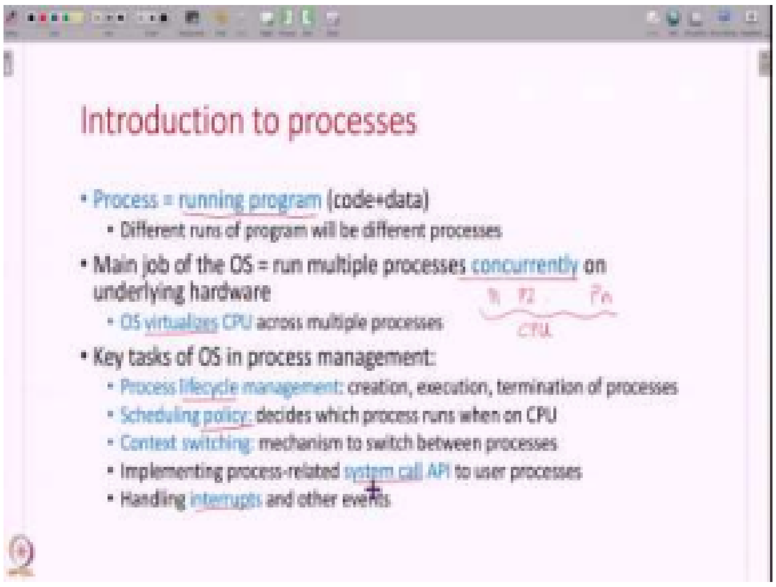
Design and Engineering of Computer Systems
Professor Mythili Vutukuru
Computer Science and Engineering
Indian Institute of Technology Bombay
Lecture 06
Processes

Hello everyone, welcome to the sixth lecture in the course Design and Engineering of Computer Systems. So, in this lecture, we are going to study about how the operating system manages processes. So, in the next few lectures, we are going to focus mostly on the operating system, which is the building block for some of the later concepts we are going to study in this course. So, let us get started.

(Refer Slide Time: 00:39)



Introduction to processes

- Process = running program (code+data)
  - Different runs of program will be different processes
- Main job of the OS = run multiple processes concurrently on underlying hardware
  - OS virtualizes CPU across multiple processes
- Key tasks of OS in process management:
  - Process lifecycle management: creation, execution, termination of processes
  - Scheduling policy: decides which process runs when on CPU
  - Context switching: mechanism to switch between processes
  - Implementing process-related system call API to user processes
  - Handling interrupts and other events

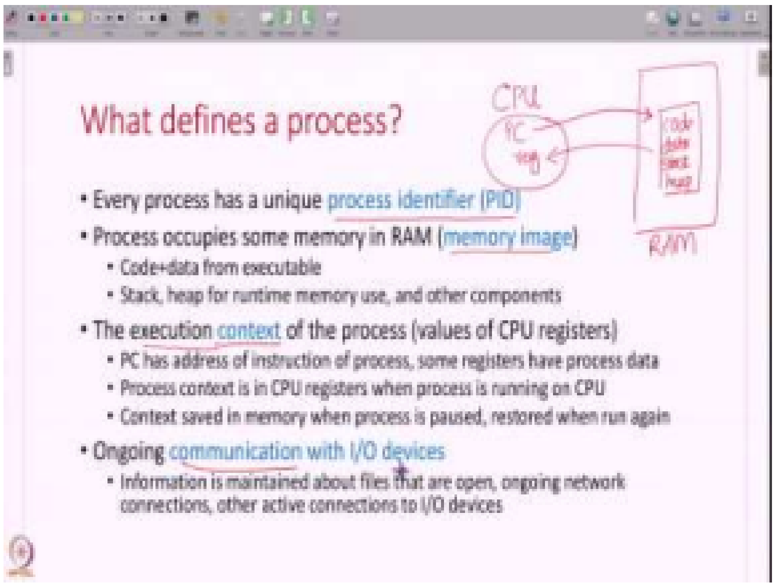
So, we have already seen what is a process. A process is a running program, you write some program that has some code and data in it, and then when you run it, it becomes a process. And when you run the program multiple times at different times, each different run of the program will be a different process. So, a running program one instance of a running program is a process.

So, we have already seen this concept that the main job of the operating system is to run multiple processes concurrently on the underlying CPU hardware. So, we also say this as the OS virtualizes the CPU across multiple processes. What does this mean? This means that each process thinks it has complete control over the CPU, if you have processes P1, P2, Pn each thing it is running on the CPU on its own, whereas the operating system is involved in multiplexing all of these processes on the same underlying CPU.

So, we have all seen a brief overview of what the operating system does in process management. We have seen that the operating system manages the lifecycle of processes, that is, it creates processes, executes them, terminates them, it also has a scheduling policy that decides which process has to run when on the CPU. It also performs context switching, once the policy decides that this process has to run or that process has to run the operating system is involved in switching the context of the process.

And it also implements various system calls. So, the system calls are the services that the operating system provides to user processes, and it also handles interrupts. So, a brief overview of all of this that the operating system does, we have seen in the previous lecture, in the next few lectures, we are going to go into a lot of detail to understand how exactly the operating system does all of these things.

(Refer Slide Time: 02:45)



What defines a process?

- Every process has a unique process identifier (PID)
- Process occupies some memory in RAM (memory image)
  - Code+data from executable
  - Stack, heap for runtime memory use, and other components
- The execution context of the process (values of CPU registers)
  - PC has address of instruction of process, some registers have process data
  - Process context is in CPU registers when process is running on CPU
  - Context saved in memory when process is paused, restored when run again
- Ongoing communication with I/O devices
  - Information is maintained about files that are open, ongoing network connections, other active connections to I/O devices

So, now, let us come back to our definition of what is a process, let us understand this in a little bit more detail. So, every process has a unique process identifier or PID, this is sort of like if you are a student, this is your roll number. So, this is your unique identifier that every process has in the system.

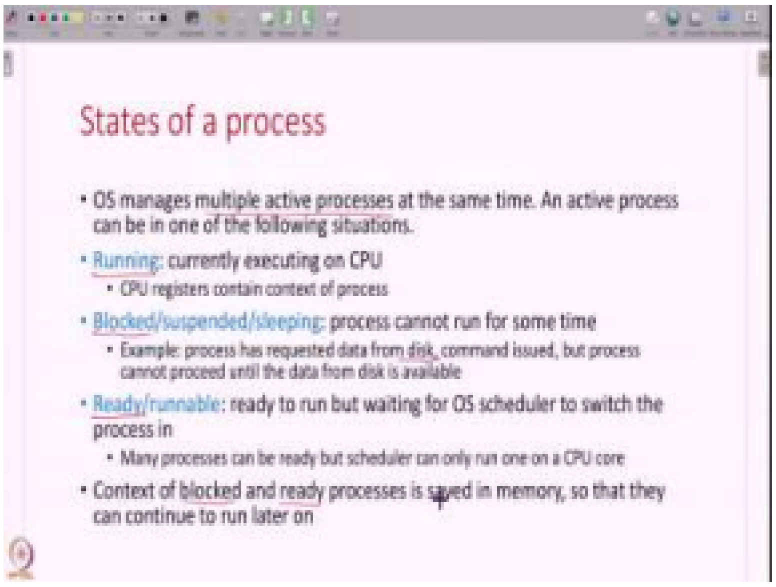
And what else defines a process, every process has a memory image that is in RAM a process occupies some memory in RAM and the code, data all of these things have a process, code, data, stack, heap and various other code, various other pieces of a process are stored in its memory image in RAM.

The other thing that a process has is its execution context on the CPU. So, when the CPU is running a process, the program counter is pointing to some address, the instruction that has to be run next, the various CPU registers have some data stored from the process memory image into temporary storage in the CPU. So, all of this the value of all of these CPU registers constitutes the context of a process.

Whether it is the program counter or other general-purpose registers and so on. So, when a process is executing on the CPU, this context is present in the CPU registers. And when we want to stop running this process temporarily, when we want to pause this process, all of this context will be saved somewhere. So that when we want to resume the process, this context will be restored again.

So, this is how the operating system multiplexes multiple processes by saving the context of a process and restoring it later on. And every process is also defined by some kind of communication with IO devices. So, our processes has some files open on disk, it is talking over the network to some other computers. All of this information about IO devices, is also maintained with respect to the process. All of these things define a process.

(Refer Slide Time: 05:06)



States of a process

- OS manages multiple active processes at the same time. An active process can be in one of the following situations.
- Running: currently executing on CPU
  - CPU registers contain context of process
- Blocked/suspended/sleeping: process cannot run for some time
  - Example: process has requested data from disk, command issued, but process cannot proceed until the data from disk is available
- Ready/runnable: ready to run but waiting for OS scheduler to switch the process in
  - Many processes can be ready but scheduler can only run one on a CPU core
- Context of blocked and ready processes is saved in memory, so that they can continue to run later on

So, a process also can exist in one of multiple different states, an operating system is managing multiple active processes and each of these processes can be in different states or situations in its execution. What are these states of a process? A process can be running, at any point of time on a CPU core exactly one process is actually running on the CPU. What does it mean? The CPU registers contain the context of a process, the program counter is pointing to some instruction of the process, registers have data from the process.

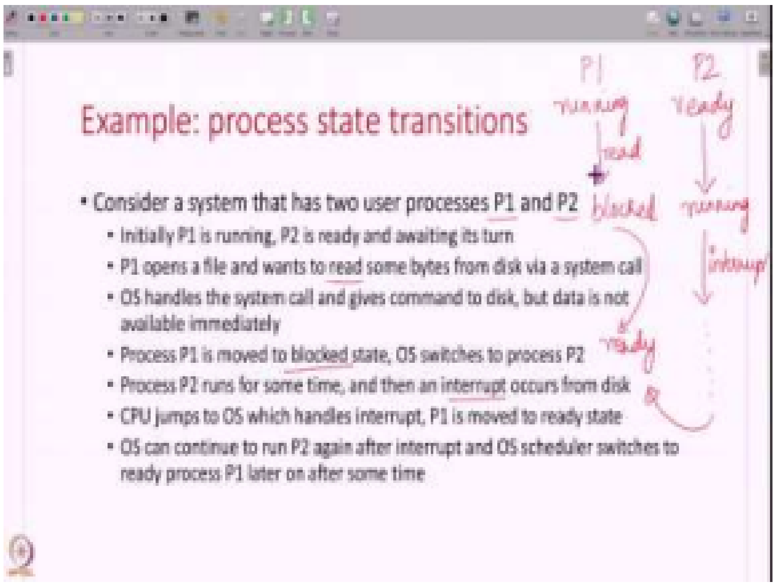
So, a process can be in a running state. And on any CPU core, you can only have one process in a running state at any point of time. Next, a process can be in what is called a blocked state. It is also called a suspended or a sleeping state. What is this? This happens if a process cannot run for some time temporarily, but can run again in the future that is called a blocked process.

For example, a process has made a system call and has started reading some data from the disk, reading data from the disk this is a blocking operation. So, when this happens, the operating system will start, the command will issue the command to the disk hardware we have seen how this happens, but the disk hardware takes a long time to return the data.

So, in this meantime, this process cannot run, it is the next line of code depends on this data being available. So, the process cannot run. So, in such cases, what do we do, this process is blocked, it is suspended or it is put to sleep, its context is saved and the operating system will run some other process. So, that is the blocked state.

Then there is also what is called the ready or the runnable state. So, some processes are ready to run, but the operating system can only run one process at a time on a CPU core. So, these ready processes are waiting for their turn to run, they are waiting for the OS scheduler to context switch them in at some point in the future.

So, note that, there can be many ready processes and the scheduler has this choice to pick one of them to run on a CPU core. And as we have already seen, if you are blocked process or already process that is not currently running on the CPU, the context is saved somewhere in memory, so that it can be restored later on when the process can be run on the CPU.
(Refer Slide Time: 07:41)



Example: process state transitions

- Consider a system that has two user processes P1 and P2
  - Initially P1 is running, P2 is ready and awaiting its turn
  - P1 opens a file and wants to read some bytes from disk via a system call
  - OS handles the system call and gives command to disk, but data is not available immediately
  - Process P1 is moved to blocked state, OS switches to process P2
  - Process P2 runs for some time, and then an interrupt occurs from disk
  - CPU jumps to OS which handles interrupt, P1 is moved to ready state
  - OS can continue to run P2 again after interrupt and OS scheduler switches to ready process P1 later on after some time

So, I would just like to give you some examples of what are these process state transitions. Just taking a simple example of a system which has two processes, say P1 and P2. So, initially, this process P1 is in running state it is running on the CPU and this process P2 is also ready, but the scheduler has picked P1 for some reason and P1 is running and P2 is ready. So, now what happens, suppose P1 makes a read system call it wants to read some data from the disk.

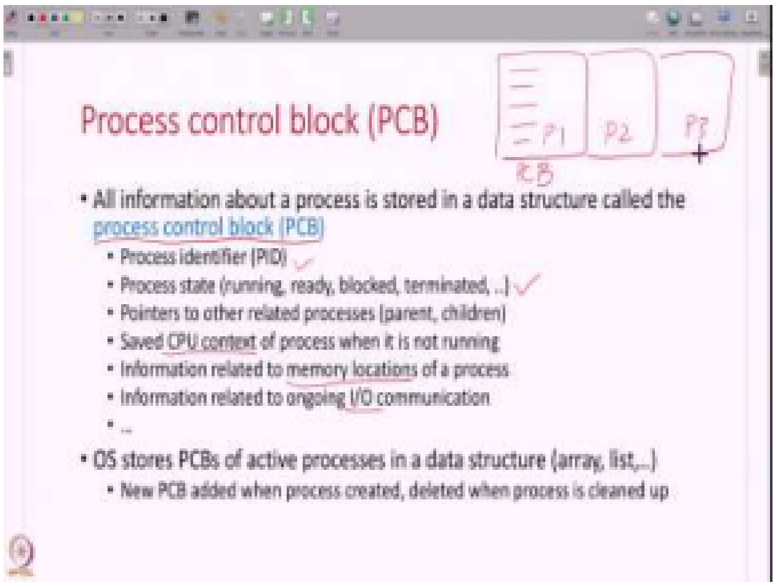
So, P1 opens a file it wants to read some data from the disk and the operating system handles this operation handles the system call and gives the command to the disk but the data is not ready. So, in this situation what happens P1 moves into the blocked state, P1 was running and it moved into the blocked state why, because it cannot proceed until the disk data is available.

And at this point the operating system will switch to process P2 and process P2 will become the running process, the operating system will save the context of P1, restore the context of P2 and P2 starts to run. And after some time, after P2 runs for some time maybe the interrupt from the disk happens the disk says the data is ready, it raises an interrupt at this point when the operating system handles the interrupt the data for P1 is ready, P1 is ready to run then at this point P1 can move to the ready state.

And P2 can continue to run for some more time, it is still running, running and at a later point of time once again the operating system can switch to P1. So, it is not necessary that as soon as the interrupt arrives the operating system will drop P2 and switch to P1. This really depends on the scheduler policy, even if P1 has obtained its data and it is ready to run operating system can wait and continue to run P2 for some more time. And at a later point of time, it can switch to P1.

So, these are some examples of process state transitions where P1 went from running state to a block state when it made a system call, then from the block state it went to the ready state once again when the interrupt occurred. And P2 was in a ready state and when P1 got blocked P2's turn came it went into running state that all of these are some examples of how processes move across these various states.

(Refer Slide Time: 10:34)



Process control block (PCB)

- All information about a process is stored in a data structure called the process control block (PCB)
  - Process identifier (PID)
  - Process state (running, ready, blocked, terminated, ..)
  - Pointers to other related processes (parent, children)
  - Saved CPU context of process when it is not running
  - Information related to memory locations of a process
  - Information related to ongoing I/O communication
  - ..
- OS stores PCBs of active processes in a data structure (array, list,..)
  - New PCB added when process created, deleted when process is cleaned up

So, now, as we have seen the operating system has many different processes, each of them is doing different things in different states. So, how does the operating system keep track of all of these processes. So, for every process the operating system maintains a special data structure called the process control block or PCB.

So, this is one data structure in which all the information about a process is stored and there will be several such blocks for the different processes, this is the information about process P1, about process P2, about process P3, and so on in some bigger data structure, all of these will be maintained.

So, for every process we have a PCB and what is all the information that is stored in the PCB about a process, we have things like the process identifier, the process state, what state is the process in is it ready to be run is it blocked, the scheduler must know this information when it is trying to do a context switch.

And various other pointers to other related processes. So, as we will see, a process also have some parents, children and other such relationships and you know information about all the relations of processes also maintained in the PCB of the process. And the other important thing that the PCB has is the saved CPU context.

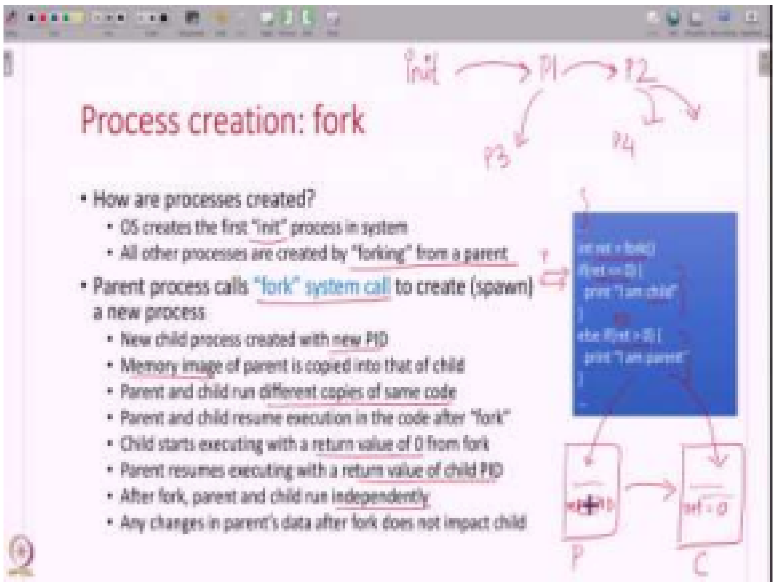
So, when a process is not running when it is either blocked or it is ready waiting for its turn in such cases, the context of the process has to be saved somewhere, so that it can be restored later. where do we save that, we save that in one of the fields of the PCB, so that this is one place where all the information about the process is kept track of. So, the CPU context is also saved in the PCB when the process is not running.

And the PCB also has other things information related to the memory locations of a process, where is the process located in RAM for the OS to find it when required, and information related to other IO communications. There are many other fields in the PCB, as we keep studying in the course, as we learn more about operating systems, you will find more information about the PCB that we will come across later in the course.

And all of these PCBs of all the processes are stored in some other bigger data structure say it could be a linked list, an array, heap or whatever if you have taken a data structures course, there are many different ways of organizing multiple pieces of information and the OS also maintains all the list of active PCBs in some such data structure.

And whenever a new process is created, a new entry is added to this list of PCBs, when a process terminates and its memory is deleted, its information is deleted, then this entry goes away and so on. So, this is a dynamic data structure that is maintained by the operating system.

(Refer Slide Time: 13:29)



Process creation: fork

- How are processes created?
  - OS creates the first "init" process in system
  - All other processes are created by "forking" from a parent
- Parent process calls "fork" system call to create (spawn) a new process
  - New child process created with new PID
  - Memory image of parent is copied into that of child
  - Parent and child run different copies of same code
  - Parent and child resume execution in the code after "fork"
  - Child starts executing with a return value of 0 from fork
  - Parent resumes executing with a return value of child PID
  - After fork, parent and child run independently
  - Any changes in parent's data after fork does not impact child

```
int ret = fork()
if(ret == 0) {
 print "I am child"
}
else if(ret > 0) {
 print "I am parent"
}
...
```

So, now, let us understand the life cycle of processes, how are processes created, how do they die, all of this information we will begin to study. So, how are processes created? So, the first process when any system starts the operating system creates the first process which is also called the init process. And from then on, all subsequent processes are created by forking from the parent process.

So, you have the init process that is created. The init process does what is called a fork and creates another process say P1 and this process P1 once again does a fork at a later point of time creates another process P2 and P1 creates a fork later creates another process p3, then P2 can fork and create P4 and so on. In this way, the processes one process forks itself and creates a child process.

So, processes are always in this kind of a family tree, you cannot just create a process in isolation it has to come from some parent process by forking from a parent process. And how is this done? How does a process create a child process like this? There is a system called, called the fork system call. Using which a parent process can create a child process. So, this is an example of some code using the fork system call.

So, here is some code, when the parent processes fork. At this point, two copies of the parent are created; a new child process is created with a new PID. But the memory image of the child is exactly the same as the memory image of the parent. That is, you have this code running in your parent process.
And when it calls fork, another new child process is created, but it has the exact same code, the same code and the data are there in both the parent and the child. The memory image is

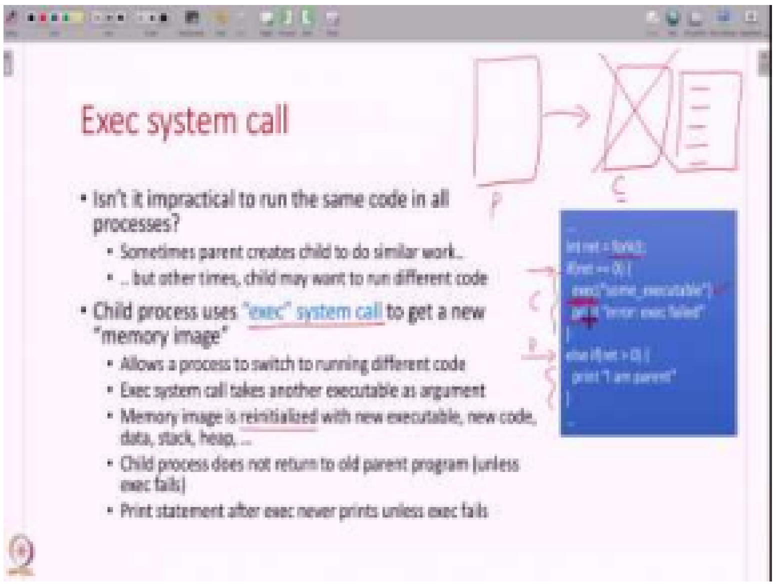
copied, this is a separate copy the child can do its own execution, the parent can do its execution. But it is an exact copy of the parent. The parent and child run different copies of the same code plus data right after fork.

So, fork is nothing but just duplicate a process create another copy, and the parent and child can then run on their copies. And both the parent and child after the fork system call. So, here, the parent was running this code, and it does fork, at this point right after fork both the parent and the child resume execution at this exact line of code after fork.

So, then will they run the exact same code? No, they do not. They both resume after fork, but the return value from fork is different. So, this fork system call returns 0 in the child and returns a nonzero value, returns the PID of the child in the parent. So, the parent and child both have two copies of the exact same code.

In the parent's code, this return value will be the PID of the child; it will be a nonzero value. And in the child code, this return value from fork will be equal to 0. So, whatever code you write in this, return value equals 0, in this if clause this will be run by the child. And whatever code you write here will be run by the parent.

So, parent resumes execution with the return value of child PID, child resumes execution with a return value of 0. And after fork, they both run independently. And if you change any data in the parent's process, it does not impact the child and vice versa. So, this is how any process in your system is created from forking from a previous parent process.
(Refer Slide Time: 17:30)



Exec system call

- Isn't it impractical to run the same code in all processes?
  - Sometimes parent creates child to do similar work..
  - .. but other times, child may want to run different code
- Child process uses "exec" system call to get a new "memory image"
  - Allows a process to switch to running different code
  - Exec system call takes another executable as argument
  - Memory image is reinitialized with new executable, new code, data, stack, heap, ...
  - Child process does not return to old parent program (unless exec fails)
  - Print statement after exec never prints unless exec fails

```
...
int ret = fork();
if(ret == 0) {
 exec("some_executable");
 print "error: exec failed"
}
else if(ret > 0) {
 print "I am parent"
}
...
```

So now, you might be asking the question, is not it just impractical, I mean to run the exact same code in multiple processes, what if I want to run a different process, I want to run a browser, I want to open a computer game. I do not want to fork an existing process that is already running. So, that is a valid doubt.

Therefore, even though a process is created as an exact copy of the parent, the child process can actually then go ahead and change its memory image. So, sometimes a child might want to run different code from the parent. So, it is valid that sometimes the parent and child might do the similar work.

Suppose, it is a web server process and the parent web server process might want to do more work process more requests and therefore it will you know have a child process that is doing the exact same thing that is valid, but sometimes your child might want to do something else. In which case we have what is called the exec system call.

So, what is the exec system call? Going back to the same code here, the parent has forked the child. Now, both the parent and the child resume execution here, the child runs this code, and the parent runs this code. And you can see the child is now calling the exec system call with some other executable, so all of this code is one executable the child is saying I want to execute some other executable.

So, what is happening here initially, you have your parent process its code is copied into the child process, the exact same copy is made and then when you do the exec system call the child removes this memory image and adds a different piece of code, a different executable in its memory, so that is the exec system call. It allows a process to switch to running a different executable or a different set of code plus data from what its parent was doing.

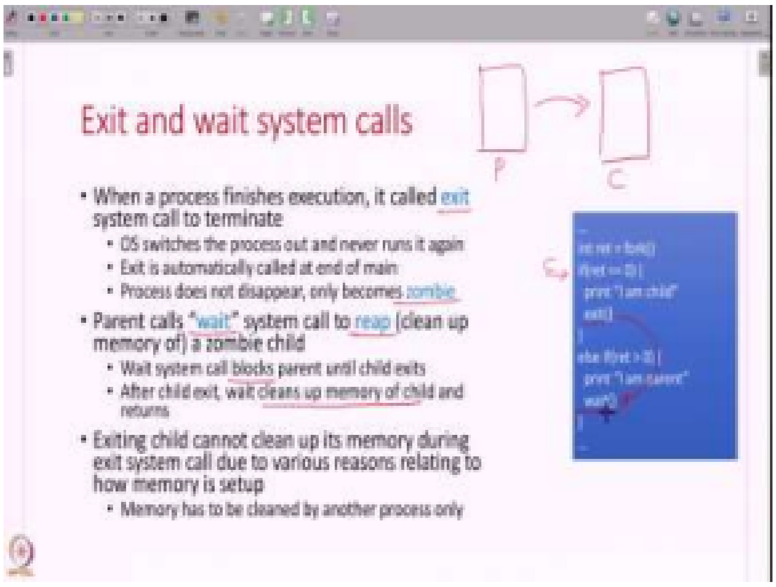
And the exec system call, it takes some executable as an argument and it will reinitialize the memory image the code, data, stack, heap, everything is re-initialized to run this new program, this new executable. And then the child code no longer runs anything after exec. Whatever you put after exec, this is gone, the child code will not execute any of these things.

Why, because this memory image has been erased and a new memory image has come. So, the print statement will never run, if your exec works fine. The only way this print statement will print is if this exec fails, if for some reason that child could not reinitialize, its memory image, then it will go back to whatever memory image it had before the parents' memory image, or that is old memory image, and it will continue to run that.

So, in this code, if you do exec, if this exec works, then all of this code has gone. You will never run anything that is thereafter Exec. But if the exec fails in the child, then it does not have a new memory image to go to. So, then it will continue to execute whatever code is thereafter exec, like it will print this error message for example.

So, this is how the exec system call works. The fork system call creates a copy of the memory image from parent to child and the exec system call allows a child or any other process to rewrite its memory image with a different program.

(Refer Slide Time: 20:57)



Exit and wait system calls

- When a process finishes execution, it called exit system call to terminate
  - OS switches the process out and never runs it again
  - Exit is automatically called at end of main
  - Process does not disappear, only becomes zombie
- Parent calls "wait" system call to reap (clean up memory of) a zombie child
  - Wait system call blocks parent until child exits
  - After child exit, wait cleans up memory of child and returns
- Exiting child cannot clean up its memory during exit system call due to various reasons relating to how memory is setup
  - Memory has to be cleaned by another process only

```
...
int ret = fork()
if(ret == 0) {
  print "I am child"
  exit()
}
else if(ret > 0) {
  print "I am parent"
  wait()
}
...
```

So, now, this is how a process is created, how it executes any programs, and then when a process has finished execution, it will call the exit system call. So, this exit system call will basically terminate the execution of the process that will go back to the operating system and the operating system will find some other process to run. So, this exit system call terminates the process.

So, this exit is automatically called at the end of main, if you have written a program, and you wonder, wait, I have never done any exit, when you reach the end of your code, they exit system call is automatically invoked. But an interesting thing about exit is that when a process exits, it does not fully disappear. It is not removed from the list of active processes and so on.

But it only becomes a zombie. It goes into this weird state called a zombie state that is it can no longer run, but it is still around as a zombie. And what do we do with these zombies, the parent has to do another system call called the wait system call, which will, what is called, reap the zombie child, that is, once the child exits and becomes a zombie, the parent has to call an another system call called wait, which will then clean up the zombie, which will fully terminate the process.

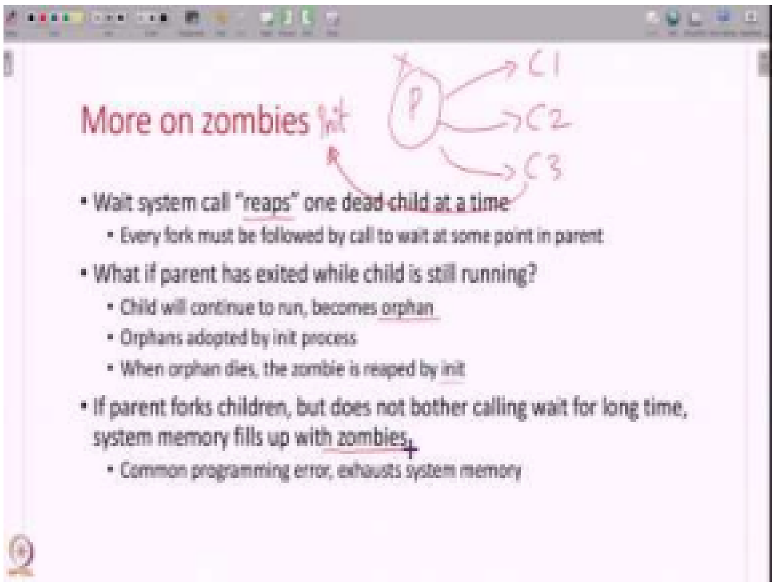
So, let us see this example. Here. Your parent has done a fork, it has created another child process and this child process has its own memory image. And here, the child returns with the value of 0, the child executes some code. Here, it just prints something and then it exits. At this point the child process has finished execution.

Now, but the memory and everything of the child process still exists. It no longer runs on the CPU, the OS will run some other process, but the child is still there as a zombie. Then at some point, you want the parent process to call wait. At this point, when the parent calls wait in its code, this wait system call will block until the child dies. And once the child exits that is when this wait system call will return.

The parent will wait literally that is why it is called the wait system call. The parent will wait for the child to finish its job. And then the parent can continue. And when the parent does this wait, that is when the memory of the child process is cleaned up. If a parent does not do wait, then the child has terminated it is not running, but it is still existing in the system as a zombie.
So, why is this complication? When a process exits, why do not we just clean up its memory? What is this, parent has to wait and all of that, why do we need this? Well, there are certain subtle reasons why this happens that we cannot go over in this course, but it turns out that the exiting child cannot clean up its memory on its own due to how the memory is set up in operating systems this cannot be done it is not easy to be done.

Therefore, when a process exits, some other process has to clean up its memory. So, that is why we have the parent perform the wait system call. And we know the parent knows that a child exits it has forked the child and therefore it will also call wait on the child process.

(Refer Slide Time: 24:31)


More on zombies

- Wait system call "reaps" one dead child at a time
  - Every fork must be followed by call to wait at some point in parent
- What if parent has exited while child is still running?
  - Child will continue to run, becomes orphan
  - Orphans adopted by init process
  - When orphan dies, the zombie is reaped by init
- If parent forks children, but does not bother calling wait for long time, system memory fills up with zombies
  - Common programming error, exhausts system memory

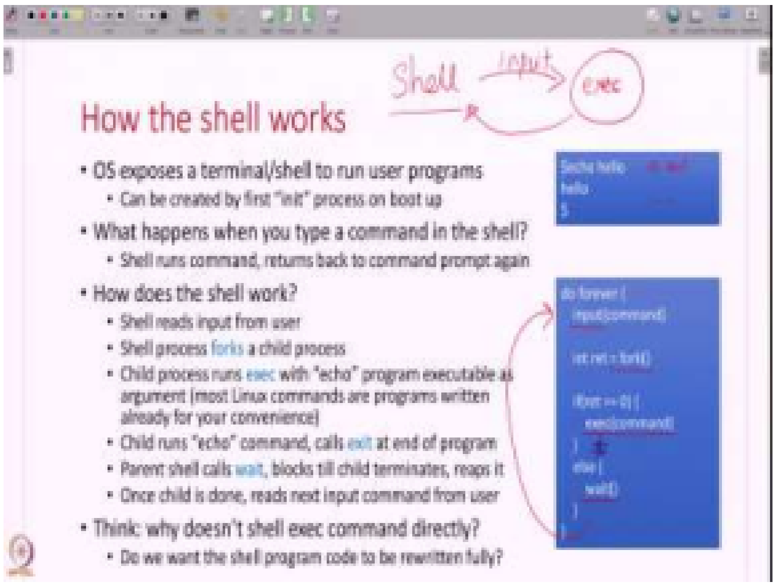
So, a little bit more detail on zombies because this is really important when you build real life large systems. So, this wait system call reaps or cleans up the memory of this is also called reaping. It reaps one dead child at a time. So, if a parent has forked, a process has forked multiple processes. It has to call wait multiple times. It has to call wait three times here in order to clear the memory of all of these three child processes.

So, the next question comes up what of this parent itself has exited, this parent is gone, then who will clean up the child. So, such children whose parents have exited, they are called orphans. And these orphans will be adopted by some other process, say the init process. This is the convention. So, the init process becomes the guardian or the pseudo parent of these child processes, which do not have parents.
And when this orphan processes terminate, then the init process will reap them. So, the zombies are reaped by the init process. So, another important thing to notice, the init process will only reap the children of other processes only if they are orphans. But if the parent is still running, and a child has terminated, the init will not reap the child.

All such children, where the parent is still alive, those children are still the responsibility of the parent and the parent must reap them by calling wait. If a parent forks a child, but does not call wait for a long time, then what will happen, all of these child processes have terminated, but they are still using up memory, their memory images are there in RAM, they are there in the list of PCBs, the system memory fills up with zombies.

And this is a very common programming error where your system memory can get exhausted. So, this is something to keep in mind. When you are building large systems, if you has a process as an application, you are creating multiple child processes to do some work, then you have to reap them after some time, you have to ensure that their memory is cleaned up.

(Refer Slide Time: 26:48)



How the shell works

- OS exposes a terminal/shell to run user programs
  - Can be created by first "init" process on boot up
- What happens when you type a command in the shell?
  - Shell runs command, returns back to command prompt again
- How does the shell work?
  - Shell reads input from user
  - Shell process forks a child process
  - Child process runs exec with "echo" program executable as argument (most Linux commands are programs written already for your convenience)
  - Child runs "echo" command, calls exit at end of program
  - Parent shell calls wait, blocks till child terminates, reaps it
  - Once child is done, reads next input command from user
- Think: why doesn't shell exec command directly?
  - Do we want the shell program code to be rewritten fully?

```
$echo hello
hello
$
```

```
do forever {
  input(command)

  int ret = fork()

  if(ret == 0) {
    exec(command)
  }
  else {
    wait()
  }
}
```

So, now a simple example let us put all of these concepts together and understand how the shell works. So, every operating system exposes an interface like a shell or a terminal, in which you can run user programs, for example if you run a program like echo, hello, then hello gets printed. Or if you run your a dot out, then your C program executes and the output from the C program gets printed.

So, the shell is a way for users to run their user programs. So, what happens when you execute a program in a shell or you type a command in a shell, what exactly happens? This is a good example to understand all of the system calls we have just seen. So, this is a very simple model of how any shell works. It will first read the input command from the user. And then it will fork the child process.

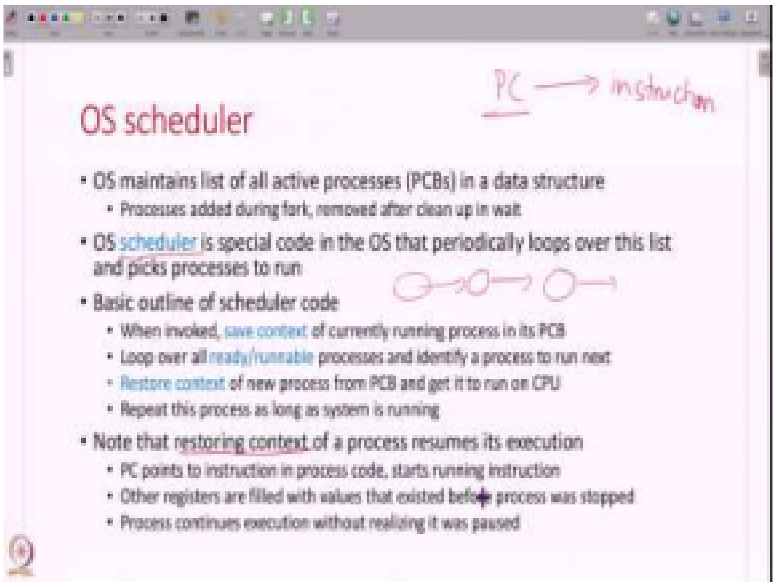
So, the shell will fork a child process whenever the user gives an input, it will fork the child process. And in this child process, it will execute the command. So, fork and then in the child execute whatever command the user has given, if it is l or echo, or a dot out or all of that has run in the child process. And the parent itself, what will it do, it will simply wait.

So, the shell folks a child, the child runs exec with whatever program executable or command has been given as an argument to the shell. So, commands like LS or echo all of these are just other programs written by somebody else and compiled and kept for you. Like you are a dot out and Linux comes with all of these built-in programs that you can directly run.

So, what shell will do is it will simply fork and call exec with that command as the argument. And the parent shell will wait. And once this command finishes, then wait returns, and the shell will go back. Take the next command input, run it again, next command input and run it again. So, this is how your simple terminal works.

So, you might have a question, why does not the shell just directly execute the command itself? Why is it even forking a child? Well, this is a subtle question, think about it. So, for example, if I directly instead of doing fork, if I just directly exec the command over here, what will happen, all of the shell logic will get destroyed and the shell cannot go back and display a command prompt and ask the user for the next input again.

So, we want the shell process to remain, we only want another process to change its code and run some commands, but we want the shell also to be running. Therefore, we do this fork, exec, wait, logic in the shell. So, this is a simple program. A shell is a simple program that you can build using these four, exec, wait, system calls.
(Refer Slide Time: 29:45)



OS scheduler

PC → instruction

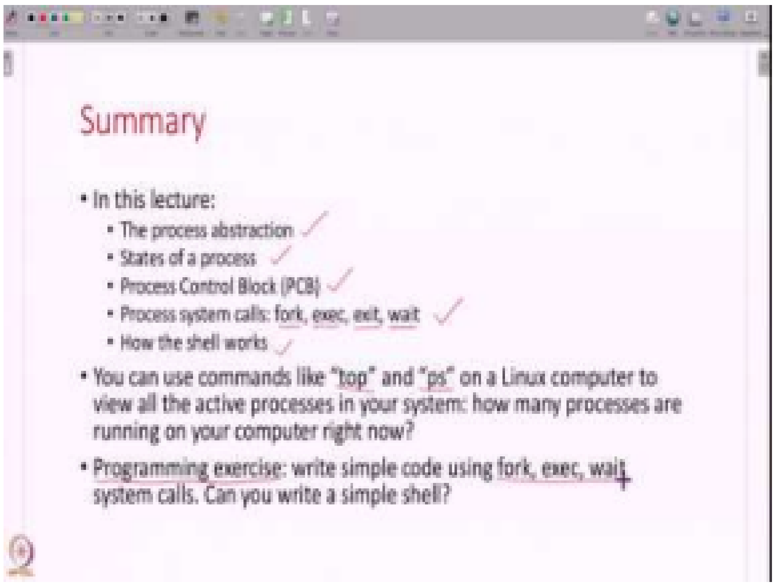
- OS maintains list of all active processes (PCBs) in a data structure
  - Processes added during fork, removed after clean up in wait
- OS scheduler is special code in the OS that periodically loops over this list and picks processes to run
- Basic outline of scheduler code
  - When invoked, save context of currently running process in its PCB
  - Loop over all ready/runnable processes and identify a process to run next
  - Restore context of new process from PCB and get it to run on CPU
  - Repeat this process as long as system is running
- Note that restoring context of a process resumes its execution
  - PC points to instruction in process code, starts running instruction
  - Other registers are filled with values that existed before process was stopped
  - Process continues execution without realizing it was paused

So, now finally, we have all of these active processes. And there is a piece of code in the operating system called the OS scheduler, which just goes over this list of processes and it will pick processes to run. So, we are going to study this scheduler in a lot of detail in a future lecture. But for now, it is enough to know that what does the scheduler do.

There is the list of processes, in some data structure, all the processes are maintained, it will go to one process, run it for some time, stop it, save its context, jump to another ready process, restore its context, run it, then again, save context, restore context of another process, run it and so on. In this way, a list of all ready processes in the system are correctly executed by the OS scheduler.

And you have to understand what is the saving and restoring context. I hope we have discussed this many times by now. And I hope it is clear. So, restoring context of a process simply means you just write some values into CPU registers that were saved before. So, suppose you were running, your program counter was pointing to some instruction before, you are about to execute some code before and then you pause the execution then you save this value of the program counter and when you restore it back again, what will happen?

The process will resume execution where you left off. So, registers like program counter other general-purpose registers, all of those values that were there when the process was paused, when you restore those values, the process will resume execution. So, this is a basic outline of what an OS scheduler does, which we will study in more detail in the coming lectures. So, that is all I have for this lecture.
(Refer Slide Time: 31:34)



Summary

- In this lecture:
  - The process abstraction
  - States of a process
  - Process Control Block (PCB)
  - Process system calls: fork, exec, exit, wait
  - How the shell works
- You can use commands like "top" and "ps" on a Linux computer to view all the active processes in your system: how many processes are running on your computer right now?
- Programming exercise: write simple code using fork, exec, wait system calls. Can you write a simple shell?

In this lecture, we have seen what is the process abstraction, what are the various states of a process like running, ready, blocked and so on, what is a process control block or a PCB. We have seen various system calls, how a process is created with fork, how it can run any new executable with exec, how it terminates and how its memory is cleaned up by the parent. And putting together all of these system calls we have seen how the shell works.

So, a small exercise for you is use commands like top or ps if you are on a Linux machine, and you can view all the processes in your system and by giving different arguments to this ps command you can actually see, what are the various states the process is in right now, how much memory is that occupying, how much percentage of the CPU is it using.

All of this information, it is good for you to just open up a terminal run some commands and see for yourself. And as a programming exercise, you can also try to write a simple shell that takes a command and runs the command using these four exec, wait, system, calls you can build a simple shell for yourself to understand these concepts better. So, that is all I have for this lecture. Thank you all and see you in the next lecture.