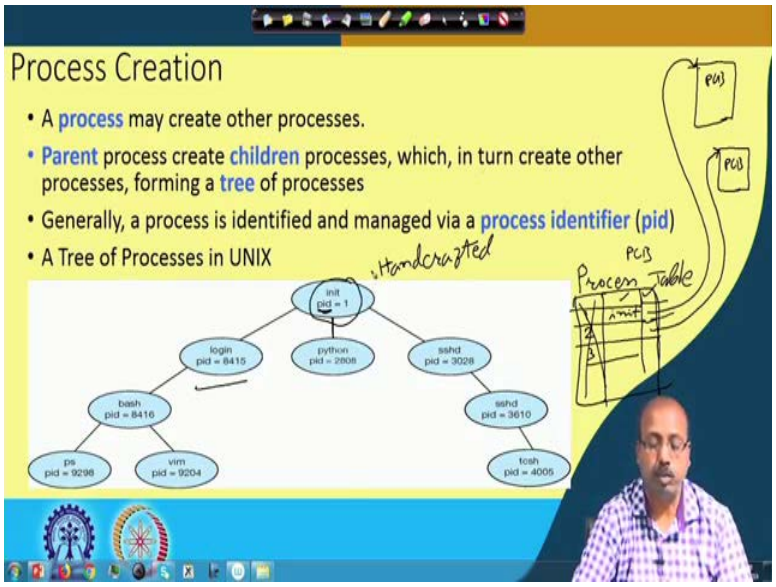
Operating System Fundamentals

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

Lecture – 15
Processes (Contd.)

So, in case of operating system one important part of it is Process.

(Refer Slide Time: 00:30)



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

So, processes are the executable fragments of codes that we have in the system. So, a process related operations the first operation that is important in process is the process creation. So, process creation means a new process is created and may be a new program or new service of the operating system starts in that. Now it has to start from some point so, we have got this first process which is for example, in the UNIX operating system that process is known as the init process that is or the initialization process which is assigned the process identification mark pid as equal to 1.

So, in our last class we have discussed something about the PCBs the Process Control Blocks. So, for the entire process many operating systems they maintain something called a process table. So, process table, this is you can consider it to be an array of indices like if I have got this like index 1. So, that is for the init process now from; so, as more and more processes are created so, they are allocated some slot from this process table. And this process table it is the name of the process and the other part is the pointer to the PCB so, if this pid. So, if this is the PCB for this process then for the first process, then this will have a pointer to this PCB.

Similarly, for the second process when it is created so, there will be a PCB for that and this point this will point to the corresponding PCB. So, this is the PCB for the first process, this is the PCB for the second process so, like that we will have a number of process control blocks and from the process table they will be linked via some pointers.

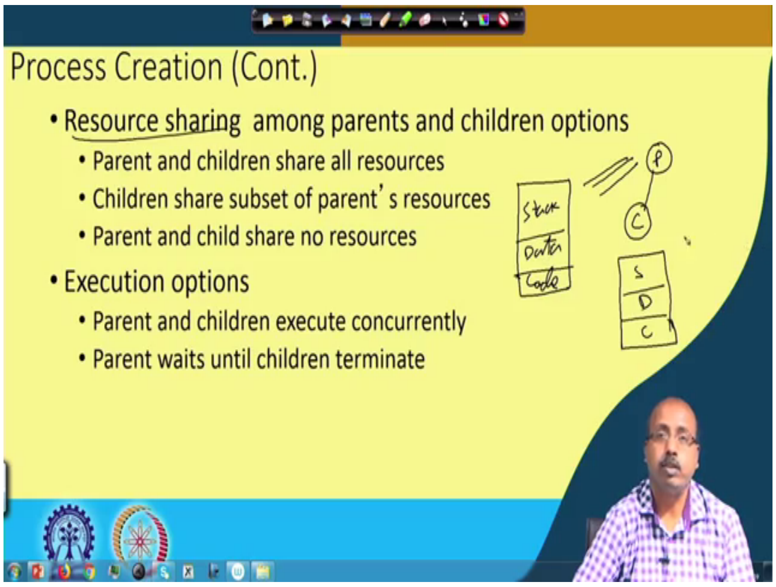
So, the first process that is created so, this is created by the operating system and this is also very often referred to as a handcrafted process. So, this is often called hand crafted process; so, there is nothing like handcrafting, but what happens is that, the other operating system creates this process on its own. So, that is why it is called handcrafted process and from this process other processes are spond. For example, if I have got say 10 different terminals in my computer system, then after the system has initialised so, it should flash login messages to all the 10 terminals.

So, as a result there maybe some login processes created. So, this is one such login process whose pid is a number 8415 that is unique across all the processes that we have in the system. Similarly, maybe some other process got created who for somebody running the python program. So, this may be a snapshot of some point of time that these are some of the processes that are created.

So, in general there is a parent child relationship like when a process creates another process. So, the process which created it is called the parent process and the newly created process is called the child process. So, we have got the parent child relationship between the processes. So, as I said that a process is identified and managed by the process identifier or pid which is nothing, but index of the process table. So, this column though I have shown it explicitly, but you can understand that this need not be there so, we can have only this column and this column present and the first column need not be there. So, that simply acts as index of this process table.

So, this process is identified by this process identifier or pid and a tree of processes in UNIX so, it will look something like this. So, most of the operating system they follow this type of tree type of structure so that this parent child relationship is maintained.

(Refer Slide Time: 04:33)



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

So, for creation of process , so, we have to answer certain question like how resources are shared among parent and child; so, child processes. So, one process so as we know that whenever a process is there so, in the system so it will have certain things like it will have the code part, data part and stack part.

So, the in the code part I will have the code for the process, then the data segment it will have all the global variables that we have and we have got the stack segment, which has got all the procedure calls, then local variables etcetera. So, these are the three memory regions for a process and the from the process table you can understand that a process might have opened a number of files or it may be using number of resources. So, all of them will be part of the process so, they are called the resource.

Now what we have said is this is a parent process and the parent process has created a child process. Now so, for the child process also I should have similar such structure. So, if I say that this structure corresponds to the parent process, then for the child process also I should have the code, data and stack segments. So, this is the code segment, data segment and stack segment this should be there. So, how this sharing will be done? So, how the parent process and child process will do the sharing part?

So, there can be different options, first option is the parent and children share all resources so, everything is shared between parent and child. So, as a result; as if they are part of the same family. So, they can have unrestricted access to all the resources that is there in the family that is allocated to the family. Second option that we can have is children share subset of parents resources.

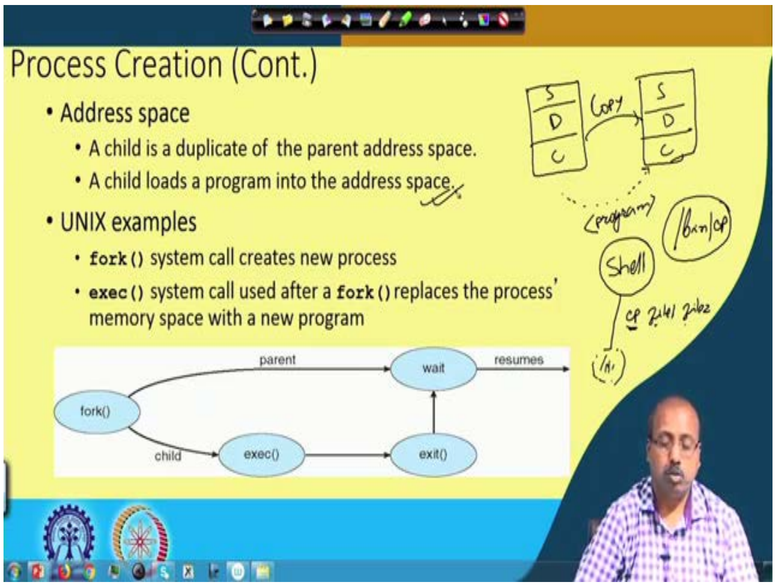
So, this is basically there all resources are not shared only some of the resources are shared. So, this may be another option and the third option or the most stringent one is the parent and child share no resources. So, they are totally two independent processes, there is a parent child relationship of course, but no resource sharing is there. So, this is the third type of alternative that we can have. Now which one is better so, that is a debatable issue right different operating systems they will have different degree of sharing between the parent and child processes.

Then as for as execution options are concerned, one option is that the parent and child they execute concurrently. So, as their two parallel processes that have been created and both are ready for execution. So, if you have got a uni processor system, then the scheduler will pick up any of them and that process will continue. If you have multiprocessor system then maybe this parent and child they are put on to two different processors and they run simultaneously.

But when we say that they are executing concurrently means, we do not distinguish between the order in which the parent and child processes are executive. So, whether they are parent before child or child before parent or an intermix of the two or they are totally parallel; so, there can be many options. The second alternative; second execution option is that the parent waits until children terminate. So, a parent has spond a number of children processes so, naturally the each of the children process it is doing some specific job.

So, it may be that the parent needs to wait for the children to be over to continue further so, this is the second option that we can have. So, again the same thing that there is no hard and fast rule like which option should be followed. So, different operating systems they will follow, different type of; different type options.

(Refer Slide Time: 08:21)



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

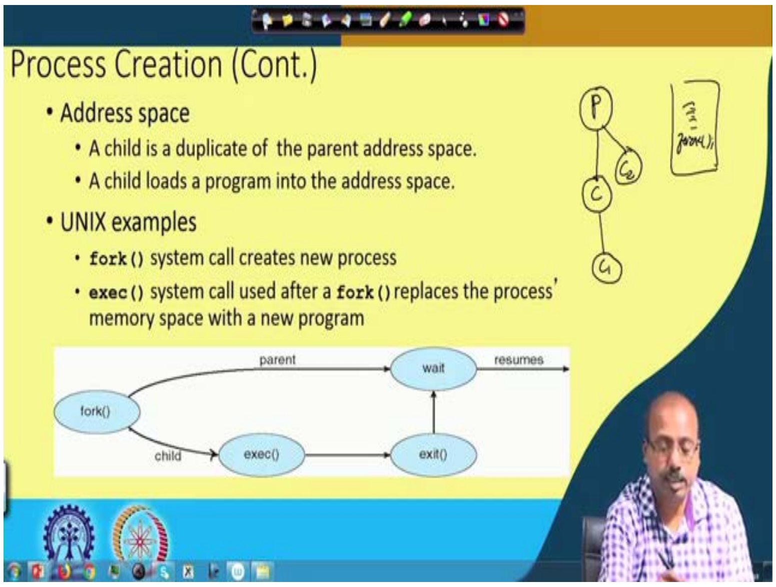
So, address space; so, child is a duplicate of the parent address space. So, what you mean; so, address is basically that code data and stack segment. So, they are actually the range of addresses that can be accessed by a process ok. So, if a child gets a duplicate of that so, that is a; that is one option and the second option is that the child loads a program into the address space. So, if we just in the previous case we were telling that we have got this code, data and stack segments; code, data and stack segment. So, when the child process is created. So, maybe we can say that it gets a copy of this code data and stack segments. So, this is one type of operation; so, this is a copy.

So, both of them see almost similar type of environment, other option is that when this child is created at the same time we mentioned some name of the program to be executed by the child process. So, as a result this child process loads the corresponding program from the secondary storage and that way its code data and stack segments are totally different compared to the parent process. So, both of them are possible again and for example, suppose if user has logged in. So, there is a shell process that is created which takes the comments from the user and executes them one by one. Now how the shell operates is that it create any command that is given suppose the user has given the command to copy file 1 to file 2; copy file 1 file 2.

So, this way; so, this child that is created so, which is not a shell basically it should execute the command CP. So, as a result it will be executing one programs in case I am talking with respect to the UNIX operating system. So, there is a slash bin directory and there is a CP file there. So, it will be copying this file from the secondary storage into the code space of this children of the child process and there the file 1 and file 2 they will be passed as arguments to that program and then this program will be executed. So, in this case the second option is being followed that is the child program is loading and all together different program compared to the parent process.

So, will be looking in more detail the UNIX operating system, how this system this process creation and this things work.

(Refer Slide Time: 11:15)



Process Creation (Cont.)

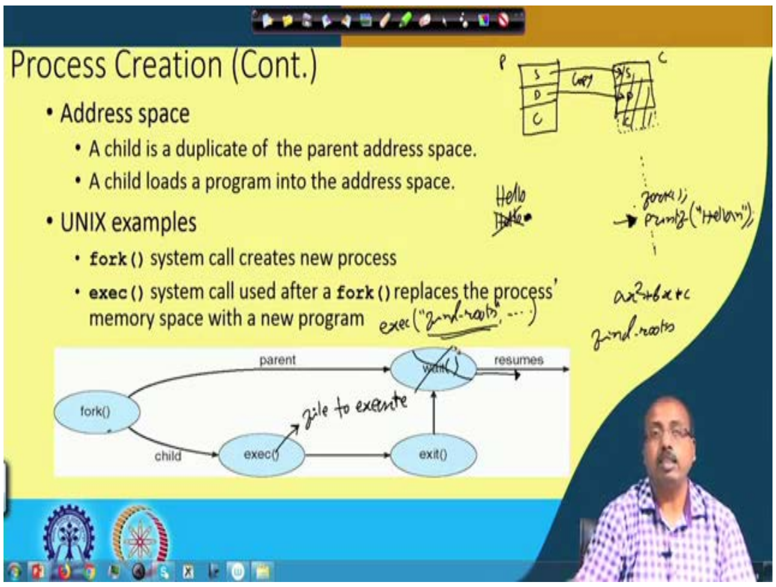
- Address space
  - A child is a duplicate of the parent address space.
  - A child loads a program into the address space.
- UNIX examples
  - `fork()` system call creates new process
  - `exec()` system call used after a `fork()` replaces the process' memory space with a new program

So, first system call that we have in UNIX operating system is the fork system called that can create new children processes. So, a fork creates like one process is there. So, suppose I have got a process P and this process executes a fork instruction like this is the code of P, then somewhere here there is a fork statement; somewhere here I have got a fork statement. So, what this fork does is that it creates two processes was it creates a new process that is the child process. So, this process is created and then this parent process continues.

So, as a result after forking so, we will have two processes. So, from this P another child process C will be created ok. So, that way we have got two processes now, again this child make a further fork create another child C 1 so, that can happen, but; or this process P can also fork another child ok. So, that can happen, but this parent child relationship will be maintained and new processes will be created whenever a fork system call is executed.

So, this child process that has been created. So, it can execute and exec system call to replace the memory space with a new program. So, when in case of UNIX operating system what happens is that, when the fork is executed the new process that is created. So, it will be sharing the code, data and stack segment with the parent process in some sense. But after that so, if the there is a another system called exec by which you can override the say the code, data and stack segments of the process. So, as a result it can execute a new program.

(Refer Slide Time: 12:59)



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

So, I will try to explain this example or this flow in more detail like what has happened is that the parent it has got some code data and stack segments. So, this is the parent process and this parent process has executed a fork statement. So, after for creating a fork so, it this child process that is created. So, it will also have the code data and stack segments. So, out of this, this code and data so, they will be sorry, this is the stake and data they will be copy of this parent process. So, this is a copy and this is also a copy; so, these are copy.

However this code part, code part is exactly same so, we do not copy code because code there is no point copying code because the code is read only segment. So, this is actually the same C that we have. Now we can try to understand that if I write a program in the parent process somewhere suppose I put a fork and after that I put a printif statement printif hello.

So, what will happen is that, when this fork is called. So, this is the situation that is created. So, this parent process is created and the child process is also created. Now, both of them are two processes or two independent processes now and this scheduler will schedule them independently. So, the scheduler when it schedules the parent process after fork statement next statement is printif so, it will print this string hello onto the screen. Then it will do something more as it is not depicted here, but after sometime the parent will either finish or it will be swapped out of CPU because its time slice has expired. So, the child process will come. So, child process will also start executing from this point onwards, because as we said that the parent and child they share all the resources.

So, this program counter value that we have for the parent process is also copied in to the program counter value for the child process. So, child process will start executing from this point. And then, and so this is the same code so it is will also getting the same code yes it will also execute a printif so, it will also print a print a hello. So, this program so, it will be printing this hello twice; once from the parent process, once from the child process.

So, this is happening when this parent and child their code segments are similar. Many a time what happens is that after executing this fork this child; in the child process we put an exec statement. So, this exec system call so, that is for to execute a new some other programs. So, you need so parameters are not specified here; so, here you basically say the file to execute we mention the file to execute.

So, for doing this, so you can name for example, you can create a child process to solve to find the roots of a quadratic equation ax square plus bx plus c. So, in that case I have a program which is finding say suppose name of the program is say find a roots. So, what I will do? In the exec system call I will write something like this, exec I will give the name of the file find roots and I will give other parameters. So, you can look into the manual of UNIX to get what are the parameters to be passed in case of exec system call. So, basically what will happen in this case is, this entire segment that we have for the for the child C so, it will be overwritten by the exec ok.

So, this find roots it will be copied from the disc and it will be overwritten here. So, as a result this parent process and child process they will now execute all together two different code, and in this case the hello will be printed only by the parent not by the child. So, instead the child will say print some messages like enter values of abc, then it will calculate roots and print like a roots are r 1 these r 2 this like that. So, this is one possibility. So, that way that that is how this exec system call comes into help.

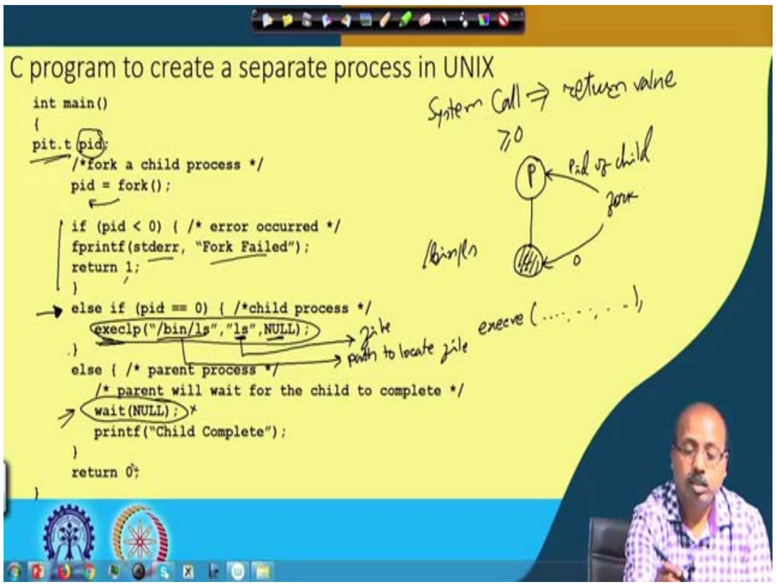
And then after sometime the child process either terminates normally or it executes an exist statement or exist system call by which it terminates. So, when it executes either way so it terminates; so, there maybe two situations like the parent that created the child may be waiting for the child to be over. So, if it makes a wait system call then it will be waiting for the child process to be over.

And the parent process will be blocked at this point till the child process has finished. So, if this is there, then the child; this parent will wait till this entire operation is done by the child process and after that when the child process is over operating system will send an event notification to the parent process that the child is over and then only it will resume its operation.

So, this parent process may be wait or if this wait call is not there, if the parent does not want to waits, then that the parent will not put on wait call after fork. So, as a result this child process the parent process will not wait for the child process and parent process will go into a its execution as it is. So, this is provide the parent will not be blocked by the operating system, it will continue doing whatever statements it has after the fork system call. So, this way we have got this fork and exec system calls in UNIX operating system by which this process creation, execution and execution or different programs can be taken care off.

So, we will be looking into more detailed example. So, this is one program that creates processes in UNIX operating system. So, this is a C program.

(Refer Slide Time: 19:10)



C program to create a separate process in UNIX

```
int main()
{
pit.t pid;
    /*fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
    fprintf(stderr, "Fork Failed");
    return 1;
    }
    else if (pid == 0) { /*child process */
      execlp("/bin/ls","ls",NULL);
    }
    else { /* parent process */
      /* parent will wait for the child to complete */
      wait(NULL);
      printf("Child Complete");
    }
    return 0;
}
```

So, in case of; so, in case of this UNIX operating system so, we have got this pidt. So, that is the data type for this pid variable. So, this is you can look into this system programs the header files to get what is this pid dot t. So, this basically it tells that the pid of a process the process blocks and all those all those information I captured in this pid dot t data type. Now, this; so, first statement that this process is executing is fork.

So, when this fork is executed so, there can be two situation; three situation rather. So, any system call that you have in UNIX so, it there is a return value. So, it has got a return value by which you can check whether the system call was successful or not. Now when this system call is executed then if the call is successful in that case it will return a value which is greater or equal 0, it returns a value which is greater or equal 0.

If the return value is less than 0; that means, there was some problem with creation of the new process may be the process table it is full or something like that. So, that is why this process could not be created. So, in that case it will return a value here which is less than 0. So, pid less than 0 then some error has occurred. So, on the standard error output; so, we say fork failed and we come out of the program.

Now if the call is successful then it returns two values and two values to two different programs. As I said that this parent process is there which executed the fork, but due to the execution of fork the child process is created. Now this fork system call it returns two values to the two processes, it returns the pid of child to parent; so, this one it returns pid of the child process that is created and to the child it returns a 0. So, you see that after this fork is executed so, this; so, assuming that this error has not occurred. So, the both parent and child they come to this point ok.

Now, for the parent process it will see, it will try to check pid value with 0 and since this return value is the pid of child which is non-zero. So, it will be this check will be will not be successful whereas, for the child process the return value is 0 so, this check will be successful. So, child process will execute, then part of this if statement that is this one, and for the parent process what will happen? Parent will get a non zero value in the return value of fork.

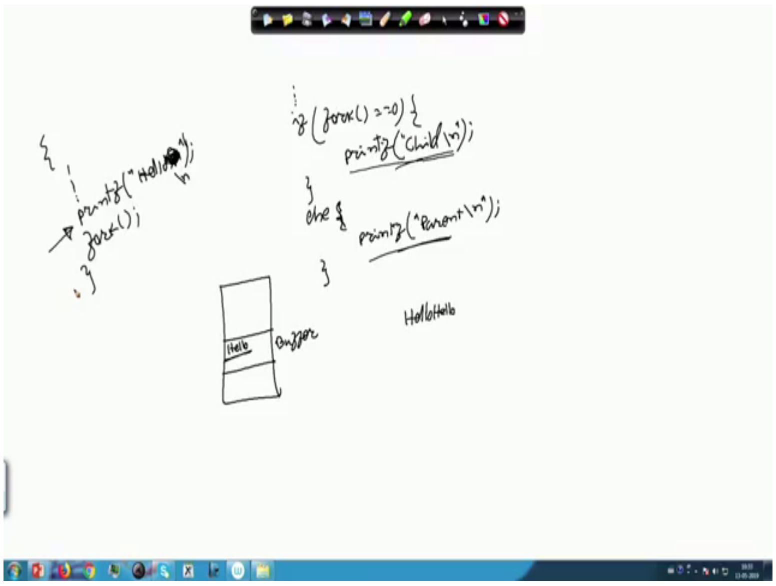
So, this check will be falls. So, it will come to this point and the parent will execute this wait statement. So, it is waiting and this so, wait for child to be over; so, if this wait statement was not there so, parent will immediately come to this statement and it will print the child compete and it will come out.

Whereas, what the child process is doing so, it is making a system call execlp. So, execlp is a variant of exec. So, there are many variants of this exec; so, if the most generalized say version is exec ve ok. So, there are certain parameter c and there are some variants of this exec v and they have been given different names. So, execlp is one such variant so, there what we need to mention so, this is the file that we want to execute. So, this is the file to execute then this one this part the first parameter that we pass is the path to the file; path to locate file. So, what it says is that you look into slash bin slash ls; slash bin directory and there you can find the file ls then the command to be executed in ls and then we are sending a NULL that is we are not interested in sending any more arguments to the ls command.

So, ls command accepts lots of argument. So, which this ls command is basically lists the file that we have in a particular directory. So, it is trying to execute that program and ls has got different type of arguments so, we are not mentioning anything here. So, this is passed to the to the execlp system call; so, this exec lp will be the once the child process is created when it executes with the execlp. So, its code data and stack segments will be overwritten by the content of the file slash bin slash ls, it will be overwritten by the content of the files as slash bin slash ls and then it will be printing this then it will be executing this command.

And after this command is successful, you see this child does not have anything more to do so, it will terminate. So, when it terminates so, this operating system will notify the parent that the child is over, and when the child is over so, it will print that it is complete the child is complete. So, this way we can execute different programs using this execlp system call.

(Refer Slide Time: 25:21)



```
{
  printf("Hello\n");
  fork();
}
```

```
if (fork() == 0) {
    printf("Child\n");
}
else {
    printf("Parent\n");
}
```

So, we can take some more examples like a very simple example that I have already talked about. So, in the parent process so, if I say like if fork equal to; equal to 0. So, with this is the if fork equal to 0 means this is the child process; so, we say this is the child else we can say that this is the parent process.

So this is a very simple piece of code that separates out this parent and child so, if fork equal to 0 there. So, this will print child this print will be executed by the child process it will print child and this print will be executed by the parent process and this will print parent. Now if you take some variants like say suppose I am writing a program, where I write like say printf hello and after that I have got a fork statement and it ends at this point.

So, this program; so, you see that before the child process is created the parent process will print the hello statement and it will after that it will fork, it will create a new child process, but child process does not have anything more to execute. So, it will terminate immediately and the parent will also terminate. Now suppose this slash in character is not there.

So, I do not have this slash in here. As you know that in case of operating systems so, whenever there is something to be printed on to the printer. So, it does not go to the printer directly, but rather it is kept in some memory buffer. So, some buffers space is allocated for every process to act as the buffer for this printer access or file access etcetera. So, when this hello is executed. So, this hello is actually stored into this buffer. So, it does not go to the output directly until and unless you have got a slash in character at the end. So, it is not flashed to the output.

So, it will be remaining in this buffet. So, in this case what will happen when the program terminates then whatever is there in the buffer so, that is actually flashed out. So, what will happen is that, in this case we will see output like this hello because when the parent terminates. So, it will play this hello will be flashed off and then after that this child process will also have a copy of the buffer with it, because as we say that code data and stacks. Data and stacks of their copies of this parent process data and stacks so, that will also be copied. So, this stack buffer will also be flashed out so, as a result you will get hello twice.

Though you have given this hello before for fork, but still because of the fact that this hello buffer is not flashed so, after this fork is executed so, it will be flashed against so, you will get hello twice. So, in this way you can come across different type of phenomena or different types of outputs and try to reason them out by putting fork at different places in the program.