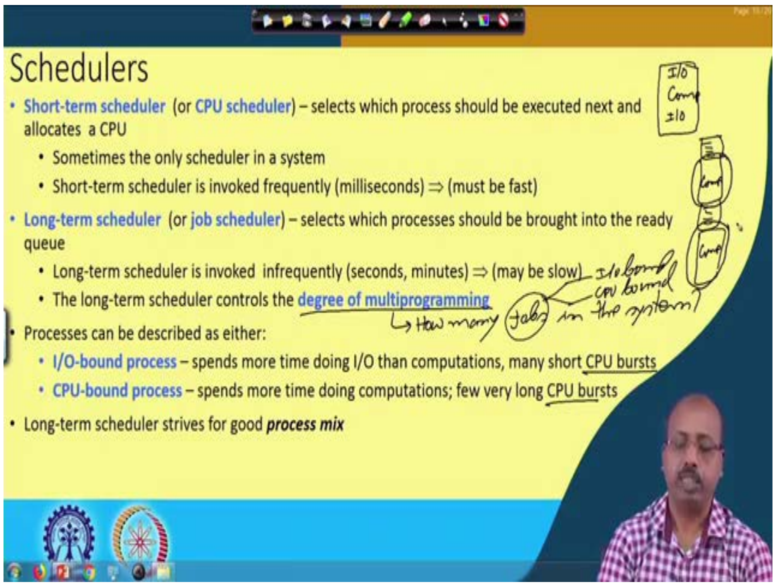
Operating System Fundamentals

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

Lecture - 14
Processes (Contd.)

(Refer Slide Time: 00:38)

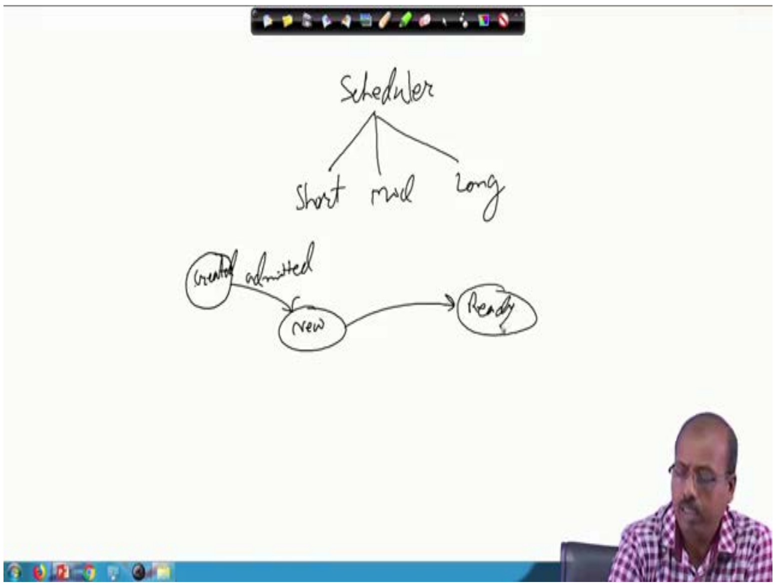


Schedulers

- Short-term scheduler (or CPU scheduler) – selects which process should be executed next and allocates a CPU
  - Sometimes the only scheduler in a system
  - Short-term scheduler is invoked frequently (milliseconds) ⇒ (must be fast)
- Long-term scheduler (or job scheduler) – selects which processes should be brought into the ready queue
  - Long-term scheduler is invoked infrequently (seconds, minutes) ⇒ (may be slow) I/O bound CPU bound
  - The long-term scheduler controls the degree of multiprogramming ↳ How many jobs in the system
- Processes can be described as either:
  - I/O-bound process – spends more time doing I/O than computations, many short CPU bursts
  - CPU-bound process – spends more time doing computations; few very long CPU bursts
- Long-term scheduler strives for good ***process mix***

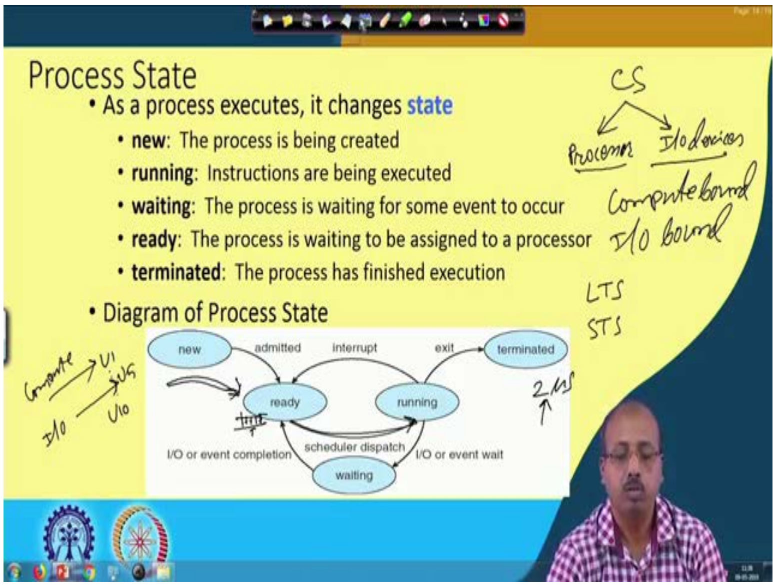
So, this CPU scheduler that we are talking about in the last class, so they have got differ, they can be divided into at least two different classes, one is short-term scheduler or CPU scheduler, and another is long-term scheduler or job scheduler. And also there is another scheduler which is known as midterm scheduler.

(Refer Slide Time: 00:50)



So, basically, so if you look into this classification then this scheduler, I can divide into three classes, short-term, midterm and long-term. Now, if you remember that process flow diagram state process state transition diagram, so we have said that, so the processes when they are residing in the disk, so this is that created state, now created to the new state that is by this admitted. Now, from this new, it was going to the ready.

(Refer Slide Time: 01:43)



Process State

- As a process executes, it changes **state**
  - **new**: The process is being created
  - **running**: Instructions are being executed
  - **waiting**: The process is waiting for some event to occur
  - **ready**: The process is waiting to be assigned to a processor
  - **terminated**: The process has finished execution
- Diagram of Process State

If we look back into this state transition diagram, for this from the new state, so it was coming to a ready state; and from the ready state it was going to the running state. Now, what happens is that there may be say 10 different users in the system ok. So, this user one to user 10, there may be 10 different users in the system. So, they have created some processes. Now, when this processes are created, so we may be choosy about which processes do we select to be admitted into the system.

Why do you do this? Because like in a computer system. So, there are two types of elements that we have in the computing system. One is the processor, so definitely which will be doing the jobs, and other is the I O devices. Now so both of them are resources as far as the computer system is concerned this processor as well as I O devices both of them are resources. So, what we would like to do have in the system is that both the processor and I O devices they are utilized 100 percent times or at least to a very high percentage of time.

Now, depending upon the type of process that we have in the system, so they that one of them maybe achievable where as the other not. For example, if we look into the database type of operations, then the competition that is involved is very less maybe in a database operations. So we have got a query to identify the people whose salary is more than say rupees 30,000 per month. So, that way as far as the operation is concerned, it has to do lot of disk axis and it has to see like if the salary is more than 30,000 or not. At the same time, it does not have lot of competition there.

On the other hand, suppose we have got this satellite data, and from there we are trying to predict the weather for the coming days, so that way there will be lot of computation. So, we have got only a few images from the satellite, but after that the competition that is involved is a very high. So, these two types of application some of them are compute bound. For example, that weather forecasting, so that is a compute bound job, and we have got this database operation so which are I O bound jobs.

Now, if in my system all the jobs are say compute bound job, then what will happen is that this I O devices they will remain ideal. On the other hand, if there are all the jobs are of type I O jobs then the processor will be ideal most of the time. So, we want to make a balance of these two. Now, how to make a balance? So, this is a controlled actually at this point. The thing that I was talking about maybe the type of jobs coming from user 1, so they are say compute bound jobs. Similarly, from coming from user 5, so they are I O bound jobs. So, that way at any point of time I can take a decision like, I can look into the ready queue and try to see what are the different types of jobs that we have in my system, what is the current load on to the processor, and the I O devices in terms of their utilization, waiting queue and all.

And then I can take a decision whether I should take admit the compute bound jobs or the I O bound jobs, so that can be the that may be the policy. So, this is done by the long-term scheduler. So, long-term scheduler, it they tries to find out the characteristic of the jobs that are coming for the system from different users, and accordingly admit some of the jobs whereas making others to wait for some time to be admitted. So, this is the long-term scheduler.

On the other hand, if I look into the this part of the scheduling that is have got a number of jobs waiting in the ready queue. So, in this ready queue, I have got a number of jobs waiting. Now, from here I have to select some job that will be going to the running state. Now, how do you do this thing? So, for doing this again I have to look into the priorities of these jobs and many other parameters.

Now, whatever we do, but it has to be done very fast because whatever time I am spending here, so that is basically are overhead for the system. And since maybe I may decide that any job that is going to the running state, so it will be executing for say two microsecond and then the CPU will be given to the next job. So, every two microsecond I have to take a decision like which job will be going to the running state, so that decision must be made very very fast. So, it cannot be of the order of microseconds again.

So how do we do this thing? So this particular scheduler that is running here which is known as short-term scheduler or CPU scheduler that has to be very fast. Whereas this long-term scheduler that we are talking about, so this long-term scheduler it may take slightly more time, but it should give me a good mix of job. On the other hand, the short-term scheduler that we are talking about, so that should be very fast and it may not be doing that much of selection between the jobs.

So we will see that the scheduling policies of this short-term scheduler. So they are very simple whereas this long-term scheduler they can do some statistical analysis of the jobs that of given by different users, and accordingly try to do something to maintain the balance between I O bound and CPU bound jobs. So, we will see that in more detail later.

So, coming back to the point that you are discussing, so this scheduling queues. So, we have got the short-term scheduler that the CPU scheduler, it will select which process should be executed next and allocates the CPU. Sometimes the only scheduler in the system, so many time many systems we do not have the long-term scheduler also because we do not have that much of load in the system. So, that much of variety of users are not there in the system.

So, that way for example, say in a laboratory environment maybe the jobs are submitted by students only, and all jobs are of similar type may be they are submitting some assignments and all. So, we do not have variation between the jobs. So, that way putting a long-term scheduler they are does not have any accepting wasting some, sometime in doing the scheduling. So, that way we may not have a long-term scheduler available in the system at all. So, this is short-term scheduler maybe the only scheduler.

So, this is invoked very frequently in the range of milliseconds or microseconds, and naturally it must be very fast ok, because that is an overhead that is coming. And this long-term scheduler, so it will select which processor should be brought into the ready queue. So, it is invoked in frequently in terms of seconds and minutes and naturally it may be slow also. So, it controls that degree of multi programming.

So, this term degree of multiprogramming, so this is actually tells like we have got how many jobs in the system; so how many jobs in the system so, this is actually the degree of multiprogramming. So, you can also refine this question to a bit like how many instead of jobs we may ask for classification that I O bound and CPU bound jobs.

Now, if this degree of multiprogramming is low that you have got less number of jobs in the system, naturally the system does not have much thing to do. So, system will remain ideal for most of the time. So, that way the system throughput will be low. Now, if you find the system throughput is low, so one possible reason is that degree of multiprogramming is low. So, you try to induct more processes into the system ok. So, more processor put into the ready queue. So, that way degree of multiprogramming goes up and the system is expected to be utilized to more.

But there are some situations that will see later where we will find that the degree of multiprogramming is very high, but system throughput is very low. So, that indicates some erroneous difficult condition that has come in the system, some undesirable situation that has come in the system, and there we need to reduce the degree of multiprogramming. So, when the system throughput is pretty low, but degree of multiprogramming is high, so that indicate some malfunctioning of the system. So, we need to reduce the degree of multiprogramming in that case. So, this is the two-sided (Refer Time: 10:35) you can say, so you can increase it or you have to decrease it at sometime.

So, processes can be described to be I O bound process or CPU bound process as we discussed. So, this I O bound process is this spends more time doing I O than computations. So, there will be the many short CPU bursts, and CPU bound processes they spend more time doing competition so very long CPU bursts. So, understand like what is this CPU bursts and this I O bursts, so it is like this.

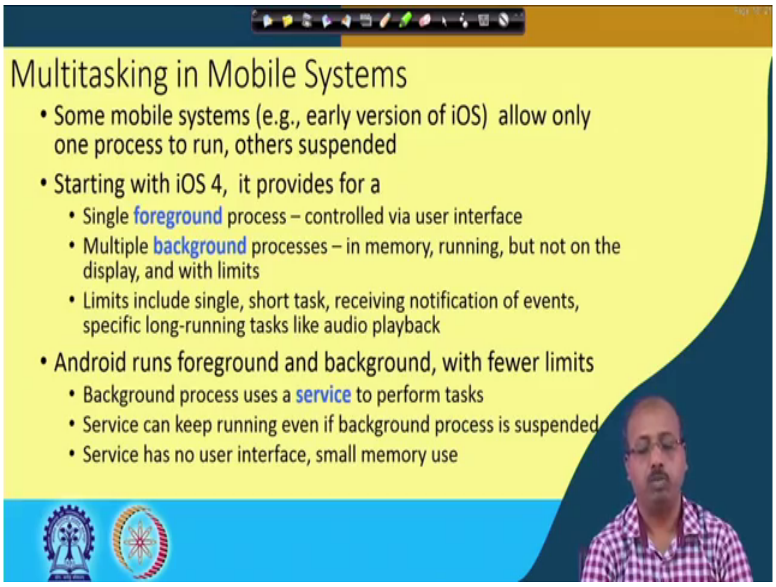
Suppose, we are writing a program, so any program that is right. So, initially it will be doing some I O operation. For example, taking back; like taking back that quadratic equation. So, initially that program should read the values of a, b and c, after that it will do some competition, then it will be compute it, it will be getting the roots and printing it. So, initially there is some I O, then there is some computation, and finally, there is some IO. So, that is the structure of our example that we to finding roots of a quadratic equation.

So, in general I can say a program, so it will be doing I O for some time, then it will be doing CPU for operational the computation for some time, then it will again do I O for some time, then again it will do competition for sometime ok. So, these individual durations of this I O and CPU that are coming, so they are called the bursts. So, this is called is CPU bursts, similarly this is another CPU bursts so like that.

So, if a job is I O bound job, then what will happen is that, this I O bursts it will be longer, but CPU bursts will be smaller. So, the very few CPU operation would be required, but most of the time it will be doing I O operation. On the other hand, if a process is compute bound, then this I O bursts will be small and compute this CPU bursts will be large. So, if you whenever the CPU is given to the processor to the process; when the CPU is given to the process, if you start a timer to see how much CPU time it uses before going for the I O bursts, so that may be a measure of the CPU bursts.

So, for see compute bound job, so you will find that this CPU bursts are large compared to I O bound jobs. So, as I said that a good job mix, it should have both CPU bound job and compute I O bound job in the system. So, this long-term scheduler, so it will try to get good process mix, so that is required.

(Refer Slide Time: 13:18)



Multitasking in Mobile Systems

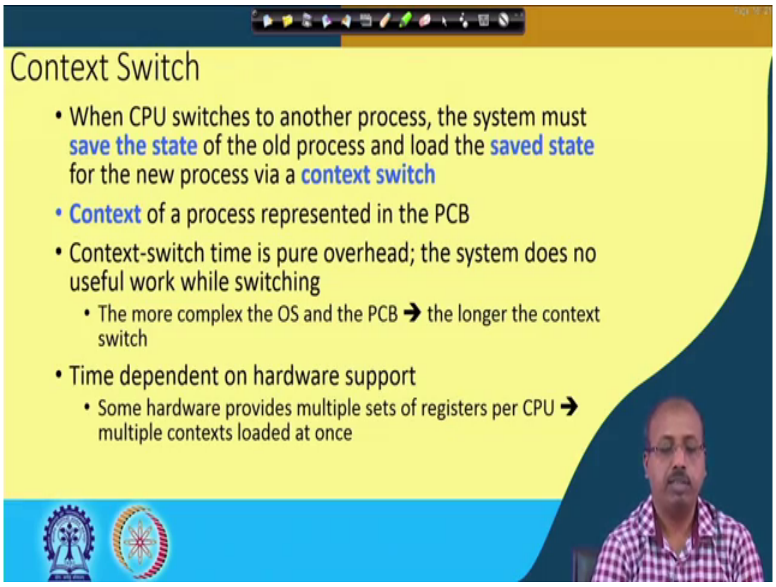
- Some mobile systems (e.g., early version of iOS) allow only one process to run, others suspended
- Starting with iOS 4, it provides for a
  - Single foreground process – controlled via user interface
  - Multiple background processes – in memory, running, but not on the display, and with limits
  - Limits include single, short task, receiving notification of events, specific long-running tasks like audio playback
- Android runs foreground and background, with fewer limits
  - Background process uses a service to perform tasks
  - Service can keep running even if background process is suspended
  - Service has no user interface, small memory use

So, we will see how this is done by different term schedulers and all. Now, in mobile system, so there are concept of multitasking. So, some mobile systems for example, the early versions of iOS allow only one process to run and others are suspended. So, this is very simple system and only one process will be running. So, multitasking is not there. Starting with iOS 4, it provides for a single foreground process controlled via user interface, and multiple background processes in memory running, but not on the display, and with limits. So, there may be one process which is the user has initiated, so that is the foreground process. And there are a number of background processes that are there in memory, they are running, but at with a lower priority.

It limits the in a limits include single short task and receiving notification of events, specific long running tasks like audio playbacks. So, this is the different types of limits that we have in the system.

On the, so if you look into android, so it runs foreground and background with fewer limits. So, background process uses a service to perform the tasks. And service can keep running even if background process is suspended, and services have user interfaces small memory usage, so that is the structure of this android. So, again it has got the foreground and background job. So, again the foreground jobs will be always running and the background jobs are there, but depending upon it is a conceptualized in terms of services.

(Refer Slide Time: 14:53)



Context Switch

- When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process via a context switch
- Context of a process represented in the PCB
- Context-switch time is pure overhead; the system does no useful work while switching
  - The more complex the OS and the PCB → the longer the context switch
- Time dependent on hardware support
  - Some hardware provides multiple sets of registers per CPU → multiple contexts loaded at once

Now, the context switch that I was talking about, so formally coming to the definition of it when CPU switches to another process, the system must save the state of the old process and load the saved state for the new process via a context switch. So, this is the definition of context switch. As I was telling that maybe previously process one was running.

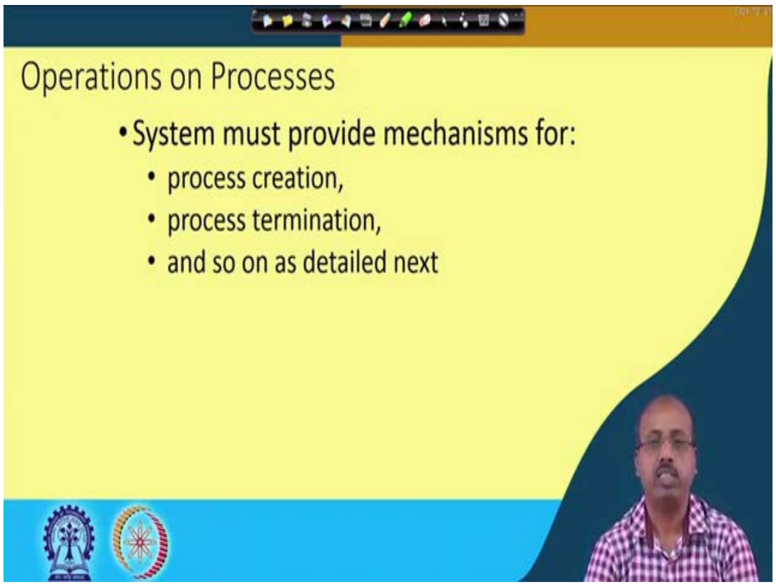
So, now, we want to suspend process 1 and start process 2 ok. So, we need to keep enough information of process 1 in the PCB, so that later on we can start process 1 restart process 1 from that point onwards, so that has to be done. So, we have to save the state of the old process into the PCB. And later on when we want to restart, so we have to restore the context from the PCB to the for the process. So, this is the saved state of the new process has to be copied, and we must save the state of the old process. So, this is known as context switch.

So, context of a process represented in the PCB. So, as I told previously that the process control block is the place where you should keep all these contexts related information. Context-switching time is pure overhead; and system does no useful work while switching. So, naturally we have to minimize the context-switching time. So, this is just the system is doing some copy and restore that is not coming into the execution of the process, so that is basically an overhead.

So, more complex operating system and PCB the longer will be the context-switching time, because you need to save and restore more information. So, context-switching time will go on. Then there are time dependent on hardware support, some hardware provides multiple sets of registers per CPU. So, for example, if you looking to say 8051 or ARM processors etcetera, you will find that there are multiple CPU register sets.

So that if you want to go from one process to another process may be one processes context is stored in one set of CPU registers, another process is store in another set of CPU registers. So, switching between them is nothing but switching between the register banks, so that way the context-switching becomes first. So, many of the processes that are designed now they provide facility for in the hardware for doing this context-switching fast, and that way it makes the whole operation whole context-switching to be done quite fast. And this is a very important issue that has to be taken care of by the architecture designers and OS designers.

(Refer Slide Time: 17:32)

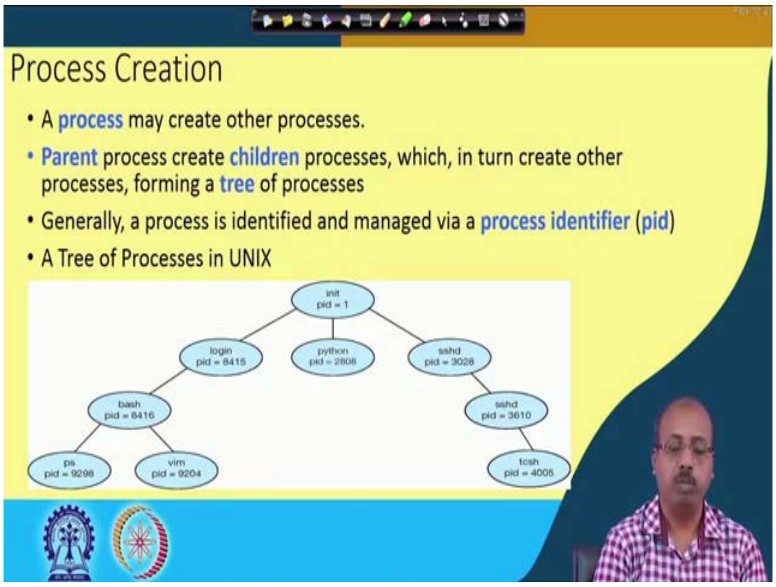


Operations on Processes

- System must provide mechanisms for:
  - process creation,
  - process termination,
  - and so on as detailed next

Next coming to the operations on processes. So system must provide mechanism for process creation, process termination and similarly many other things that will see slowly. So, process creation means I should be able to create a new process, where I will try to do something new, and then a process has to be terminated, so maybe it is voluntary termination or involuntary termination. So, voluntary termination is process ends telling that I am over. And sometimes the processes they misbehave and the system administrator has to intervene and terminate the process. So, this process termination has to be there. So, both of them are required.

(Refer Slide Time: 18:16)



Process Creation

- A process may create other processes.
- Parent process create children processes, which, in turn create other processes, forming a tree of processes
- Generally, a process is identified and managed via a process identifier (pid)
- A Tree of Processes in UNIX

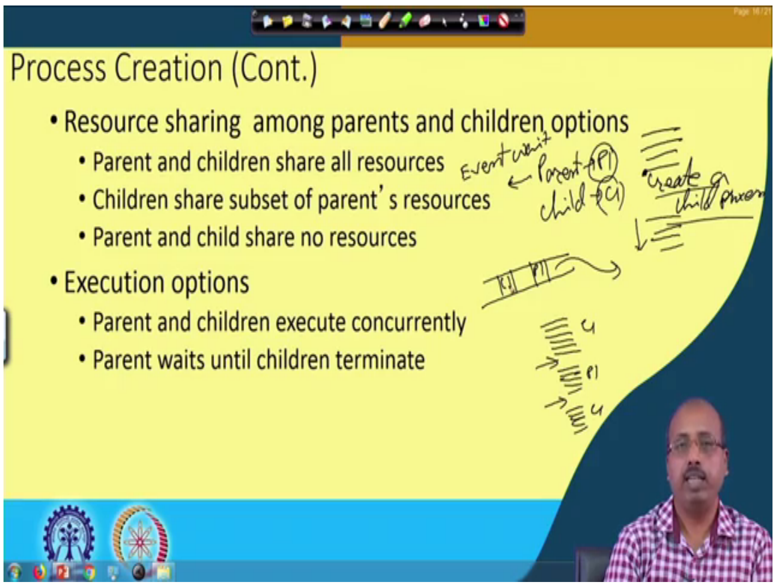
Now, how do we create a process? Now, one process may create other processes, and there is a parent child relationship between the processes. So, parent process creates children processes, and which, in turn create other processes, forming a tree of the processes. Like, so this is the in it whose pid is equal to 1, and then this for examples in an UNIX operating system with this init is the first process that is created its pid is equal to 1.

Now, this init processes, it can create a number of processes children processor, one of them may be a login process that prompts the user to enter the login data, then there may be another process that takes care of the python programs python scripts. Then there may be a shell script sshd so, that is there.

Similarly, from login it can create another process called b shell or bash. Similarly, in the shell, the user might have given some comment to know the process status or these some editor ok; so, this is on vim editor. So, this way you can have a number of interfere number of children created from a parent process.

So, generally a process is identified and managed via a process identifier. So, every process that is created, so it has got an identifier and this identifier value, since one is reserved for init process so it starts from two onwards and it goes on implementing one by one as newer processes are created. And when it comes through some maximum value, then it actually rolls back to the value 2, so expecting that by that time the process will be over. Of course, it should be a check that the pid is not duplicated. So, this is a typical tree that we have for processes in UNIX operating system.

(Refer Slide Time: 20:11)



Process Creation (Cont.)

- Resource sharing among parents and children options
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources
- Execution options
  - Parent and children execute concurrently
  - Parent waits until children terminate

Now, resource sharing among parents and children, the options may be like this that parent and children share all resources. Children shared a subset of parents resources and parent and child share no resources. So, these are the three alternatives like maybe the parent has got some variables, some memory locations that are associated with the parent. Now, whether those variable locations will be visible to the child process or not, so that is one type of decision.

In some cases, so we may want that those locations be visible for some other cases we may say they know they should not be visible to the children processes. So, this way there can be different types of decision. And as so to be say it frankly so there is no hard and fast rule like which one to be followed; and different ways designers they follow different degree of sharing.

So, all of these are common like who can have this parent and children that share all resources they may they may, child may share a subset of parents resources or they share no resources, that is a resource sharing option. Another option is that execution options. So, execution option maybe that parent and children execute concurrently, and parent or the parent waits until children terminate. So, this is like this that we have got a parent process ok. So, it was executing. So, when it came to say this point, so it create a new process, it has create a child process.

Now, what happens in the so after that again it goes on doing something else. So, it is like this that this process suppose this parent process was P 1; parent is P 1 and the child that is created is the is the process C 1. Now, in my system I have got two processes, one is P 1 which after creating the child will continue, and another is the new child process that is created, so it will do something.

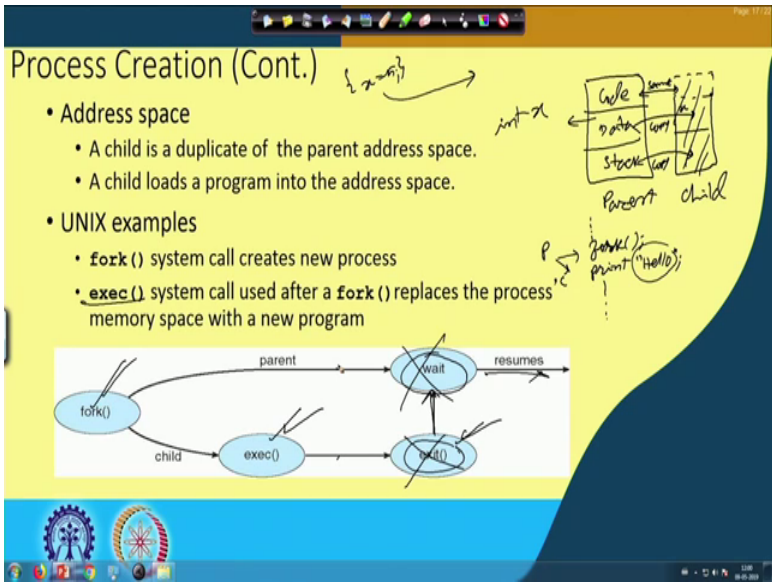
Now, as far as this system is concerned, so there may be different options like this P 1 process it may continue parallely with C 1 like in the ready queue that I have I may have the entry for P 1 as well as the entry for C 1, and whichever process is scheduled by the a short-term scheduler, so that will get chance for execution. So, P 1 or C 1 may execute and then that that again that execution maybe mixed up also maybe C 1 gets the chance for execution C 1 executes for some time and then it gets descheduled and then this P 1 gets chance for execution. So, and this range P 1 executes then P 1 get descheduled, again C 1 executes for sometime so it can happen like this.

So, as if so P 1 and C 1, they are two parallel processes. Now, there is no dependency between them. So, they are two parallel processes. Other possibility is that once this P 1 has created the child, it waits for this child to be over. So, this is waiting so, as soon as this process is created; child process is created, the parent process P 1 it is put into some event wait state; it is put into some event wait state for the child to be over.

And only when the child gets over, then the process P 1 will be taken back from that to wait state and will be put back on to the ready state. So, that way it will wait for some more sometimes when the child process is executing. So, parent is suspended.

So, they again these are the two different thoughts ok. So, it is not mandatory that one of them will always be superior to the other but both of them are available in different operating systems. And some a many operating system, so they will provide some mechanism how you can make the parent process to wait for the child. And sometimes you want that, sometimes you do not want that. So, accordingly, we have to use the facilities provided by the operating system.

(Refer Slide Time: 24:24)



Process Creation (Cont.)

- Address space
  - A child is a duplicate of the parent address space.
  - A child loads a program into the address space.
- UNIX examples
  - `fork()` system call creates new process
  - `exec()` system call used after a `fork()` replaces the process' memory space with a new program

Now, regarding the address space, so this address space a child is a duplicate of the parent address space, so this is one option. Other option is that the child loads the program into the address space. So, like the parent process, so it had its code data and stack segments. Now, it may so happen that the child process, it also uses the same code data stack segments of the parent process. So, this is one possibility.

Other possibility is that there since this child is a new process, it should have its own code data and stack segments, so that is the address space of the child is totally different from the address space of the parent, and we can have this address spaces loaded separately.

So, if you look into this UNIX operating system, so it follows a mix up of all these things. Like it for creating a new process, so fork is the system call. So, if fork system call it creates new process, and then there is a exec system call that can be used after a fork that replaces the process memory space with a new program.

So, by default when the so, if you look into the parent process, so parent has got this code data and stack, this code segment data segment and stack segment. So, this is the parent process. Now, when the child process is created, so it uses the same code segment, the code part is a shared. So, this is shared, but this data and stack so they are separate. So, they are new, but it is copied here. So, this is also copied here. So, this is a copy, this is a copy, and this is same that is shared. So, code part is shared, but data and stack are separate.

So, if I write a program like this say I write a fork here, and after that I write a print, Hello. Now, if this program is executed then after this fork has been done, so I have got two processes the parent was there and the new process child is also created. Now, since the parent will continue, so parent will print this Hello to the screen. And since child is also using the same address space as the parent, so child will also execute the code and it will also print the Hello. As a result, this Hello will be printed twice ok. So, we so this code part is shared.

Now, that data, so if I have got a variable x here, so if I have got an integer x in this in the data segment, now it will also have an x here, but these x and these x they are different. For example, taking going back to the previous case like if I say that if this is the parent process and here it assigns x equal to 5, then this x equal to 5 will not be visible to the child process. So, for example, if this parent does after forking it does x equal to 5, so that is not visible to the child process, it will see the old value of x.

So, we will see that in more detail. But another thing is that some. So, this is by default what is provided by the UNIX, but we may not want it. So, you may want that the child should do something new and for that purpose there is another system called exec. So, what is exec called as is that it replaces the processes memory space with a new program. May be the new process that I have created here I want to execute some other program. So, what it will do this exec calls, so it will over write all these portion and it will be loading, it will be copied by with the new program that you want to execute. So, that is why this child process will be executing a new program.

Now, the parent process may wait for the child to be over ok. So, when the child process executes exit on completion, then it sends an information to the parent process which was waiting, and this wait process this child can this parent can continue from this point.

So, these are the different system calls that we have where the fork is one system call for creating, exec is for overwriting the memory part for the new for the child process. And exit is for exiting the execution, and this exit will give information if the parent was waiting, then it will get the information that the child is over. And it can if it wants that type of synchronization then it can resume from that.

Ideally if this exit is not there, then this parent will parent is has not given the call to wait. Then this after this point this parent and child they are two parallel processes, and they are scheduled independently. So, that way it can continue. But if this wait is there, then there may be synchronization between parent and child, so parent will wait for the child to be over and then only it will continue.