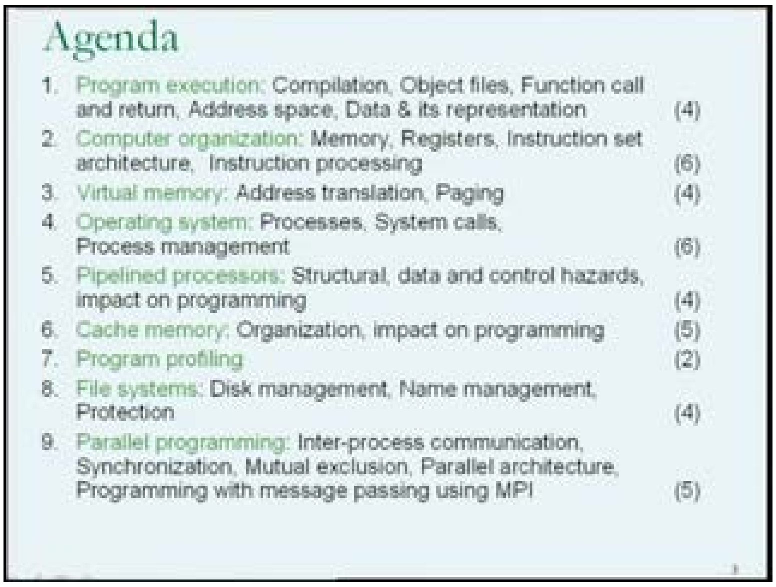
High Performance Computing

Prof. Matthew Jacob
Department of Computer Science and Automation
Indian Institute of Science, Bangalore

Module No. # 03
Lecture No. # 11

Welcome to lecture 11 of the course on, High Performance Computing. Today, we are actually moving into a new set of topics, a new kind of discussion. Until now, as you look at the agenda on the slide, you will notice and until now, we were talking either in general terms about programs or we were talking in specific terms about the hardware and those were the two items in the beginning of the agenda for this course, which are listed one and two.

(Refer Slide Time: 00:41)

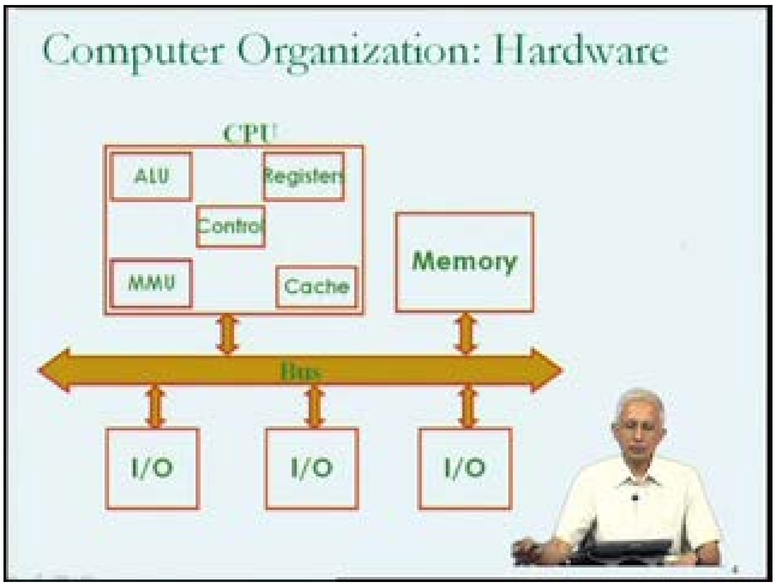


Agenda

1. Program execution: Compilation, Object files, Function call and return, Address space, Data & its representation (4)
2. Computer organization: Memory, Registers, Instruction set architecture, Instruction processing (6)
3. Virtual memory: Address translation, Paging (4)
4. Operating system: Processes, System calls, Process management (6)
5. Pipelined processors: Structural, data and control hazards, impact on programming (4)
6. Cache memory: Organization, impact on programming (5)
7. Program profiling (2)
8. File systems: Disk management, Name management, Protection (4)
9. Parallel programming: Inter-process communication, Synchronization, Mutual exclusion, Parallel architecture, Programming with message passing using MPI (5)

And from today on, for the next 10 lectures or so, we are going to look at some of the software, which is on a system to help in the execution of the programs that we run. So, that is the agenda for the medium term future.

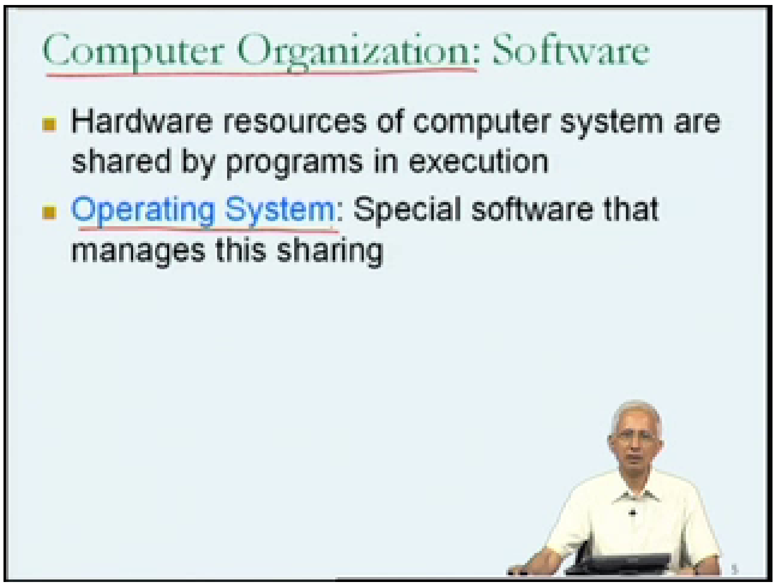
(Refer Slide Time: 00:59)



So, just to recap in the hitherto, we have been looking at the computer at this level and we have pretty much understood the components of each of these building blocks. We know a lot about the main memory; we have seen how important the registers are. We have not actually looked at the design of arithmetic or logical functional units, but that is not really that important to us as people who are primarily interested in learning what happens when our programs execute on a computer, from the perspective of trying to make them more efficient or faster.

In the previous lecture, we looked a little bit at how the control hardware manages the execution of instructions. We talked about the different special purpose registers that might be present and how they might be used by certain logic circuitry to achieve the execution of instructions. We also have an approximate idea about what the memory management unit does, and we understand that without the presence of this cache memory, the amount of time to execute a single instruction might be very large due to the wide speed disparity between the main memory and the processor.

(Refer Slide Time: 02:15)


Computer Organization: Software

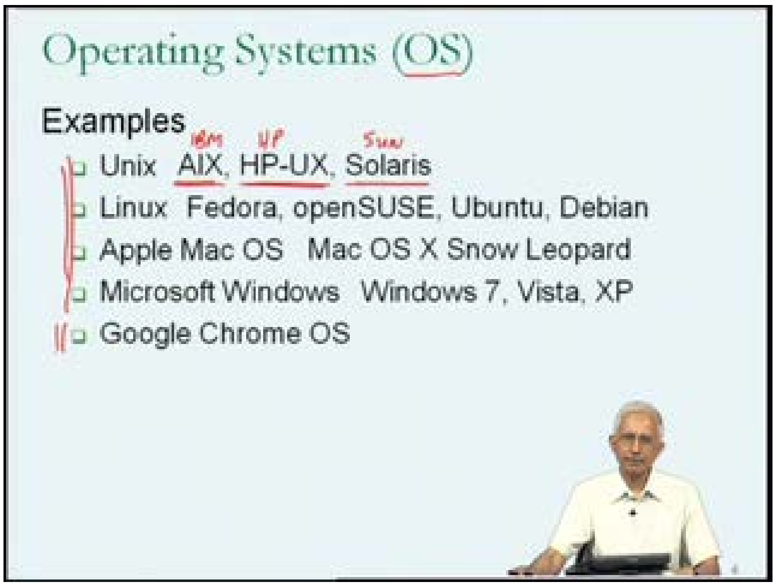
- Hardware resources of computer system are shared by programs in execution
- Operating System: Special software that manages this sharing

So, with this general, but in fact, somewhat specific understanding of how the hardware is organized, we can now move into the software. We can start little bit what happening on the software side and I am going to continue to use the term, Computer Organization, because we are going to look at certain aspects of the software other than the programs that you write. There are, as you may be aware, several components of software, which are sort of permanently available on computer systems. Your programs come and go, but they are certain components of software, which are always there and are integral parts of the computer system.

Hence, I will continue to use the word computer organization, since these pieces of software are integral part of the computer system. Now, in the previous lecture, when I raise the idea that we have an idea what the hardware is like, but they could be more than one program running at a time or concurrently, I use the term concurrently.

So, what this told us is that we must expect that the hardware resources of the computer system that we have seen in the computer organization hardware slide are actually shared by those programs in execution. They could be five programs in execution. So, they must be some software, which is helping in doing this and one of the very special pieces of software that manages or helps in managing the sharing of the hardware resources among the many programs in execution is known as the Operating system. So, we do need to know a good deal about the operating system to understand what happens when our programs run on a computer system.

(Refer Slide Time: 03:41)



Operating Systems (OS)

Examples

- Unix AIX, HP-UX, Solaris
- Linux Fedora, openSUSE, Ubuntu, Debian
- Apple Mac OS Mac OS X Snow Leopard
- Microsoft Windows Windows 7, Vista, XP
- Google Chrome OS

So, let us say its look a little bit at operating systems. So, the operating systems is typically abbreviated as, OS, operating system. And in your experience with using computers, you will have encountered, I am sure many different operating systems or would have come across literature where many different operating systems are mentioned. Today one of the more prominent operating system names that you may come across is, Unix.

Unix is the name of family of operating systems, which has been around for about 40 years now. It is been around for a long time, it has its roots in some design activity at ATNT bell labs, followed up by some design activity at some universities, both in Europe and in the U.S. Today, if you look into the term, Unix we will actually find out that there are many variants of Unix. I am just giving a few examples of variants of Unix from the not too distinct past. So, for example, there is a variant of kind of Unix called AIX and AIX, in fact, was the I B M version of Unix.

It was the version of Unix that you would use if you were running or wanted to run a Unix operating system on hardware that had been built by I B M, the company I B M. Similarly, there was original operating called HP-UX, which as you would imagine was the HP, the Hewlett Packard Operating System. So, on their hardware they would typically run HP-UX operating systems and then there is Solaris which was the Sun operating system.

So, clearly Unix, is not a specific operating system with, the term is used to refer to as a family of operating system and the elements of the family could be hardware specific, as we see in this example. AIX meant for I B M hardware, using I B M processors, HP-UX meant for Hewlett packard systems, using particularly Hewlett packard design processors.

So, Solaris use to run on the Sun hardware, but has been modified so that they can run on any of the Intel processors as well. So, all of these have something in common and that is why there are all called Unix and therefore, there are some standard properties of a Unix Operating System, and there could be some other variants, which are do not comply entirely with those standards, and might be called Unix like in some cases, since they have a general look and feel of Unix.

Now, another term which or name of an operating system that you may have come across is Linux. Linux has become very wide spread; I imagine that many of you have Linux running on your laptop or desktop at home, certainly in your labs at school, I am sure there may be a Linux machines. The property of Linux is that unlike Unix of the upper line like AIX or HP-UX or Solaris, which you typically bought from the company, Linux is actually available through in a form called free software. It is not only as software, but its open source so the source code, the actual C programs, which form the base, the implementation of the Linux kernel or the Linux core operating system are actually freely available, one can down load them.

However, just having the kernel in source from is not very useful to you or to me, since what we want is an operating system, which can run on our, let us say a personal computer, and help in the sharing of the resources of that personal computer, and we do not need to know with the source code, the C programs that are doing that look like.

So, we are more interested in getting some kind of a distribution, which might include the Linux kernel along with other programs of interest to us, such have may be compilers or development environments etcetera. So, you will have possibly come across many distributions of Linux, for example, you may have heard of Fedora or Red Hat Linux OpenSUSE, Ubuntu to Debian. So, these are all just not various of Linux, all of them will have the Linux kernel, there is only one Linux kernel, at any given point in time, of course,, they could be copies of older Linux kernels, but there is one most recent version of Linux, which would be the current Linux kernel and one could get it in any one of the distributions that are listed here.

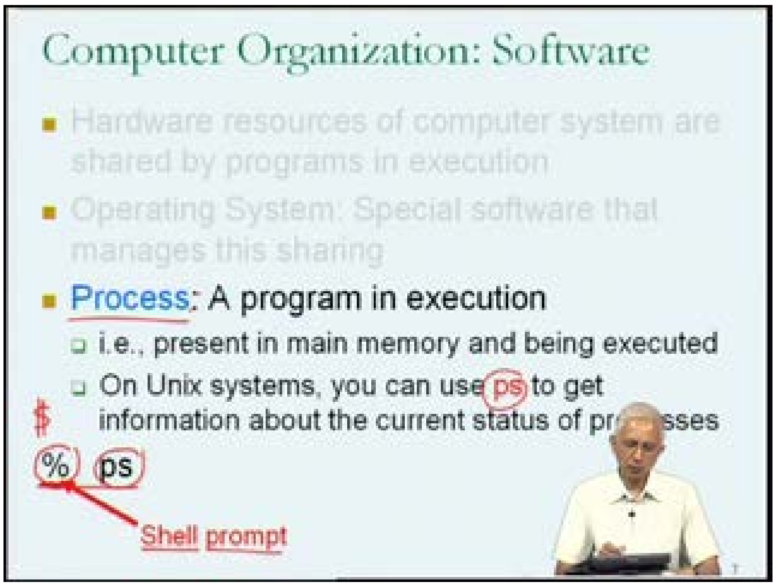
Now, there are others as well. It is the world is not just Unix and Linux. For example, if you have a Macintosh, an Apple computer or I Mac or something like that then you will be aware that the operating system on those on those computer is the Apple Mac OS, which was originally developed on certain Unix like, from Unix like this called Mac, and the current version of the Apple Mac OS, which you have on your own computer it is called the Mac OS X Snow Leopard, previous OS called the O S X Leopard, I believe.

So, that is certainly different from either of the UNIX or the Linux. Once again, this is an operating system that one buys or one gets on a Apple machine, when one purchases the Apple machine and that is not end of the list of operating system examples. If you have a windows machine then you should be aware that the operating system there is what is known as Microsoft Windows, and there have been many variation developments in the Microsoft Windows Operating systems.

Today, you have Windows 7, previously there was Windows Vista, before that Windows XP and align of Windows Operating Systems, prior to this. So, these are all developed and sold by Microsoft and they are all operating systems, quite different from the Unix or Linux or the Apple Mac OS. In more recent time, you may have heard of the Google Chrome Operating System, which and it development by Google and this is primarily intended for web applications. It is not meant to manage hardware, in the sense that the others are. And it is once again based on a Linux like kernel, Linux kernel. I believe.

So, there are many operating systems and it is not our objective to learn something about each of these, what we are actually going to do is to try to talk about operating systems and an abstract UNIX like sense. So, we will talk about things for the perspective of, since all of the examples that we have here for the most part are UNIX like, we will use that perspective in our discussion of operating systems.

(Refer Slide Time: 10:06)



Computer Organization: Software

- Hardware resources of computer system are shared by programs in execution
- Operating System: Special software that manages this sharing
- Process: A program in execution
  - i.e., present in main memory and being executed
  - On Unix systems, you can use ps to get information about the current status of processes

$
% ps
Shell prompt

Now, we are talking about the software side of the core components of the computer system and the operating system seems to be an integral part of that, but there are other concepts that we must be comfortable with, before we can understand what happens when our program executes, and one of the fundamental concept is the concept of a process and basically the process is not a piece of software process, is an abstract entity, and we will think of it as being a program in execution.

Henceforth, rather than referring to my program in execution on a computer system, I could talk about the process, which is the execution of my program on the computer system, and the processes are, at this point, as far as from what I am telling you, an abstract entity, which represents the execution of the program of your program on the computer system. Clearly, it must have an implementation; otherwise it could not cause the execution of instructions. Therefore, it is complete entity, as far as, the software side of the computer system is concerned and we will talk more about that a little bit later.

So, we will use this term Process from now on, instead of talking about a program in execution. In shortly, we will see that this not actually very good, this is not a definition, this is just the term Process does not mean program in execution, but for the moment, we just use it in this sense and get a final understanding as the next three or four lectures.

Now, what it means for a program to be in execution as we have seen is that the program must be present in the main memory of the computer system, and it is in a state where it is instructions are being fetched from main memory, into the processor and executed, based on the hardware understanding, what those instructions mean. So, that is what it means for a program to be in execution, and we seen that there could be more than one program in execution, concurrently. In other words, more than one program present in main memory, and the instructions of these multiple programs are somehow fetched into the processor, and executed and that is something we have to learn more about.

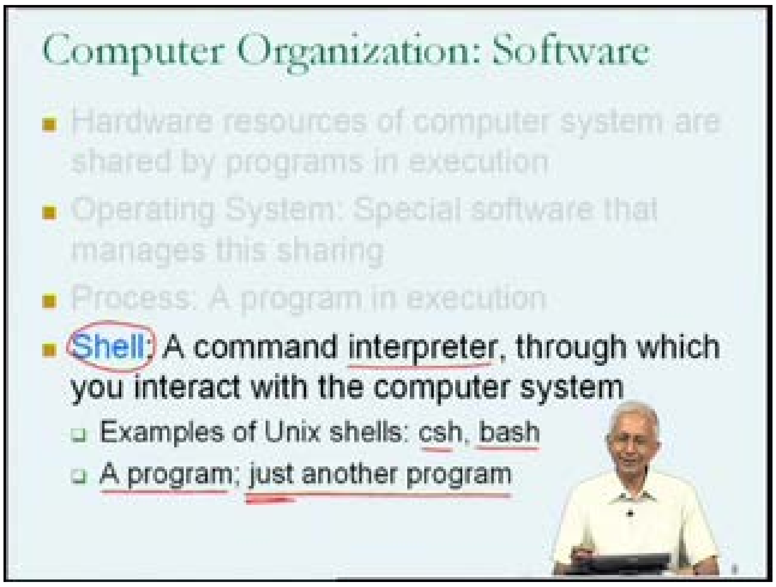
But, we can get some practical understanding of the fact that they could be more than one program, in execution on a computer system, at a time or specifically on a Unix system, because there is a mechanism available, in other words, this thing, which I now call p s, (Refer Slide Time: 12:27) which in fact, stands for process status, which can give you information about the status of the processes, which are currently on the computer system.

So, you could actually use p s, if you are logged on to a Unix computer system, then we could type p s, in as a command and be given, you will expect that in written, you will get lot of information, about the current status of the processes, which are on that computer system, in other words, the programs in execution on that computer system.

Now, in this notation, that I have used here, I have used this before. Remember, that percent sign (Refer Slide Time: 13:09), which I am showing on the left is what I call the Shell prompt. I never actually told you what the Shell prompt was, but the impression that you have is that when you logged into a computer system, in this particular mode, a Unix like computer system, then you will see percent sign or something like that on the screen. In general, that thing which you see on the screen is called the Shell prompt. It is prompting you to type something in a command, in this particular instance, I have typed p s, because that is the command, which I want to execute, I want to get information about the current status of the different processes.

On different Unix Systems, you may find out that the shell prompt is different. In some UNIX systems, shell prompt might be a dollar sign. In others, somebody may have configured is so that it has more information in it, but in even there are some string of characters, which prompts the user the type something in, and that is what I mean by the shell prompt and in my examples I will used the percent as a shell prompt. So, we understand where the word prompt is coming from, because the purpose of that percent appearing on the screen is to prompt the user. In other words, to request the user, to type a command in, we do have to understand what the first term is, what is the shell.

(Refer Slide Time: 14:18)



Computer Organization: Software

- Hardware resources of computer system are shared by programs in execution
- Operating System: Special software that manages this sharing
- Process: A program in execution
- Shell: A command interpreter, through which you interact with the computer system
  - Examples of Unix shells: csh, bash
  - A program; just another program

So, let me tell you that the Shell, the word Shell is used for a command interpreter and it is through a command interpreter or Shell that you interact with the Unix computer system. The command interpreter or the shell, asks you for a command through the shell prompt. You type a command in and then the shell takes the command and interprets it. In other words, it does whatever has to be done to achieve the command that you have typed in. We, primarily, in this mode, think about interacting with the computer system, the hardware and the software through the Shell, which raises the question of what, is the shell. It is possible that the Shell is a special piece of hardware. It is possible that the Shell is just a part of the operating system.

The third possibility is that the shell is just an ordinary program and it is in fact, what we are going to find out. On Unix Systems even or Linux System, you will find out that they are many different shells available. For example, there is a shell called the C Shell. There is another shell called the b a shell or bash, and you could actually choose on Unix System, choose any shell among those, which are provided or which are available on their computer system. So, it is a user choice.

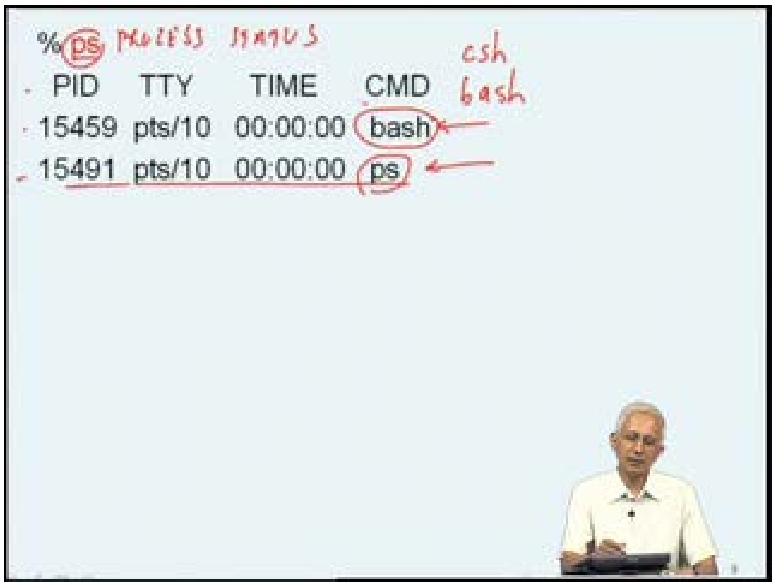
To answer the previous question that I had raised about whether the shell is a piece of hardware or part of the operating system or just a program, the answer is that the shell is a program. It is just another program. The reason that I stressed is just another program is because when if I were to say that the shell is a program that would not demystify the question of whether the shell is an important program like the operating system.

The operating system, too for the most part, is just our program is a program. sorry, But, the shell is just a program in that, it is a just a program like the a dot out, which you generated by writing a program dot c. You could write your own shell, you could write a shell dot c program, which does a command interpretation. You could compile it into a dot out, and then that a dot out, could be your shell. In that sense, shell is just another program, and will understand specifically what is shell does, shortly.

So, the operating system is a very important piece of software. When a program is an execution, it exits in the computer system, as something, which we will call a process. The program itself is a piece of software; the process is the execution of that piece of software. Then we interact with the computer system through a shell command, interpreter in the case of Unix systems.

The two examples of shells are CSH and BASH. Now, on the previous slide, I had mentioned that in response to the shell prompt, if I had typed the command P S, then I could get information about the different processes and execution on a computer system. So, let us just try that.

(Refer Slide Time: 17:10)



```
% ps
PID    TTY     TIME     CMD
15459  pts/10  00:00:00 bash
15491  pts/10  00:00:00 ps
```

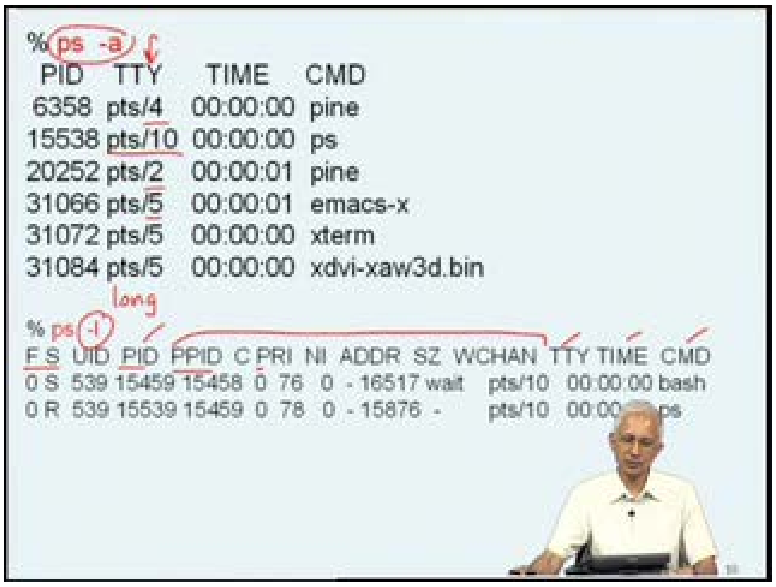
Let us suppose that I am on I am logged into a Unix System and I typed p s, and then I hit return. I will get an output from the system that might look something like this. So, there is a lot of information, which is output and remember that the purpose of the command, was to give me information, about the status of processes on the system. So, since there are two lines of output that I receive, I suspect that I have two programs, at least two programs in execution, and that is what I see.

I notice that the two lines and one of them strangely enough is an indication that the command, which I had typed in, is one of the programs in execution, but that should not be too surprising, because when I typed the p s command if p s is the name of a program then the program has to execute, and while it is executing, it is causing these lines or output to be printed out, and since while it is executing, it must be, it is a program in execution or a process, they will be a line in the p s output corresponding to itself.

The other line in the p s output is also interesting, because we have come across this name bash before on the previous slide. In fact, where I had mention that I give you two examples of command interpreters in UNIX, one was C S H, and the other was bash, and we see that bash itself is listed. So, there is a process, which is the execution of the command bash. So, bash too is just a program and a dot and a dot out, which has been renamed as bash, and it is executed and it is the execution of this program, which is creating the prompt on the screen, reading in whatever I type in and causing the different instructions in the bash program are what are causing the execution of the p s to happen, p s program to happen.

So, we clearly see that bash is not that complicated and shortly we will see how could write your own simple shell. Now, from this it looks like the p s program is capable of giving us information about the important processes on the system, and in the system at which I typed this command there were these two important processes running, but this is not the whole story. If you actually read a more about p s, you will find out that p s can be executed with different command options.

(Refer Slide Time: 19:28)



```
% ps -a
  PID  TTY      TIME     CMD
 6358  pts/4   00:00:00  pine
15538  pts/10  00:00:00  ps
20252  pts/2   00:00:01  pine
31066  pts/5   00:00:01  emacs-x
31072  pts/5   00:00:00  xterm
31084  pts/5   00:00:00  xdvi-xaw3d.bin

% ps -l
F S  UID  PID   PPID  C PRI  NI  ADDR  SZ   WCHAN  TTY    TIME     CMD
0 S  539 15459 15458  0  76   0  -  16517 wait    pts/10  00:00:00 bash
0 R  539 15539 15459  0  78   0  -  15876 -       pts/10  00:00    ps
```

For example, if I type p s with the command line option minus a, this is a specific request for information about all the processes, which are executing on the computer system. So, the previous listing was clearly not just, not of all the processes, but of some subset of the process. So, if I now type p s minus a, what I get is a larger listing. So, this is the same system on which I have got two lines by just typing p s by typing p s minus a, I get an output from this particular program called p s, which has six lines in its output.

Clearly, there are more processes, there more programs in execution on the system than just B A S H and p s at this point in time. And again, if you look at the output, you notice that one of the columns, in this column output of p s is the TTY (Refer Slide Time: 20:16) and it was present in the first also. Previously, when I type p s if I looked at the

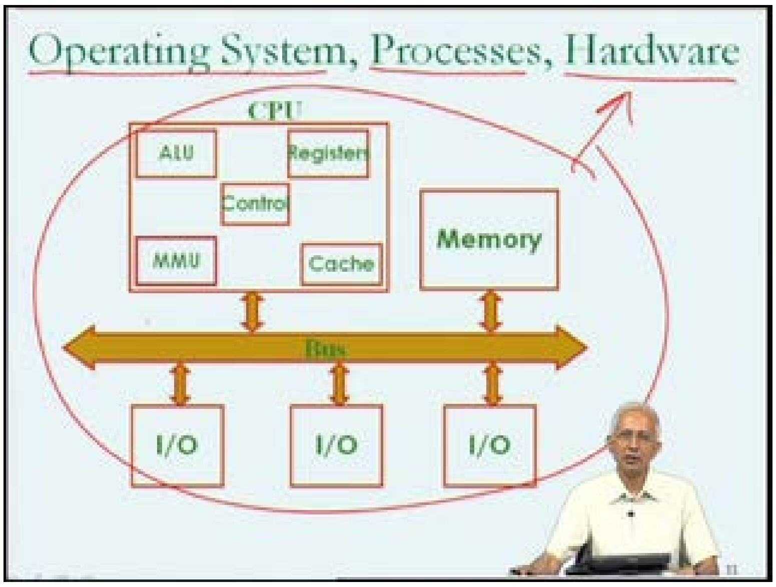
TTY column, I notice that you could you can read TTY as terminal. TTY stands for terminal, essentially saying that from the terminal number ten, in some sense, these two commands had been executed. What I am getting in the p s minus a, listing is information about what is happening from other terminals as well, considerably other user something of that kind.

So, looks like there are at least three other terminals active, and some activity being initiated from each of those terminals, programs in execution from those terminals as well. So, from p s minus a, I get more information, p s seem to be given me information just about the particular terminal, at which I was sitting, or the particular processes, information about the processes in connection with me, whereas, p s minus a, is giving a border prospective of what is happening one that computer system, programs in execution due to possibly other users and there is another possibility, I could actually type in p s minus l and this is asking for a long listing, which will give a lot more information.

For example, when I type p s minus l, I get information about a long listing about the processes associated with me, and there are many more fields. In addition, in the previous listing, I had the P I D, and I had the TTY, and I had the time and I had the command, but none of these fields, which now give me a lot more information about the status of the process, of the different processes.

And we will you see more about what these fields mean as we understand more about what it means to be a process what happens when a program is in execution on Unix like system, as a process.

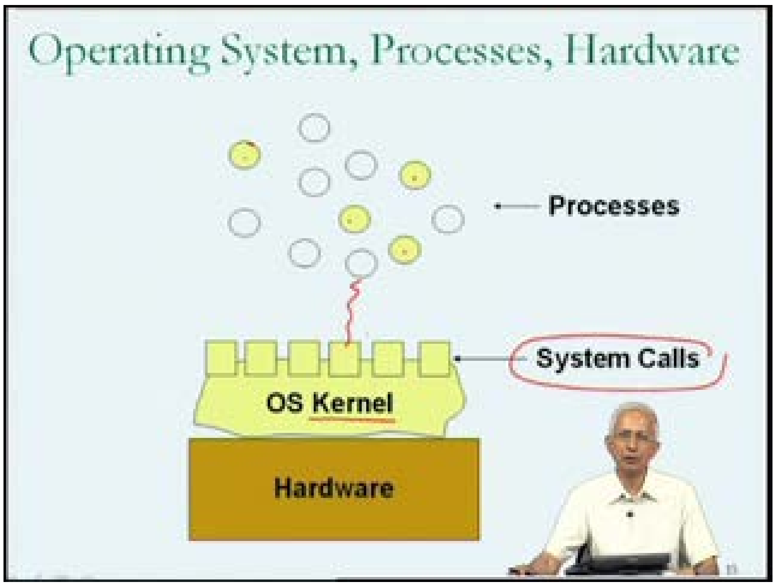
(Refer Slide Time: 22:08)



Now, moving ahead, we have seen that from the perspective of the hardware, this is what the computer system organization looks like. Our objective is to understand the relationship between the operating system, which is the most important piece of software in some sense, the processes, which are all the programs and execution on the computer system and the hardware. So, the diagram that we have on the screen is the hardware and we want to understand the relationship between this and the rest of the important components, the rest of the software organization of the computer system.

So, I am going to shrink this particular box down in size, is no longer as important. We do not need to know the details; we do not need to discuss the details of the individual components of the hardware any more.

(Refer Slide Time: 22:53)



So, I will shrink it down in size to about this big and least the rest of the screen for our discussion about the organization the software side of the organization of the computer system, and since we are not particularly interested in the building blocks of the hardware any more, we know something about it, I will just abstract it out as a box labeled hardware. So, remember, when you see the box labeled hardware, it has that CPU, memory, hardware, bus, I/O devices and so on.

So, what else is there in the computer system? Now, we understood that the operating system is the piece of software, which manages the sharing of the resources, including the hardware resources of the computer system. So, we expect that they must be some and we also understood that they are many programs in execution on the hardware. So, they will be, I will represent the different programs in execution of the different processes by these circles, which you see up towards the top, that different color circles, I will mention why, shortly.

So, this particular example they seem to be about 11 processes sharing this particular piece of hardware, 11 programs in execution. Now, we saw that the actual sharing of the hardware is sources were going to be managed by the operating system. So, the operating system must enter this picture. I will show a box labeled operating system over here. The reason that I show it between the processes is in the hardware is because the hardware resources, the sharing of the hardware resources among the processes is managed by the operating system.

So, in some sense, the operating system is an intermediary between the processes, programs in execution and the hardware. Therefore, I show the operating system in between the processes and the hardware. Now, whenever one of the processes wants to do something with specific part of the hardware, for example, let us suppose one of the processes wants to read from a file, the file is one of the I/O; the file is possibly associated with one of the I/O devices of the hardware. It is somewhere down here. So, if one of the processes wants to do something on a file, it must clearly get help from the operating system to do that.

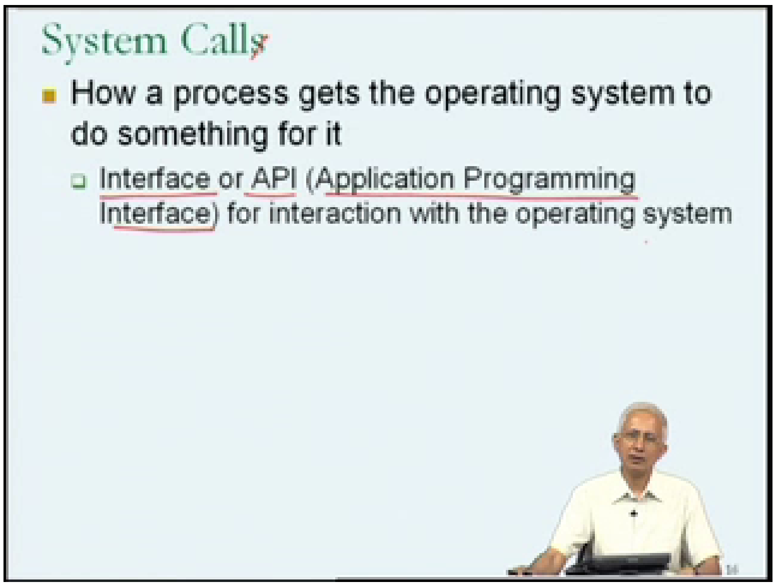
And the way the operating system is structured is that there is a collection of things called System calls, which are worked, the processes are going to call in order to get the functionality required, as far as, the shared hardware resources are concerned. Therefore, the process which wanted to do something to a file, would call the appropriate system call and through that you would actually be able to access to file do whatever you wanted to do.

So, the system calls an important aspect of the operating system. Now, instead of referring to the other odd shaped box as operating system any more, I am going to refer to it as the operating system kernel, since it is not the entire operating system. The system calls are also, in some sense, part of the operating system. The operating system kernel, as a name suggests, is the central, integral, important, core, content of the operating system. Some of the functionality of the operating system may be implemented elsewhere. For example, some of these processes that you see over here are yellow operating systems and would have noticed that the color yellow, in this slide, is being used for operating system components.

So, what is being suggested is that some of the processes, in other words some of the programs in execution, are actually operating system processes. They are doing something for on behalf of as part of the operating system. Therefore, the operating system should not be viewed as a single program or a single piece of software but as many pieces of software. There is a core component, the core or kernel, part of the operating system, which is the OS kernel. They are the system calls through which functionalities made available to the processes. There are also some processes, some programs in execution, which are really operating system related and that is why they are all colored.

The other white processes are just programs in execution of ordinary users like you or me, ordinary programs in execution on ordinary processes. So, the first thing to note is that the system calls are an important thing for us to know about. Because it is through the system calls, a process of mine, in other words, the program of mine in execution, will be able to get some kind of functionality out of the operating system.

(Refer Slide Time: 27:17)



System Calls

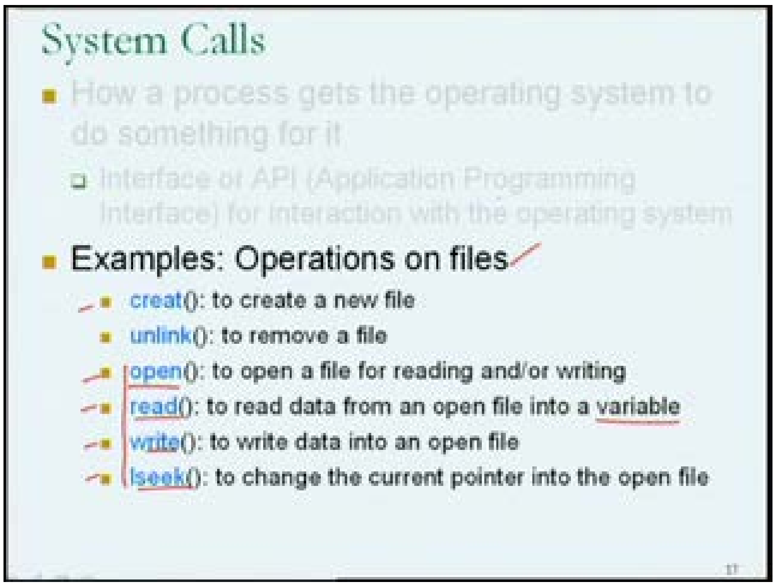
- How a process gets the operating system to do something for it
  - Interface or API (Application Programming Interface) for interaction with the operating system

So, we do need to know a good bit about system calls, and will spend some time doing just that. So, as I said the system calls or a system call is how a process gets the operating system to do something for it and so you get functionalities of the operating system or the operating system kernel by making a system call, and in that sense, you could view the collection of system calls as an interface for interaction with the operating system. The only way that process can interact with the operating system to get functionality is using the system calls and so there is this collections small number of system calls in a typical Linux or Unix scenario, the number of System calls is going to be a few hundred two or three hundred, I forget exactly how much.

So, in that sense, I refer to it as an Interface. It is a collection of functions in some sense, one could call, it is a specification of the mechanism for interaction between the process and the operating system, the OS kernel. Often in reference to software systems, people will use the term A P I, which expands into Application Programming Interface as to indicate that it is a specification; A P I is the specification of how piece of software can interact with another piece of software.

In this particular case, system calls are a specification. They tell you exactly what, how exactly, what information you must provide and with what details, in order to interact with the operating system, in that sense, it is an Interface. So, they are going to be many different system calls for the different kinds of functionality that the process may need to get from the operating system.

(Refer Slide Time: 29:02)



System Calls

- How a process gets the operating system to do something for it
  - Interface or API (Application Programming Interface) for interaction with the operating system
- Examples: Operations on files
  - creat(): to create a new file
  - unlink(): to remove a file
  - open(): to open a file for reading and/or writing
  - read(): to read data from an open file into a variable
  - write(): to write data into an open file
  - lseek(): to change the current pointer into the open file

So, let us look at some of the system calls. So, in some unix System, you will find out that the operations on files, are provided or functionality, as far as operations on files, are provided through system calls. I believe you all are aware with files, for example, when you have a program, which needs to maintain data, well beyond lifetime of the program itself, and then the only option is to store the data in a file. It is basically, a mechanism through which information can be saved for long durations of time. Remember, when your program executes, it may have variables in memory, but they seize to exist when the program finishes execution.

The largest lifetime of a piece of data was the execution time of a program. Therefore, if you want data to exist beyond the lifetime of a program, one basically has to be using files, and many of you will have written programs, they deal with files, so there is a permanent form of storage. One can, read write various other operations on files, the list of operations, which I am going to put up next, will not be too surprising to many of you and except that they are system calls for doing these operations.

So, first there is the system call using which you can create a new file. System call in many Unix Systems call is c r e a t so you can be used to create a new file. The new file once created, will be empty, one could, and one may want to add information into the file. The protocol that is used in operating on a file is in a Unix like systems, is that one must first formally open the file. So, there is a system call, through which one can open the file, and by opening the file, one has essentially saying, making it possible, for subsequent reading or writing of the file to take place.

Now, I had listed system call before that called unlink, and unlink is the system call, which is provided, when one decides to remove a file, so after when you are sure that the files no longer going to be necessary, for other programs, or this particular program, one could use unlink to possibly remove the file. So, as I said, the protocol for using files is that one must, first for an existing file, one must first open the file using open and one can subsequently read from the file. So, there is a read system call, using which one can read data from an open file into a variable. Remember, that read is going to be used essentially, by a process and objective of getting data from a file is going to be to operate on it, within a program, a program in execution.

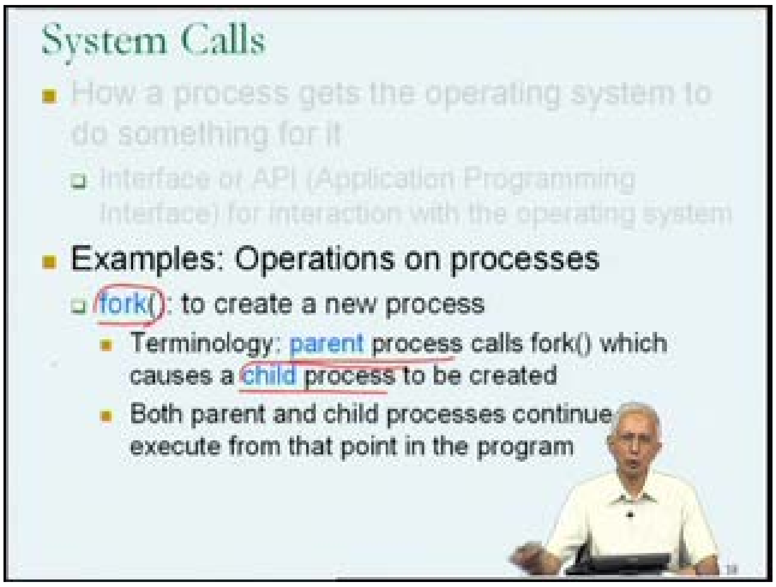
Therefore, it must be available in a variable. So, read will read data from a file into memory locations, main memory locations, which are what we refer to by a variable. Similarly, if one wants to add data into a file, one can do so you using the write system call to write data into an already opened file. So, one could add additional information into a file using write or one could over write information in the file using write.

There is a system call relating to changing the current pointer into the open file. Note that when I am reading from a file, I may want to read from the end of the file, in which case I could use lseek to change the current pointer into the open file, and then read from the end of the file, and with this, there are other possible system calls related files, what this is a representative set.

Now, many of you when you wrote your C programs you would have used open, read, write and lseek and possibly create. You should be aware that the function which you call to do this within the Cc program was not the system call directly, but essentially a function inside a file I/O library of some kind and that function would actually make the sstem call on your behalf.

So, even though you did use functions called open, read, write and sleek, you are not directly calling or making the system call, but for calling a function, which was going to make a system call on your behalf; is just to clarify that you have used these names before, but you have not actually used the system calls directly from your C program. So, we have seen that there are many kinds of functionality, which the operating system can provide; relating to requirements that program in execution may have, relating to files operations on files.

(Refer Slide Time: 33:29)



System Calls

- How a process gets the operating system to do something for it
  - Interface or API (Application Programming Interface) for interaction with the operating system
- Examples: Operations on processes
  - fork(): to create a new process
    - Terminology: parent process calls fork() which causes a child process to be created
    - Both parent and child processes continue execute from that point in the program

Now, they could also be operations a functionality that a program an execution will need to operate on processes. This is an interesting idea. Remember, the system call is going to be calling by a process, and a process may have to operate on itself or on other processes, so this is a new kind of a concept, possibly in other words, not if you have written programs which use these system calls before, but by looking of the first example, we understand why this might be useful.

The first system call that I have listed is the system called fork and it can be used to create a new process. So, what this essentially means is I could write a C program, within the C program, I could use the fork system call and the effect is going to be to create a new process and in fact, I could have a program, which has multiple calls to fork. I could write a program which calls fork five times and the net effect is going to be the that program is going to execute as one, its original one process, which was its original the process, which came into existence, when I ask that program to be executed, and then five additional processes is due to the five forks that may have taken place.

So, the program were ultimately run possibly a six processes, and this is the reason that I mentioned earlier that it is not entirely fare to say that a process is program in execution, because the particular example that I just described, the six processes, which represent the program in execution, no one of the processes fully represents the program in execution.

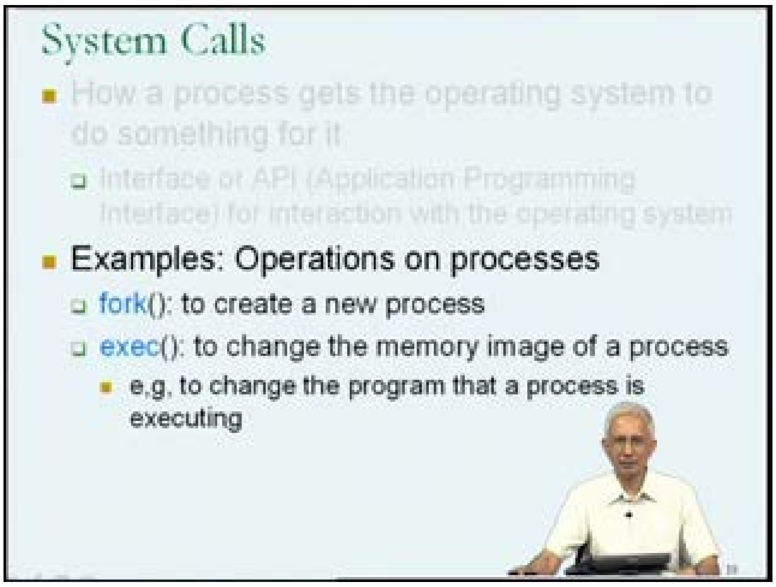
So, fork is clearly going to be interesting. Now, the terminology which we are in reference to fork is remember that when if there is a process, which calls fork, the net effect is going to be there a new process is going to come into existence, which means that I will then have two processes, representing that program in execution, and I do therefore, need to have some terminology to distinguish between them. So, there was standard terminology used is to use the term, parent process to refer to the process they did or called fork, child process for the process that came into existence as the result of the execution of fork.

So, in this particular example, the parent process calls fork, which is a request to the operating system to create a new process and the new process, which comes into existence is the child process. So, a single parent process can have multiple child processes, any child process will have only one parent process, which is the process that caused it to come into existence. So, fork is a important system call

Now, the one of things which you will learn about fork is, after the fork is done, there are two processes in execution, as far as my program is concerned, and we are now, if you look at the next point, which I have you put, both the parent and the child processes continue to execute from that point in the program, and what I mean by that point in the program, is the call to fork, and we look at this in the next slide, the ramifications of this idea, but this is the bare essentials, as far as, fork is concerned.

Fork is a system call that can be used to create a new process, and the effect is a child process comes into existence and both the parent and the child continue execution from the point of the return from the fork the call to fork.

(Refer Slide Time: 36:51)

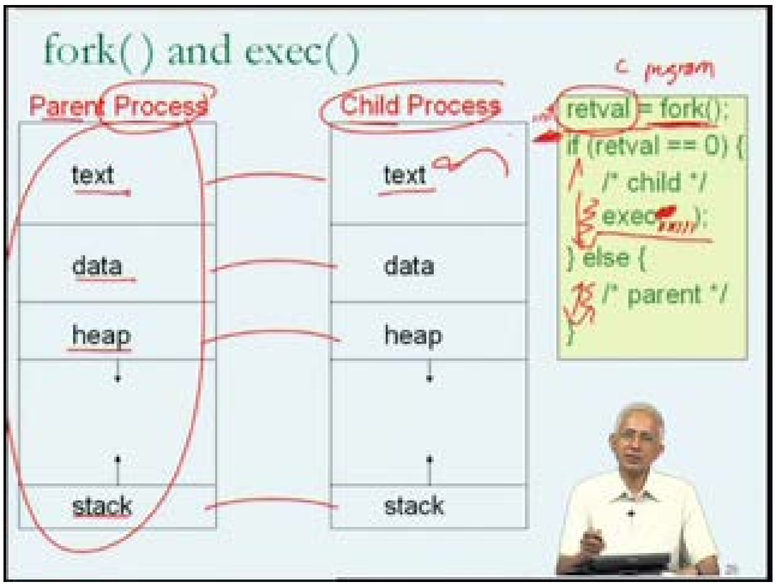


System Calls

- How a process gets the operating system to do something for it
  - Interface or API (Application Programming Interface) for interaction with the operating system
- Examples: Operations on processes
  - fork(): to create a new process
  - exec(): to change the memory image of a process
    - e.g, to change the program that a process is executing

Now, another important system call related to processes is the exec system call. In brief, the description is that exec is used to change the memory image of a process. Now, this might be useful for example, to change the program that a process is executing, and we will see how this works, shortly. But, this is one of the possible uses of the exec system call to change the program that a process is executing, an odd concept.

(Refer Slide Time: 37:21)



fork() and exec()

```
retval = fork();
if (retval == 0) {
    /* child */
    exec(...);
} else {
    /* parent */
}
```

Now, let us think about this. Let us put fork and exec together in this one example. Let suppose that I have a parent process in execution. So, representing that I am sorry I have a program in execution and it is represented by a process, which I will, for the moment referred to as the parent process, since this parent process is shortly going to do a fork.

Now, this process is represented in memory, by things in memory that relate to it. For example, we saw that associated with any program in execution, there is its program which we call the text. There is its statically allocated data, its heap allocated data, and its stack allocated data, and both the stack and the heap can grow and shrink as the program executes. The data is a fix size instead a statically allocated and the text, the instructions of the program typically do not change when the program is an execution.

So, when I talk about the memory image of a process, this is what I am referring to. This is the memory image of the parent process. So, when the parent process is an execution, the parent process could do a fork, and when a process does a fork, as we saw, the effect is that a new process comes into existence. This new process is called the child process, it turns out that the child process starts off, identical to the parent process, in all except one or two subtle points.

We just learn from the previous slide that after the fork is done, I have two processes, the parent and the child, and both continue execution from the point at which the fork returns. So, let us just look at a code example. So, at this point we understood from what I have told you the child process starts off with the same text as the parent, the same data is the parent, the same heap as the parent and the same stack as the parent. It is basically identical to the parent as far as this memory image is concerned.

Now, how do I actually set this up? Let us look at a C program, in which I am calling fork and exec. So, the yellow object on the right is some extract from my C program. So, the way that I call fork is by calling fork as a function in this way. Once again, when I use fork in this form inside a C program, it is conceivable that I am not actually making the system call directly, but I am calling a function call fork, which will make the system call on my behalf.

Now, if you read details about fork, you will find out that fork has a return value. Therefore, when I call for fork, I call it in with an assignment and I remember the return value from fork in the variable call retval; retval is an integer, so the integer variable retval. Now, what follows from what I just described, you understand that both the parent process and the child process, are going to continue execution, from this point in the program. In other words, the point at which return, has been transferred back to my program from the fork system call. So, both the parent and the child are going to be executing at this point, in the program, with basically the same, practically the same memory image.

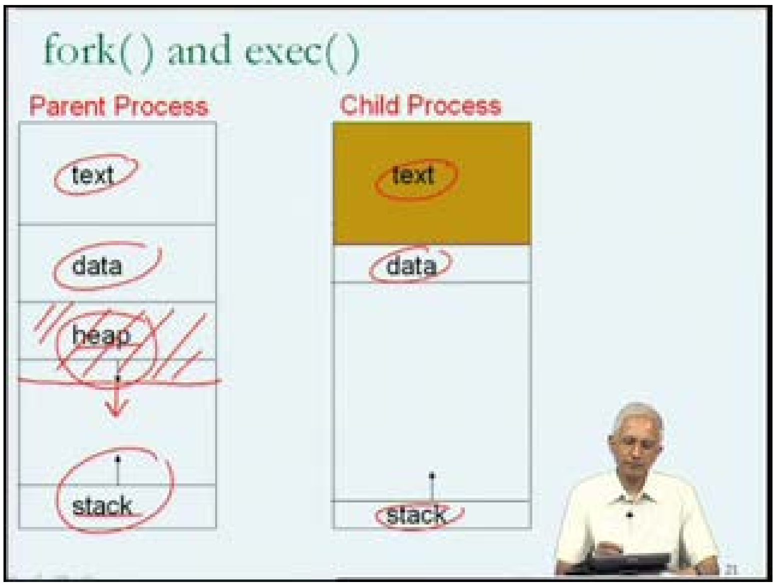
They have, however, do differ in one or two certain ways, and one of the subtle ways in they which they return are in the variable retval. So, if you look at the details of the fork system call, you will notice that the parent and the child will differ, in their value, as far as the variable return val is concerned. Therefore, after return from the function from the system call, if my program is written, so that the first thing, it does is it checks retval, the value return by the fork system call, then I can actually distinguish between the child, which will have a retval of 0, and the parent which will have a retval which is not 0.

Therefore, I can cause the parent and the child start doing different things. It would not, it is not always useful to have the parent and the child returning and both doing exactly the same thing, because it may be useful from me to divide the functionality of the rest of my program, across the parent and the child, rather than it causing both of them to do exactly the same thing. I could therefore, achieve it by writing my program to have different operations for the parent and the child, after return from the fork system call.

One thing, I could for example, do is to call the child to do something substantially different, by actually making a call to exec. Remember, that the exec system call is used to change the memory image of the process, which essentially calls it. In this case, the child process is calling exec the parent process is not and therefore, what is going to happen is that the memory image of the child process can be cause to change.

Now, I am not showing you the parameters to exec, but as you to imagine, one of the parameters to exec is going to be the identity of a program, which could be executed. Then what will in effect happen is that the new program, which I have listed, which I mentioned over here, is going to become the program that the child process is executing.

(Refer Slide Time: 42:22)



So, that is going to cause the text, and the data, and the stack of the child process to change. So, you notice that after the exec, the text of the parent, and the text of the child are completely different, because the child process is now running a different program and therefore, it is statically allocated data is different, and it is stack is going to be different, and it would not have the same heap as the parent process any more.

Therefore, the child and the parent processes are not different from each other and that was achieve by this call to exec and so we see that both fork and exec are important operations on processes. Fork is the operation on processes and that you can be used by one process to create another process and that is why I might included it in a program that I write.

So, that when the program is in execution, the process which is that program in execution will fork and have a child process, and exec is useful for a process to modify its memory image, exec actually has a few area there are few it is a family of calls, and you need to read little about this before using it safely, so much for fork and exec. These are, as I said examples of operations on processes, we are talking about system calls. These are two examples of the system calls that are provided by Unix like operating systems for operating on processes. I am going to mention I think just one or two more.

There is a system call called exit and this can be used by a process to gracefully terminate and what it mean to gracefully terminate is that the process, which calls exit will cleanly terminate. There will not be any uncertainties about the state of it or its computers or the computer system as whole, when a process terminates, if it is terminating using exit, and even if you do not explicitly call exit or cause a rapper function call, even if you write programs, we should do not explicitly contain exit, inside the code, inside the program, you can understand that it might be implicitly called for you at the end of the main function or at the point at which your program normally terminates.

This is something that the compiler could quietly easily insert into the a dot out, in other words call to the system called exit, in order to cause graceful termination of your program. One other system call, I will mention is the wait system call, and the wait system call relates to the relationship between a parent process and it is child. If the situation is that you want the parent process to do nothing, but maintain its current state, in other words, to sleep, until the child terminates, then you could use the wait system call within the parent for that purpose.

So, this allows one to call some kind of synchronization between the activity of the parent process and the activity of the child process. So, the parent process, after while excluding the wait, in executing the wait system call, is essentially saying that it would like to stay as it is, until it is particular child process has completed, whatever it was suppose to do, whatever it was created to do.

So, there are several other instances of system calls for operations on processes, but this representative set of four, will give us some idea of that the nature of the functionality. Finally, I am just going to talk about the possibility of the being system calls for operations on memory. You will remember that until now, we talked about main memory and we know that there is text, data, stack and heap. We know that they could be multiple programs in execution at the same time, in which case, they could be multiple text, data, stacks and heaps, in other words, multiple memory images, one for each process in the memory, at the same time.

But, when I talk about operations on memory, I am talking about something relating to specific operations that a single process might request of the operating system. I will give you one example over system call in this area, and that this is the system call known as, s b r k or s break or set break. And from our perspective to understand what a system call like set break might be useful for, let me just mention that it might be useful if to me, if I was implementing a memory allocation library, a dynamic memory allocation library.

Now, when we talked about the memory image, (Refer Slide Time: 46:47), we talked about the heap and the stack, as both growing. So, the heap may have to grow, as more and more, dynamic variables are allocated, and it might shrink, as more and more dynamic variables are freed. The stack has to grow, as more and more function calls are made and it has to shrink as the function returns are executed. We know how the growth and the shrinkage of the stack happen. That happens explicitly, because of the stack operations that are generated by the compiler in connection with function calls and returns, but I did not say anything about the growth and shrinkage of the heap.

Now, as it happens, the growth and shrinkage of the heap, is managed by the memory allocation library, and not in most cases, in along the lines of how the stack growth and shrinkage are managed. Rather, every once in a while, the memory allocation library will request the operating system to increase the size of its allocation. In other words let me just go back to the slide where we had that memory image (no sound between: 47:55 – 48:03)

In other words, If there is a need for growth of the heap, then the memory allocation routine would call s break and the result of s break, would be that they would be in increase in size of the heap.

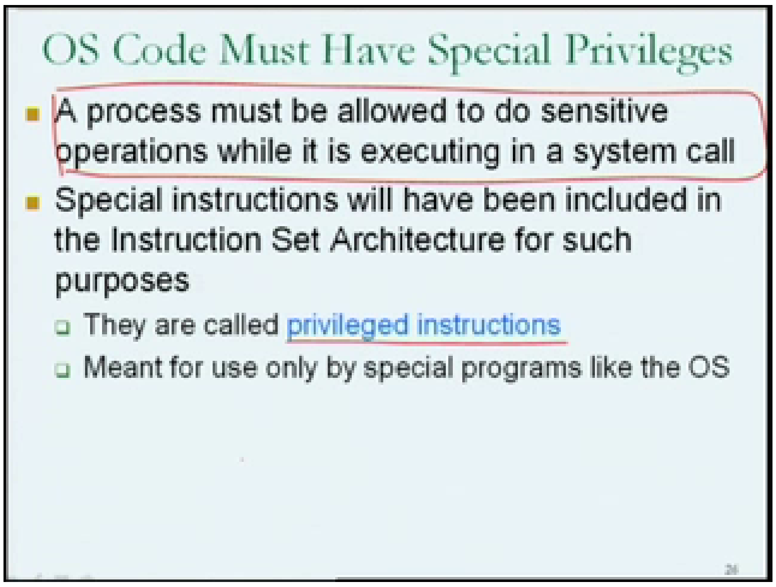
The heap is now be this big, rather than, how big it was before. So, it has something to do increasing the size of data associated with this region of memory, which is why for the first time is showing the heap. Until now, whenever I showed text, data, stack and heap, I was showing you text, data, stack growing downwards, in other words, to higher, I mean stack going to higher and higher memory addresses, and heap growing to lower and lower memory addresses. This is the first time where I am showing you heap growing to higher and higher memory addresses, and stack growing to lower and lower memory addresses, and it is primarily for this purpose that is conceivable that the heap growth is dictated or manage by calls to s break.

S break is a system call, which is a memory related system call and could be used by memory allocation library to increase the size of the heap. There could be again other, but this is the one, which we can sort of understand at this point in time. In short then there are many different system calls, which are basically inter-phase mechanisms, which are available for use by any process, for getting functionality out of the operating system and you could write programs as we saw using these system calls.

We saw a program, which use fork and exec. We have seen that the memory allocation library might uses, s break and so on. So, we see many examples of system calls. Now, one key observation, which I will make at this point, is that from our diagram of the computer organization call in software, we understood that the system calls were in yellow.

The code of the system call, the instructions associate with the system calls should be viewed as being part of the operating system. Therefore, when a process calls the system caller, executes in a system call, bear in mind that it is actually executing operating system code. The diagram that we had was of that yellow operating system blob with those yellow system call rectangles sticking out at the top. So, when a process calls the system call it is executing that the yellow code, inside that yellow system call block, it is executing operating system code. So, when this process calls the system call, it actually starts executing the instructions in that system call that is something we have to bear in mind.

(Refer Slide Time: 50:37)



OS Code Must Have Special Privileges

- A process must be allowed to do sensitive operations while it is executing in a system call
- Special instructions will have been included in the Instruction Set Architecture for such purposes
  - They are called privileged instructions
  - Meant for use only by special programs like the OS

Now, from the time, I introduce the term operating system, I have in careful to say that the operating system is a very special program. It is not like a ordinary program, like a program that you and I could write, because the operating system, does very special things. It inter-phases between the processes and the hardware, it manages the sharing of the hardware among the different processes and therefore, it is it is capable of controlling managing, manipulating the hardware. Therefore, it is special and in order to do the sharing of the hardware resources, the operating system code must have special privileges. It must be able to do things that ordinary programs that you and I write cannot do.

So, when a process execute in a system call, it is temporarily very special. The process is executing the yellow instructions inside the system call and the yellow instructions inside the system call may be doing very special operations, very privilege operations, since they are part of the operating system. Therefore, when a process of yours, when a program that you write a program, which use call fork, when your process is executing fork it is actually executing those yellow, very privilege instructions.

So, a process must be allowed to do sensitive operations when it is executing in a System call and therefore, the entry into system call is a very critical juncture in the execution, during the execution of your program. When your program enters the system call, starts executing a system call, certainly it is doing things, which might be very privileged, and therefore, this has to be control very carefully, and therefore, in any instruction set architecture, special instructions will be included for this transition, the transition between your process doing ordinary things which entirely entitled to do in your process actually entering a system call, and therefore, certainly having to do very sensitive things.

Now, in the operating system, the instructions must be provided for it to do the privileged operations, and these on are call privilege instructions and we have not seen any of the privilege instructions of the MIPS one instructions set, because we will not see them in the programs that we write, and may not going to write, and since we are not going to write in operating system, but if I actually look that operating system code, like the Linux binaries or something like that, you may actually see the privilege instructions. So, the privilege instructions are included in instruction set for use by special programs like the O S, operating system kernel, will use many of the privilege instructions.

Some of the operating system processes, the yellow circles at the top of a diagram, may also be able, may also have to use some of the privilege instructions to do the special things that they do, and therefore, we do need to understand the transition, which I talked about over here, much more carefully, in much more detail, before we can see that the operating system is in fact, this very special entity, which is being protected and why this protection may have to takes place.

So, I will stop here for today, reminding you that in today's lecture number 11, we have started looking at the organization of the computer system from the perspective of not just the hardware, but from the perspective of the software that executes on the hardware. We have understood that very special piece of the software is the Operating System and that the key functionality of the operating system is present in something call the operating system kernel. The different programs in execution and they could be more than one, on a particular computer system is each represented by some, by at least one process is a program in execution.

Some programs in execution will be represented by more than one process due to the fact that they may explicitly create more processes through special request to the operating system. In general, special request from a process to the operating system are achieved through what are call system calls, the collection of system calls is the inter-phase between the ordinary user program and the operating system.

Thank you.