (refer above slide). So, this is very similar to the process identifier or PID which we have studied in the process video. The 2nd argument to pthread create is the attribute.

So through this attribute you could specify several properties of the thread that you have been creating. So, what pthread_create() function would do is that when it is invoked, it is going to create a thread context or rather a new thread context in the process and it is going to start executing that new context from this function specified over here. So this 3rd parameter specified as start_routine is a pointer to a function, which begins to execute in the new thread context.

So this function (3rd argument) will start to execute in a different thread. The last argument for pthread_create is arg which is a pointer to the arguments to this particular start routine function (3rd argument). So, in the next couple of slides, we will see examples of how to use pthread create and also examples of the other functions in pthread.

So now, that we have created a pthread and let us say it has done its job. The next thing is to actually destroy a thread and this is done by this function called pthread_exit(). So, pthread_exit() will also pass a pointer to the return value, in order to pass the return status of the thread (mentioned in above slide image). So, this is in many ways similar to the exit system call that we have seen in processes.

(Refer Slide Time: 27:28)

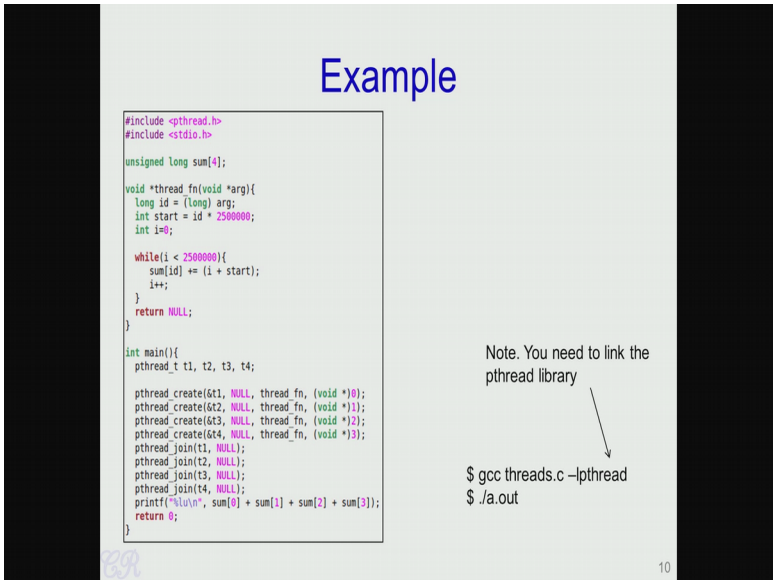


Now, another important pthread library function is this particular function call pthread_join(). So through this, a process or a thread could wait for a specific thread to complete. So, this is much like the wait system call that we have seen with respect to processes. So, in pthread_join() what is specified is the TID of the thread to wait for that is the thread Id of the thread and what is obtained over here is the exit status of the thread (refer above slide for arguments of pthread_join()).

This pthread_join() will block the calling thread, until the thread specified by this TID passed over here (first arguments of function) exits and when the thread exits then pthread_join() will wake up and it will able to read the exit status of the exited thread

specified by the TID. So, given these particular 3 functions pthread_create(), pthread_exit(), and pthread_join(); Let us see how we could use these pthreads in order to parallelize our program of finding the sum of the first 10 million positive integers.

(Refer Slide Time: 28:48)



So, this is the code to do that (mentioned in above slide image). So, what this particular code does is that it creates 4 threads and each thread finds the sum of a different 2.5 million set of numbers. And these 4 threads execute in parallel and therefore, very quickly find the different 2.5 million numbers and then we add the results to get the sum of the first 10 million numbers.

So let us see this example in more detail (refer above slide code). So, first of all we defined 4 variables of type pthread_t. So, these are the t 1, t 2, t 3 and t 4 (mentioned inside main()). So, these are going to be used to create the 4 threads and these would contain the thread ID of the TIDs. Now as we have seen in the earlier slide, creating the thread is done by this function call pthread_create(). So, each of these pthread_create() invocations (mentioned inside main()) would pass the pointer to one of this that is t 1, t 2, t 3 and t 4 which then gets filled with the thread identifier or the TID of the newly created thread.

The second parameter that is the attributes of the thread is specified as NULL while the third parameter is function pointer, which specifies the function where the thread should execute. So, over here the function pointer is a pointer to this function thread_fn, which

is defined over here (above while condition i.e void *thread_fn(void *arg)) and it takes an argument void *arg. So, the last parameter which is passed to pthread_create is the argument which would be passed over here in the arg of this particular function (i.e *thread_fn()).

So, in this way when pthread_create()is invoked, it would create a thread context in the process and this thread context or this execution context will begin to execute from this particular function (i.e void *thread_fn() mentioned in code). Now the argument passed in this particular thread would be as specified in the last argument of pthread_create(). In this way we have invoked 4 pthread_create()and thus creating 4 threads within the single process. All these 4 threads have the same starting function that is thread_fn and the thread IDs will be filled into this first parameter that is t 1, t 2, t 3, and t 4 respectively.

Now each of these 4 threads which will begin executing from this thread function (i.e *thread_fn()) which is like, you could think of like the main of the thread, would see a different value of arg. For example, thread 1 would have a arg value of 0, thread 2 would have a arg value of 1, thread 3 would have an arg value of 2 and thread 4 would have an arg value of 3. So, using this arg value which is then type casted to ID and then it is actually used to determine the start.

So, start (variable defined in above slide) is then use to determine which of the 2.5 million numbers that particular thread should add. For instance if arg in a thread is 0, then the value of ID after typecasting is 0 and the stacked value would also be 0. So, this thread with the ID 0 would add the first 2.5 million numbers that is the number from 0 to 2.5 million.

Now, in addition to this, what is defined is a global array called sum (defined on top in above slide). So, this is an unsigned long array of 4 elements. Now this sum is used to accumulate the sum for each thread. So, sum[0] would contain the sum of the numbers from 0 to 2.5 million which is filled by the thread 0 or thread 1 that is the t1 thread.

Similarly, sum[1] will have the sum of the numbers from 2.5 million to 5 million and this sum[1] is filled by thread t2. In similar way sum[2] and sum[3] would have the corresponding 2.5 million sum and it is filled by t3 and t4. So, while this process is going on in the 4 threads that is this while loop which takes considerably long time, is being executed by the 4 threads independently.

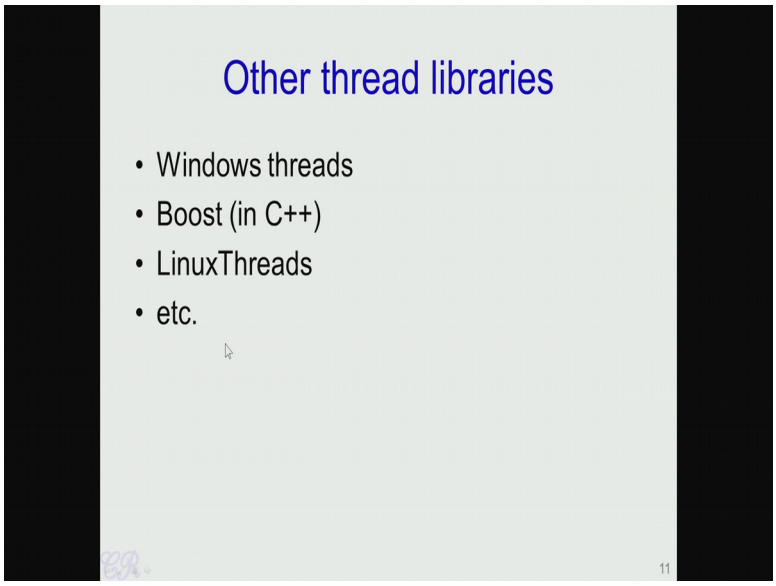
What the main thread does is after creating the 4 threads it will invoke join (mentioned inside main() after pthread_create()). So, it invokes 4 joins corresponding to the 4 threads and what this join call would do is that it is going to block until the corresponding thread specified by the TID over here will exit. Thus when thread t1 exits from this particular function (i.e *thread_fn()), this pthread_join() of t1 will wake up and it will complete executing. So, in this way all 4 join functions on completing will indicate that all the 4 threads have completed their execution.

Then after the 4 threads have exited, the elements of sum which contain the partial sums of the 2.5 million numbers are taken and added to get the final result which is printed on using the printf(). So, this final sum printed is the sum of the first 10 million numbers and therefore, we have seen that the large job of adding numbers from 0 to 10 million is broken down into 4 parts and each part is then computed by a thread to get a partial result which is then added on.

Now in order to execute and run this program we use something like gcc threads.c which is the compilation and we also specify a minus l P thread which is the pthread library, which also needs to be linked into this particular executable (i.e gcc thread.c –lpthread). So, gcc would result in an a.out executable which is then going to be executed.

So, you can actually try out this particular program on a LINUX system and you could try to determine the amount of speed up that you obtain by having different threads to do the job, compared to different processes and also a sequential program where you have only one process executing this job.

(Refer Slide Time: 36:04)



Besides the pthreads there are several other libraries which you could possibly use, to create multi threaded programs. For example, the windows threads library which is a library used for Microsoft windows based applications, boost is a library for C++ then there is also another library known as Linux threads and so on. So, if we actually Google for thread libraries you will find a large list of such libraries which are present.

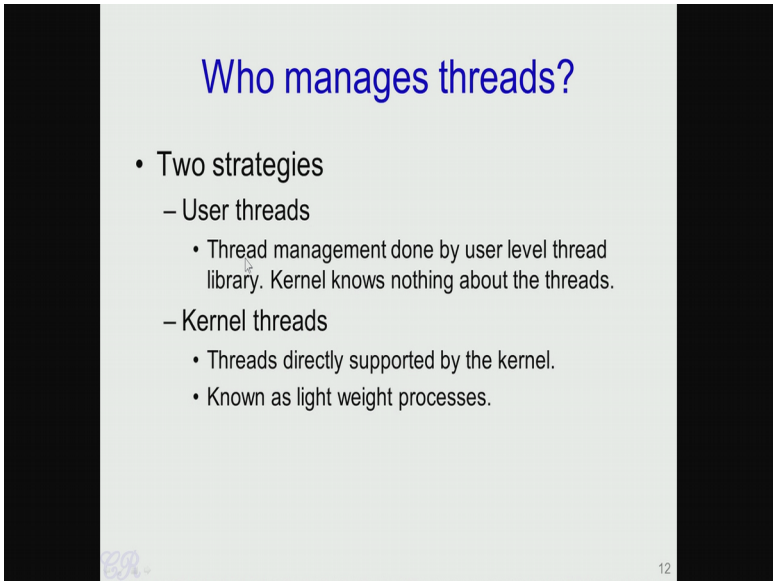
Thank you.

Introduction to Operating Systems
Prof. Chester Rebeiro
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Week - 07
Lecture - 34
Threads (Light Weight Processes) Part 2

So now that we have seen how threads are created and how they can be used to solve large jobs by parallelization. Now, we will look at how threads are managed in systems. Essentially, as we have seen threads are executing context and therefore, we need some entities which manages the thread resources, decides which thread should execute in the CPU what CPU, should be used and so on.

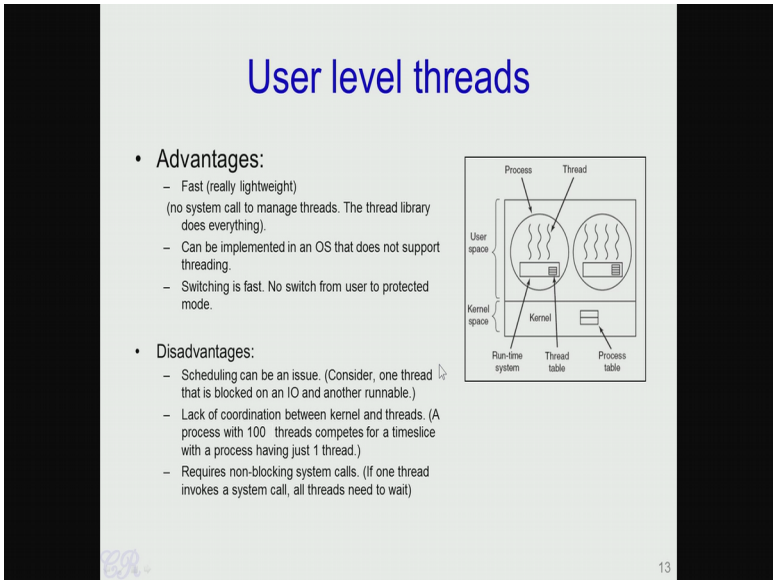
(Refer Slide Time: 00:45)



So in order to do this the two strategies that are available are known as the User threads and Kernel threads. So, User threads are threads where the thread management is done by a user level thread library. So typically in this, the kernel does not know anything about the threads running. In Kernel threads, the threads are directly supported by the kernels and these are sometimes known as light weight processes.

So, we will look at user threads and kernel threads in more detail taking one at a time.

(Refer Slide Time: 01:20)



So let us start with User threads. So in a user thread, as shown over here (mentioned in above slide image) we have the kernel space where the operating system or the kernel executes and we have the user space which has different processes executing. Now, each of these processes would have multiple threads running. So in addition to this we have a run time system over here this rectangle over here (inside process circle), which manages the several threads in this particular process. Also what is required is a thread table which is stored as part of this run time system. So, note that this thread table contains information which is local to only the threads in this particular process.

So for another process, it would have its own run time system and its own thread table. So, notice that the number of entries in the thread table is equal to the number of threads that are executing. Also note that the thread table which is sometimes known as the TCB or the thread control block is different from the process control block which is stored in the kernel space. Essentially besides the fact that the thread table will be have far fewer entries compare to the process control block in the kernel space, it is also in a user space.

So, the advantage of user level threads is that it is extremely fast, and it is really light weight. Essentially this is because there are no system calls required to manage threads.

The run time system which is present over here (mentioned in slide time 1:20) does all the thread management, all the context switching between the threads and so on.

As such the kernel would not have any knowledge that a particular process has multiple threads. So, the second advantage which we will see is that this particular mechanism of supporting threads in a system is useful on operating system that does not support threading. An other important advantage of this user level thread is that switching between threads is extremely fast, and the reason why it is fast is that, there is no switch from user mode to protected mode and back to user mode again.

So on the other hand, switching between threads would just require a switch within the user mode itself from one thread to another. So, the drawbacks are also several. So, one important drawback is that there is a lack of coordination between the operating system kernel and the threads. So, this is because the OS is not aware that a process is multi threaded, and also it has no indication about how many threads are present in a particular process.

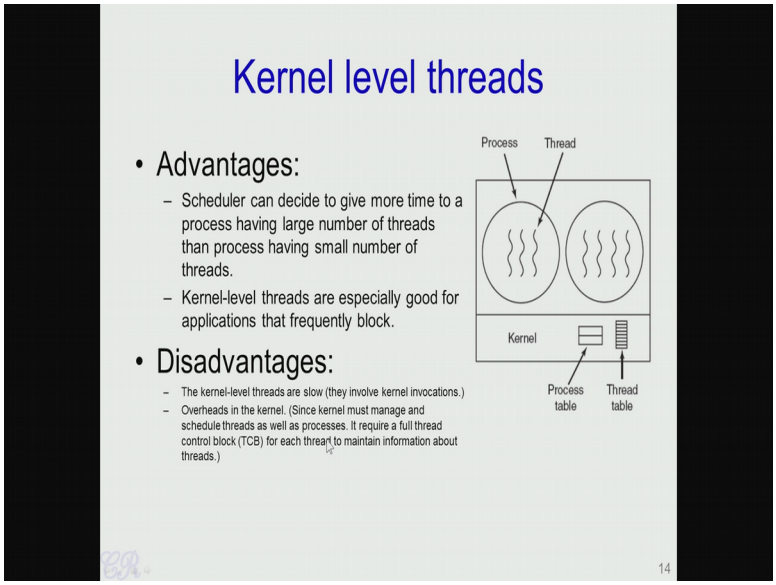
So to take an example of what problems it could cause. So, consider a system which has 2 processes, and one of these processes has 100 threads. Now the kernel does not know about this because the kernel is unaware about the several threads that are present in a process and therefore, when it does context switching it is unaware that one process requires a far more number of threads compare to the other one. So, it would allocate the same time slice interval for the process with 100 threads as well as the process with the single thread.

So, what happens is that, because the scheduler in the kernel is unaware about the number of threads that are executing therefore, scheduling decisions cannot be made to favour processes with the larger number of threads. So, another drawback with respect to the scheduling comes, when the threads are in different states. For example, let us say this particular process has three threads, and one of these threads is in runnable state, while the other two threads are in blocked state. Now the operating system is unaware of this. So, what should the decision be? So, since one of the threads should be runnable it can execute in the processor or should it be considered as a runnable process, or since

there are two threads which are blocked, should this process be considered as a blocked process and not going to execute in the processor.

A Third issue occurs with respective system calls, now suppose system calls are blocking in the sense that when one of these threads invoke a system call, then all other threads will need to wait until that first thread completes its system call invocation. Thus in order to support this user level thread model, what is required is that the OS should preferably be supporting non blocking system calls.

(Refer Slide Time: 06:35)



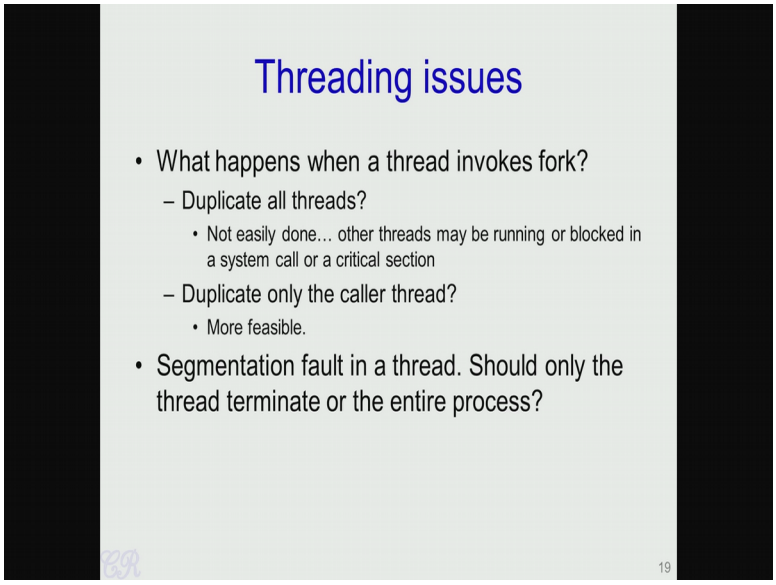
So now let us look at the kernel level threads. So, as before the process which runs in user space could have multiple threads with each thread being a separate or a independent execution unit and the management of these thread resources is done in the kernel space. So, along with the process control block, that is shown here as a process table, the kernel also maintains a thread control block in the kernel space (mentioned in above slide). So, this is quiet unlike the user level threads, where the thread table is maintained in user space, here the thread table or the TCB thread control block is maintained in the kernel space. And therefore, the kernel is aware of the number of threads that a process executes, and therefore, could make a lot of decisions based on this fact.

For example; the scheduler could make more smart decisions about how much time slices should be allocated to a process, depending on the number of threads the process is running. So, for example, the scheduler could decide to give threads with a larger number of processes more amount of time to execute. An other advantage is that since threads are managed by the kernel, so, the blocking on system calls is not required. Essentially when a thread executes a system call the other threads in thar particular process can continue its execution.

It does not have to block until this thread (thread mentioned in above slide image) which invoke the system call has completed its invocation. The drawback of this particular kernel level thread model is that it is slow. Essentially managing the thread would in kernel invocations, and therefore; since this kernel invocations involve system calls therefore, it is considerably slow. Also there are overheads with respect to the operating system and this occurs because the OS, the kernel needs to manage the scheduling threads as well as the processes.

So, this means that in addition to the metadata present in the kernel about the process, more metadata needs to be present for each thread that is executing in the process and therefore, the overheads in the kernel could be significant. So, when we actually design an operating system or for that matter an entire system which supports threads, there are several aspects which need to be taken care of which may lead to lot of complexities. So this particular slide (refer below slide image) highlights some of those issues.

(Refer Slide Time: 09:15)

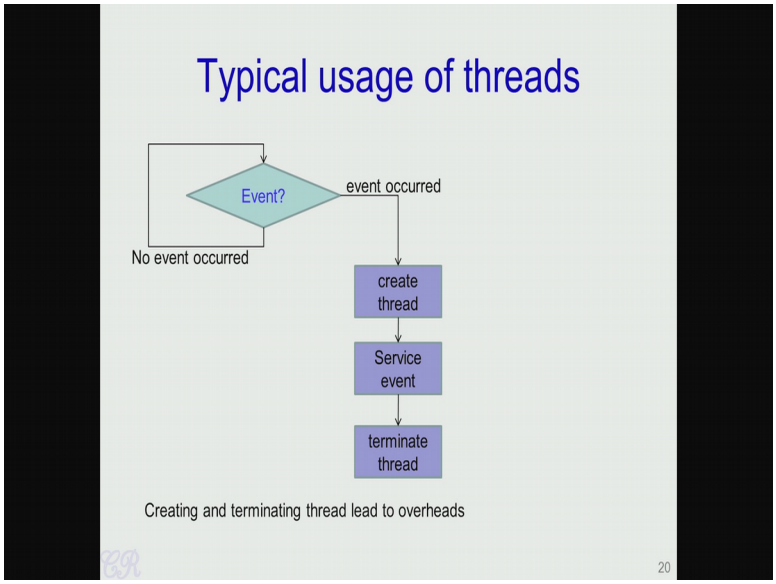


For example what should the system do when a thread invokes a fork? So, as we know a fork is a system call which causes the operating system to execute and it would result in duplication of that process. So, a new process would be created which is an exact copy of the invoking process. Now what would happen if a thread invokes the system call fork? So, there are several options that one could think off. So, what should the OS do? Should all the threads that are executing in the process should be duplicated, in other words should a new process be created, which also has the same number of threads executing in the same stages.

So, this is easier said than done, essentially it could also create a lot of synchronization issues. For example, consider the case that we have a process with two threads, one thread is executing a critical operation say in a critical section, which is accessing a critical resource while the other thread invokes the fork system call. So, what would happen if the OS duplicates the entire process along with all its threads? So, as we have mentioned the second thread in the new process would also be in the critical section and as we have seen in earlier videos this could be catastrophic, essentially it could change the output of the program. An other approach that one could follow while designing or managing this particular aspect is where we duplicate the caller thread?

So, even though the process may have 10 different threads, one of these threads invoke the fork system call, the new process created will only duplicate that thread. So, the new process created will not have the remaining 9 threads, but will only have one of these threads. So, another thing to think about is what should happen when there is a segmentation fault in a thread. So, should the operating system terminate the just the single thread, or should the entire process be terminated? Now making choices for these is not easy and operating system designers would need to make critical choices about how to manage these aspects while designing the operating system.

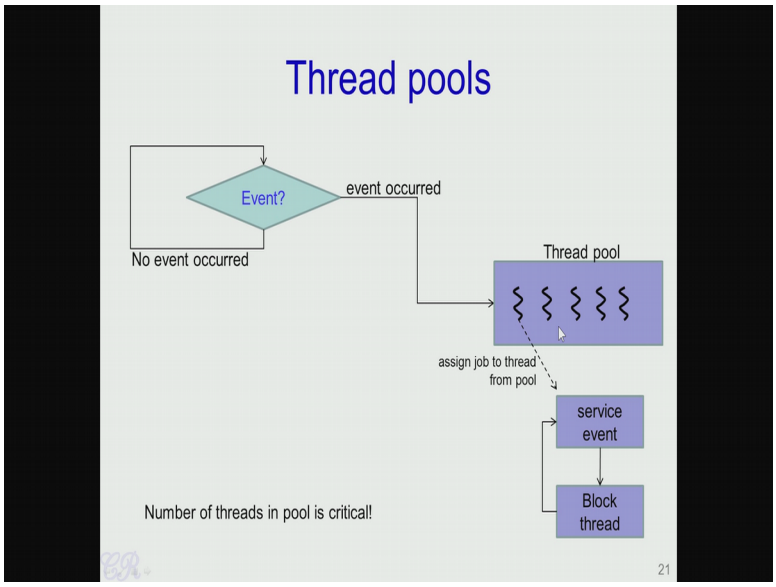
(Refer Slide Time: 11:46)



So let us see one typical application of the use of threads. So, you could take for example, a network card where packets keep coming in through the network and these packets have to be serviced let us say through threads. So let us say we have a loop over here (mentioned in above slide), which keeps waiting for an event to occur for example, the packet on the network and when that when an event occurs, it spawns or it creates a thread which services that event and then terminates. So, during the process of servicing of that event, if another event occurs, then a new thread is created. In this way we could have several events which are serviced simultaneously.

So this approach is scalable because it could service multiple events simultaneously. So, the drawback of this particular model is the overheads. Essentially creating and term terminating threads though have less overheads, is still an overhead which could reduce performance and these could affect the entire system. So, what applications typically do is use the technique known as Thread pools.

(Refer Slide Time: 13:10)



In this particular technique, what the application does on creation is that it is going to create a pool of threads. For example, it could create say 50 or 100 different threads and these threads are would typically be in a blocked state.

Now, there would be a main loop which keeps waiting for an event to occur (mentioned in above slide image) and when an event occurs one of these threads which in the blocked state is woken up, that thread would then service the event and go back to the blocked state. So, in this way if there are 50 threads in this pool which are already created, there are 50 events which could be serviced simultaneously without any overheads. If the 51st event occurs while all the 50 threads in the thread pool are busy servicing the event, then the 51st event would need to wait. So, this way we see that we have eliminated the overheads of creating and destroying threads whenever an event occurs.

So now, the only requirement is to pick out a thread from the thread pool which would then service the event. Now an other important aspect with thread pools is the number of threads in the pool. So, this is critical for every application and is going to be very application dependent. For example, if you have a thread pool with very few numbers of threads may be say 4 or 5, then it could service only 4 or 5 events simultaneously.

If more events occur then the events would have to be queued until the threads have completed their servicing and therefore, your performance is affected. On the other hand if we have a large number of threads in the thread pool for example, 1000 threads while the events do not occur so often therefore, a large number of threads are simply wasting resources sitting idle in the thread pool. Therefore, the number of threads in the thread pool is critical choice that an application designer will have to make.

So, with this we have given a brief introduction to threads, we had seen the difference between threads and processes and how threads could be actually used to reduce the execution time and improve performance of applications, and we have also seen a brief introduction of how threads are managed in operating systems and the various usage of threads.

Thank you.