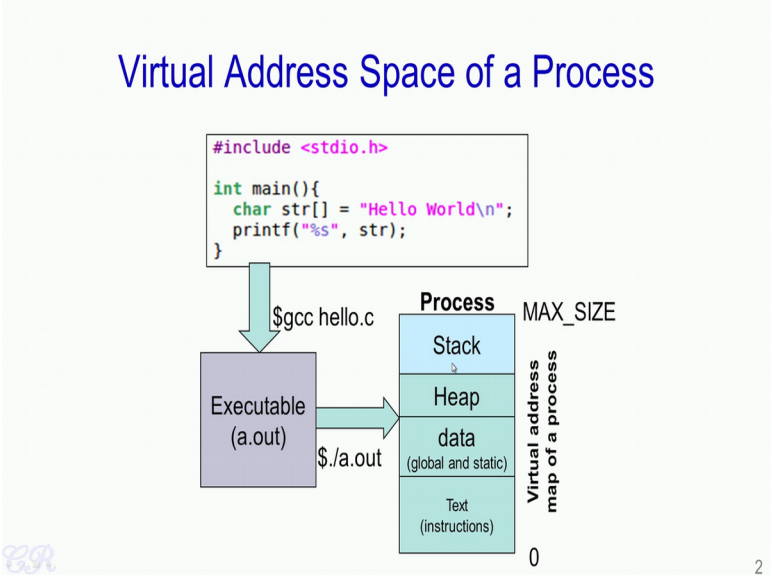
Introduction to Operating Systems
Prof. Chester Rebeiro
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Week - 03
Lecture - 11
Operating Systems (Processes)

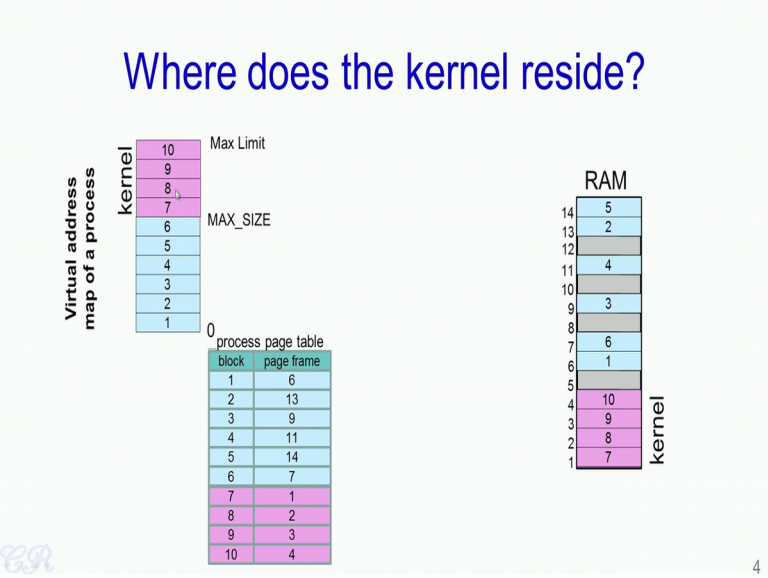
Hello and welcome to this video. So in this video we will look at Processes, which is possibly the most crucial part of operating systems. So, processes as we know is a program in execution. So we will see today how operating systems manage processes.

(Refer Slide Time: 00:38)



So let us start with this now famous example of printing "Hello world" on to a screen. So, when compiled with $gcc.hello.c it creates an executable a.out. When $a.out is executed a process is created. Part of this process will be in the RAM and it is identified by a virtual address map. So the virtual address map is a sequence of contiguous addressable memory locations starting from 0 to a limit of MAX_SIZE. So, within this virtual address map we have various aspects of the process including the instructions, global and static data, heap, as well as the stack.

(Refer Slide Time: 01:28)



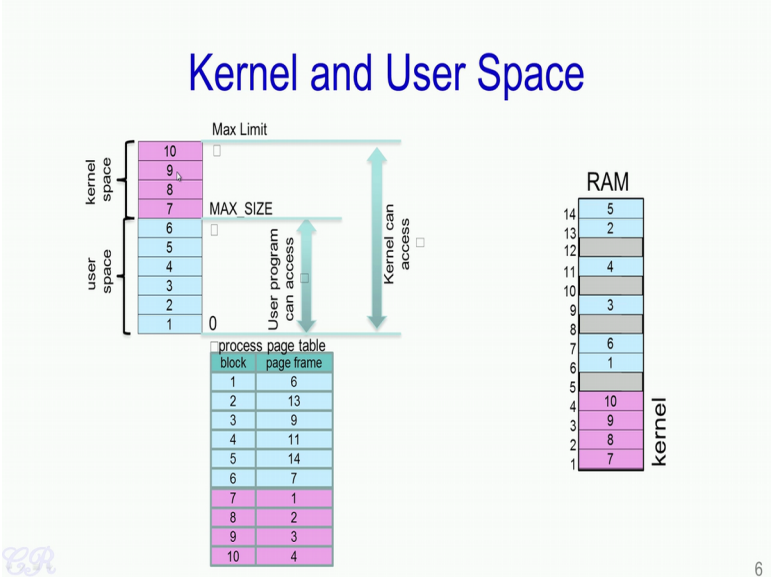
So as we have seen in an earlier video, the virtual address space or virtual address map of a process is divided into equally sized blocks. So typically the size of each block is 4KB. Again there is each process would also have a process page table in memory which maps each block of the process into a corresponding page frame. The RAM, as we have seen is divided into page frames of size 4 KB similar to the block size. And these page frames contain the actual code and data of the process which is being executed.

Now we have seen these in a previous video. But the question which we need to ask is; where does the operating system or where does the kernel reside in this entire scheme. So as we know the kernel is an other software and has to be present in the RAM to execute. Thus, in most operating systems such as Linux as well as in the operating systems which we are studying that is xv6, the kernel resides in the lower part of the memory starting from page frames 1, 2, 3, and so on.

Just like every other page frame the kernel two is divided into page frames of equal sized. Now since we are using the virtual addressing in the system, the page frames corresponding to the kernel are mapped into the virtual address space of the process. So, the kernel code and data are present above the MAX_SIZE and below a limit known as Max Limit. Now, again in the process's page table there are entries corresponding to this

map. For instance 7, 8, 9, and 10 corresponding to the blocks that have the kernel code and data and the page table tells us that they are mapped into page frames 1, 2, 3, and 4.

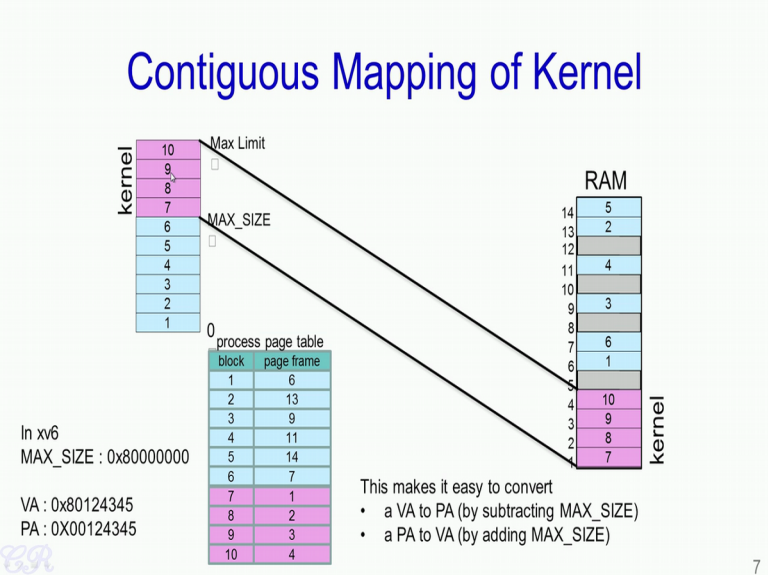
(Refer Slide Time: 03:42)



Now, we could divide this particular virtual address space into two components. One is the user space which corresponds to the blue area which contains the user processes, code, data, and other segments such as the stack and heap. Again, there is the kernel space which corresponds to the kernel code data and other aspects of the kernel.

So, the MAX_SIZE defines the boundary between the user space and the kernel space. A user program can only access any code or data present in this user space. The user program cannot access anything in the kernel space. On the other hand, the Kernel can access code as well as data in both the kernel space as well as the user space. So, this prevents the user space programs from maliciously modifying data or modifying kernel structures.

(Refer Slide Time: 04:51)



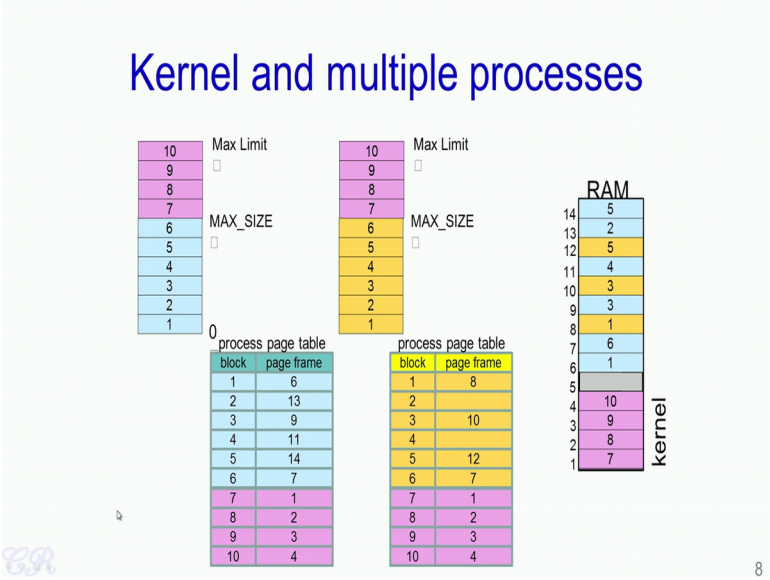
So, another thing to notice is that there is a contiguous mapping between the kernel addresses in the virtual space of the process to the corresponding physical frames in which the kernel gets mapped into. For instance, the kernel blocks 7, 8, 9, and 10 get mapped into the contiguous page frames 1, 2, 3, and 4. So, why is this contiguous mapping actually used? So, one most important aspect is that given this contiguous mapping it is easy for the kernel to make conversions from virtual address to physical address and vice versa.

For instance, to convert from virtual address in the kernel space to the corresponding physical address in the page frames of the kernel a simple subtraction by MAX_SIZE would do the trick. For instance, in xv6 where the MAX_SIZE is defined as 0x80000000, a virtual address of 0x80124345 can be converted to the corresponding physical address by subtracting the MAX_SIZE. So, the physical address would be simply written as 0x00124345.

Similarly, a physical address corresponding in the kernel code and a data in the kernel page frames can be converted to the corresponding virtual address in the kernel space by adding MAX_SIZE. For example, in this case the physical address 0x00124345 can be

converted to the corresponding virtual address in the kernel space by adding the MAX_SIZE to get - 0x80124345.

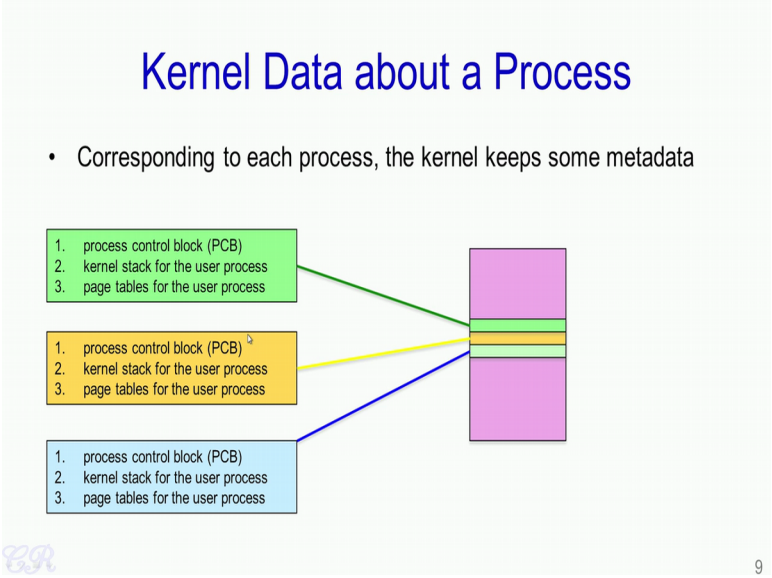
(Refer Slide Time: 06:52)



So, what happens when we have multiple processes in the system? The kernel space is mapped identically in all virtual address spaces of every process. For instance, above MAX_SIZE and below Max Limit the kernel space is present in all processes. Similarly, the page table in each process also has an identical mapping between the kernel page tables and the corresponding page frames that the kernel occupies, as can be seen (in above mentioned slide) in these few entries as well as these few entries.

Now one thing to be noticed is that, all though the virtual address space of each process has different entries for the kernel. However all processes eventually map their kernel space into the same page frames in the RAM. So, what this means is that, we have just a single copy of the kernel present in the RAM. However, there can be multiple identical entries in each processes page table corresponding to the kernel code and data.

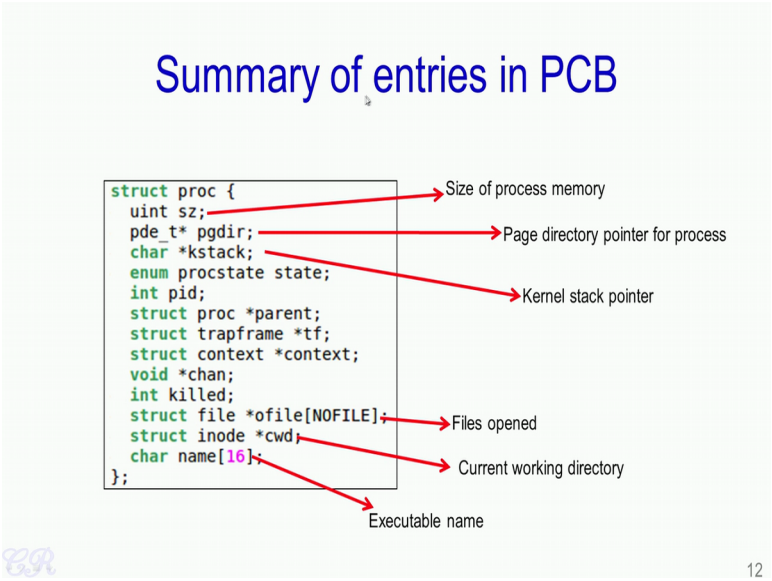
(Refer Slide Time: 08:12)



Now that we have seen where the kernel exist in the RAM as well as where it gets map to in the virtual address space of each process, now we will look at what metadata the kernel has corresponding to each process that runs in the system.

So, each process in the system has 3 metadata known as; the process control block, a kernel stack for that user process and the corresponding page table for that user process. So, each process that runs in the system will have these three blocks that is unique for that process. So we have already seen a page tables map the virtual addressable space of that user process to the corresponding page frames that the process occupies. Now we will look at the other 2 metadata.

(Refer Slide Time: 09:06)



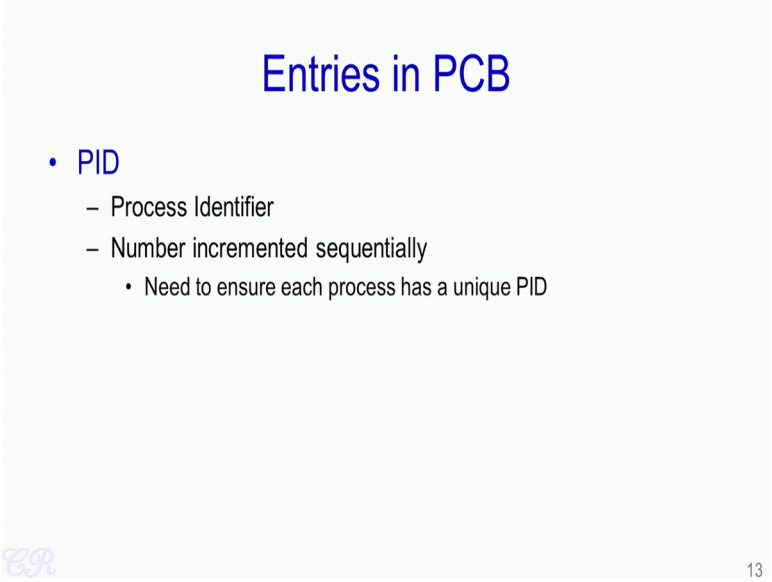
So, we have learnt that corresponding to each process there are various segments, and one important segment is the stack of the process. So this process stack present in the user space (mentioned in above slide) would have information such as the local variables and also information about function calls. So this we will now call as the user space stack. In addition to the user space stack each process will also have something known as the kernel space stack or the kernel stack for that process. So this kernel stack is used when the kernel executes in the context of a process. For instance, when the process executes a system call it results in some kernel code executing and these kernel code would use the kernel stack for it is local variables as well as function calls.

Also this kernel stack is use for many other important aspects such as to store the context of a process. In addition to the standard use of the stack such as for local and auto variables as well as for function calls, the kernel stacks plays a crucial role in storing the context of a process which would allow the process to restart after periods of time. So, why do we have two separate stacks? Why do we have a user stack for the process as well as the kernel stack? The advantage that we achieve is that the kernel can execute even if the user stack is corrupted. So attacks that target the stack, such as buffer overflow attack will not affect the kernel in such a case.

So, let us look at some of the important components in the PCB. So this particular structure is taken from the xv6 operating systems PCB which is defined as struct proc. Some of the important elements or aspects of this particular structure is ‘sz’, which is the size of the process memory, ‘pgdir’ which is a pointer to the page directory for the process. ’kstack’, which is a pointer to the kernel stack which we have defined a few slides earlier.

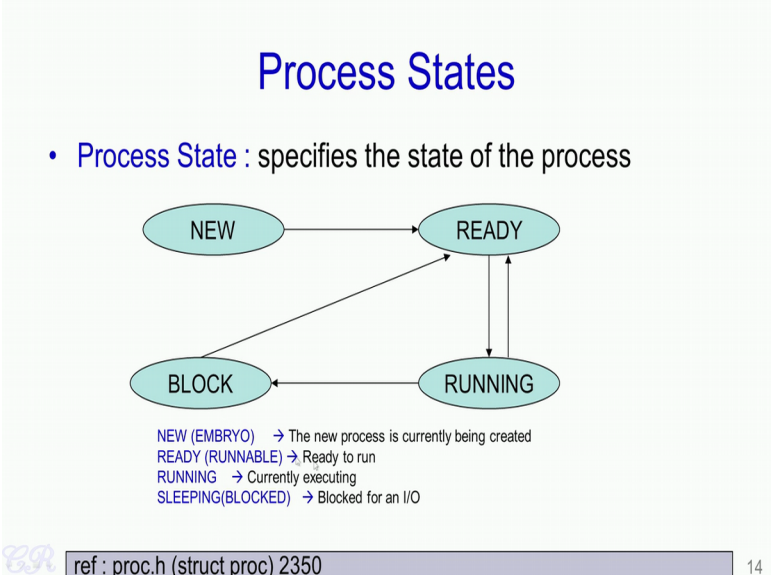
And there are other aspects such as, a list of files that are opened by the process, the current working directory of the process, and the executable name; for instance, a.out in our example. So, we will look at some of these other parameters in the forth coming slides.

(Refer Slide Time: 12:04)



So an important entry in the PCB corresponding to each process is the PID or Process Identifier. So this is an identifier for the process essentially defined as an integer and each process would have a unique PID. So typically, the number would be incremented sequentially in such a manner that when a process is created it gets a unique number.

(Refer Slide Time: 12:35)



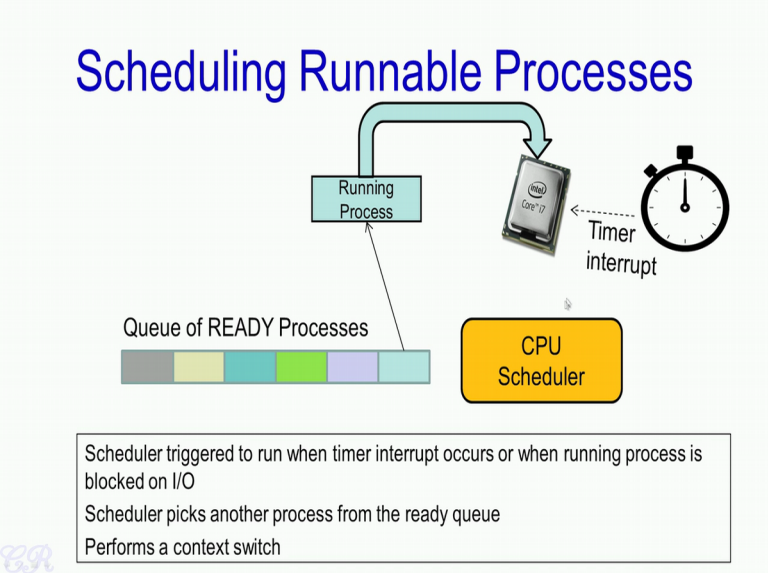
An other very important aspect in the PCB is the state of the process. So from the time process is created to the time it exists it moves through several states. Such as the NEW, READY, BLOCK state, or RUNNING state. The xv6 calls these states by different names, such as the NEW is called the Embryo which means that a new process is currently being created, while READY is known as the Runnable which means it is ready to run, while the Sleeping is known as the BLOCK state and essentially blocked for an I/O.

So, how and when does a process actually go from one state to another? So when a new process is created it is initially in the state known as NEW. When it is ready to run the state is moved to what is known as the READY state, and when it finally runs on the processor it get shifted to the RUNNING state. After running for a while the process gets preempted from the processor in order to allow other processes to run, and in such a case it goes back from the running state to the ready state.

Now, suppose during the execution of the process there is some I/O operation that is required. For instance the process could call invoke a scanner which requires the user to enter something through the keyboard. In such a case the process would be moved from a running state to a block state.

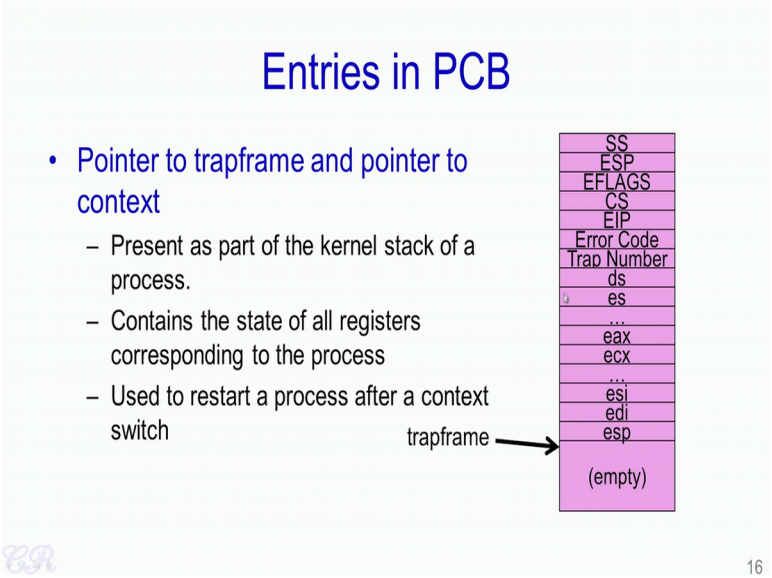
So, the process will remain in the block state until the event occurs. For instance when the user enter something through the keyboard, so when this event occurs the process moves from the block state back to the ready state. And this process of moving from one state to another from ready to running, from running to back to ready or from running to blocked and then ready keeps going on through the entire life cycle of the process. At the end when the process exists or gets terminated it goes to what is known as an EXIT state, it is not shown in this diagram. So, you could actually look up the xv6 code proc.h and which will tell give you more information about the various states. So, what is this ready state?

(Refer Slide Time: 15:19)



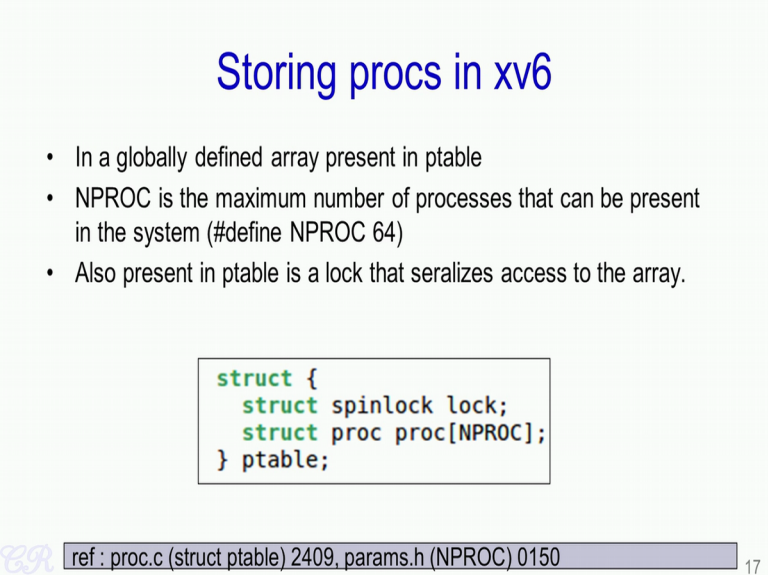
Operating systems maintain a queue of processes which are all in the ready state. When an event such as a timer interrupt occurs, a module within the operating system known as the CPU scheduler gets triggered. This CPU scheduler then scans through these the queue of ready processes and selects one which then gets executed in the processor. This selected process then changes it is state from ready to running. The running process would continue to run until the next timer interrupt occurs, and the entire cycle repeats itself.

(Refer Slide Time: 16:03)



Another entry in the PCB are pointers to what is known as a trapframe and context. So, these trapframe as well as context are part of the kernel stack and as seen in this figure (mentioned in above slide) they have a lot of information about the current state of the running process. For instance, it would say the stack met segment, the stack pointer, the flag register, the code segment, instruction pointer and so on. So, this particular trapframe and context is used when a process is restarted after a context switch.

(Refer Slide Time: 16:43)



So, how are these various PCB stored in xv6? So, in xv6 a structure known as ptable is defined. So this structure has an array of struct procs, so remember that struct procs is actually the PCB structure in xv6. So the array has NPROC entries, where NPROC is defined as 64. So, each process that was created in xv6 will have an entry in this particular array. So, you could have more information about this particular structure by looking at the xv6 code proc.c and the structure ptable. Also params.h is a file in xv6 which defines what NPROC is.

So, this gives us a brief introduction to how processes are managed in the operating system. In the next video, we will look at how a process gets created, executes, and exits from the system.

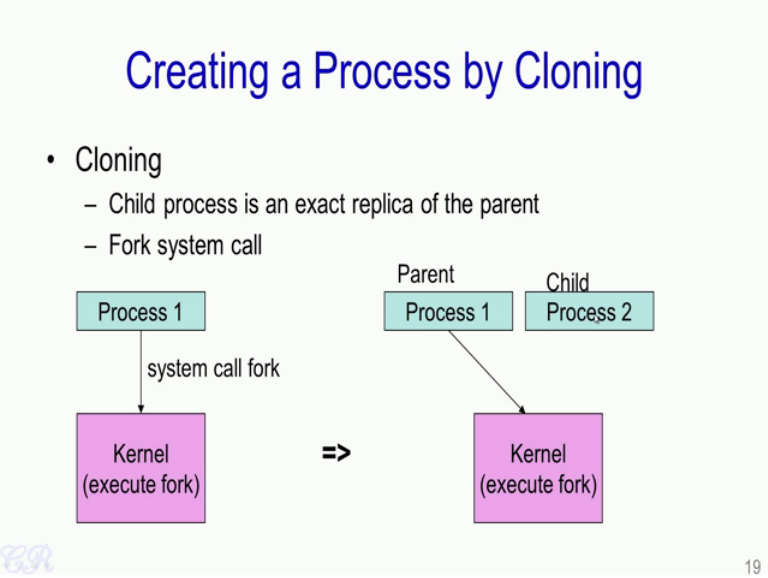
Thank you.

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

Week – 03
Lecture – 12
Create, Execute, and Exit from a Process

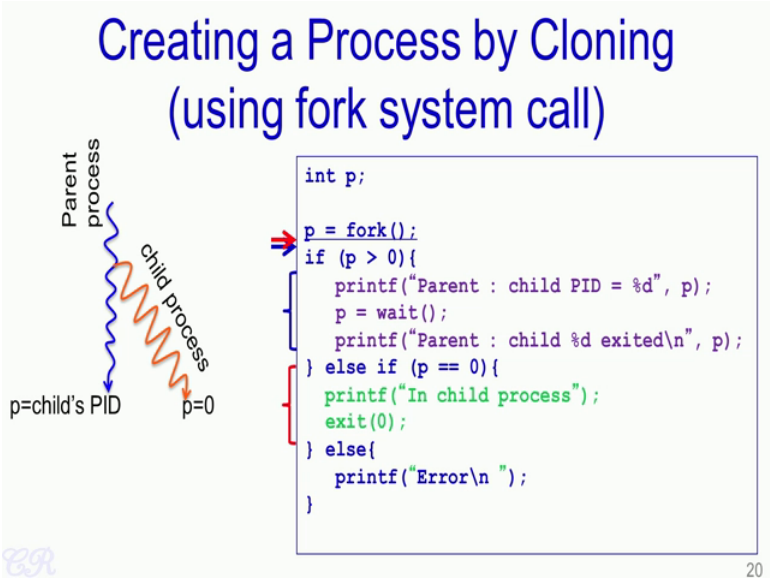
Hello, and welcome to this video. In this video, we will look at how to Create, Execute and Exit from a Process.

(Refer Slide Time: 00:27)



In order to create a process, operating systems use a technique called Cloning. When a process coincidence process 1 over here invokes a system call called fork, it results in the kernel being triggered. The kernel then executes the fork system call in the kernel space and creates what is known as a child process; here it is referred to as process 2. The child process is an exact duplicate of the parent process. So, let us see how the fork system call works with an example.

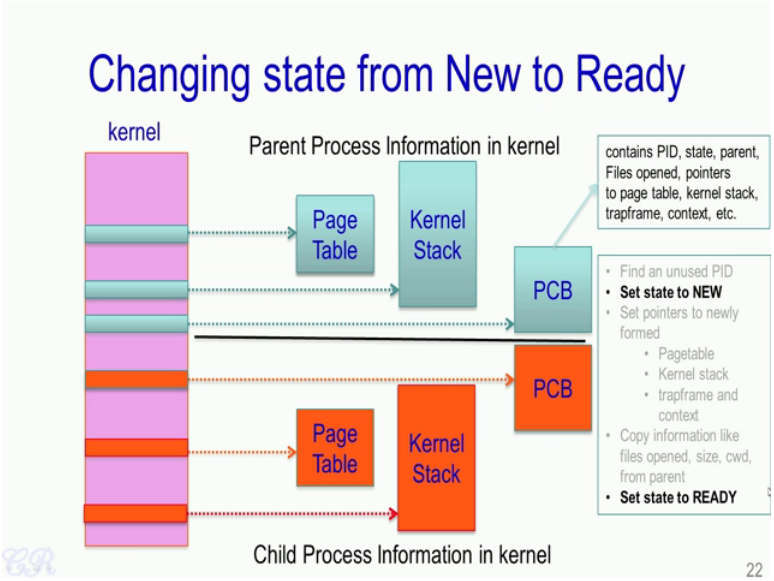
(Refer Slide Time: 01:04)



Let us say that we have written this particular program (mentioned in above image) which invokes the fork system call 'fork()'. The return from the fork system call is p and p is defined as an integer, and it can take values of -1, 0 or something > 1. So, when this particular program gets executed, the fork system call would trigger the operating system to execute, and the OS would then create an exact replica of the parent process. So, this would be the parent process (mentioned in above image) and the exact replica of this process is known as the child process. The only difference between the parent process, and the child process is that p in the parent process is a value which is the child's PID. So, this value is typically > 0; however, in the child process the value of p = 0.

So, when the OS completes its execution, both the parent as well as the newly formed child would return from the fork system call with their different values of p. In the parent process, the child's PID is > 0, therefore the parent process would execute that is 3 statements below condition if (p > 0). However, in the child process, since the value of p = 0 the else if part of that is this green statements would be executed (mentioned in above image). Now if by chance the fork system call fails a value of -1 would be returned which would result in this printf Error being printed onto the screen.

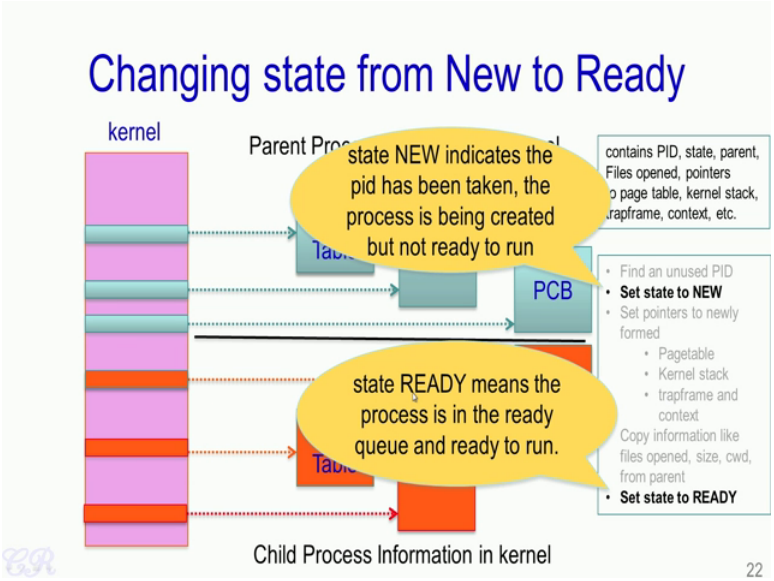
(Refer Slide Time: 02:48)



Now, let us see how this fork system call works inside the operating system. So, as we have mentioned before, each process has 3 metadata stored in the kernel. So, one is the Page Table, second is the Kernel Stack, and third is an entry in the proc structure which is the process control block - PCB. So, when the fork system call executes, a copy of the page table is made (that is Page Table in orange). So, this is (mentioned in above image) corresponds to the page table of the new child that is being created.

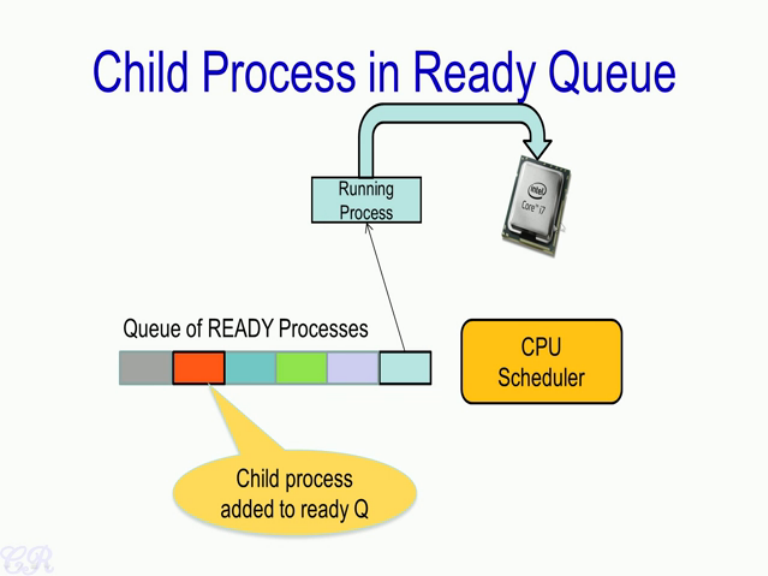
Similarly, the kernel stack of the parent process is duplicated as the child's kernel stack. Further a new PCB entry is created corresponding to the child's P C B entry. In order to do this, 1st an unused PID needs to be found by the operating system. Then a state of the newly formed process or the state of the child process is set to NEW. Then several pointers are set in the PCB such as the pagetable pointer that is the page table pointer will point to this particular page table (child page table), the kernel stack pointer would point to this newly formed copied entry (child Kernel Stack), which are present within this kernel stack. Other information such as the files opened, the size of the process, the current working directory 'cwd' etc are also copied from the parents PCB into the child's PCB.

(Refer Slide Time: 04:32)



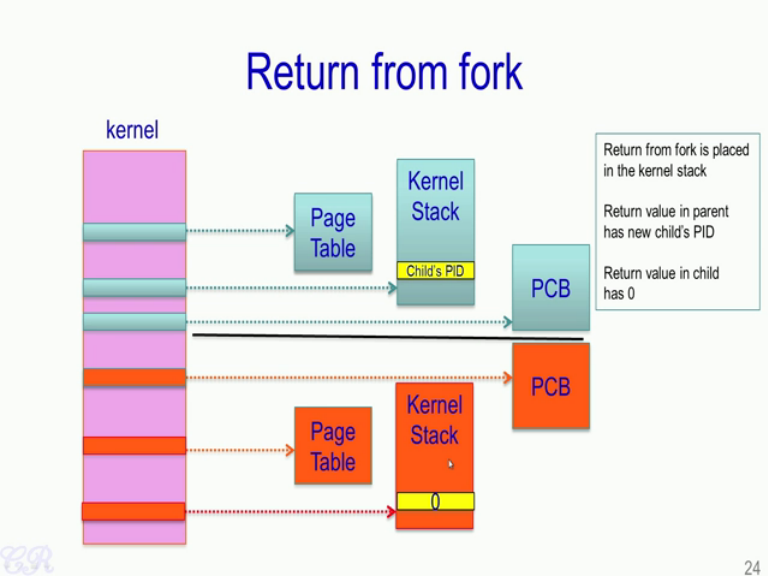
Finally just before returning, the operating system would set the state of the newly created child to READY. So, at the start of execution of the fork, the child's state is set to NEW while towards the end of execution of the fork, the OS would set the state to READY. So, why do we need this intermediate state NEW? So, what is NEW signify and what does the READY signify. So, the NEW state indicates that the PID, the chosen PID has already been taken, and the process is currently being created but not ready to run. On the other hand, when the state is set to READY it means that the various metadata within the kernel has been initialized, and the process is can be moved into the ready queue and is ready to be executed.

(Refer Slide Time: 05:31)



So, with respect to the ready queue the new process would have an entry in the ready queue and whenever the CPU scheduler gets triggered it may pick up this particular process and change its state to from READY to RUNNING, and the new process will begin to get executed within the processor.

(Refer Slide Time: 05:57)

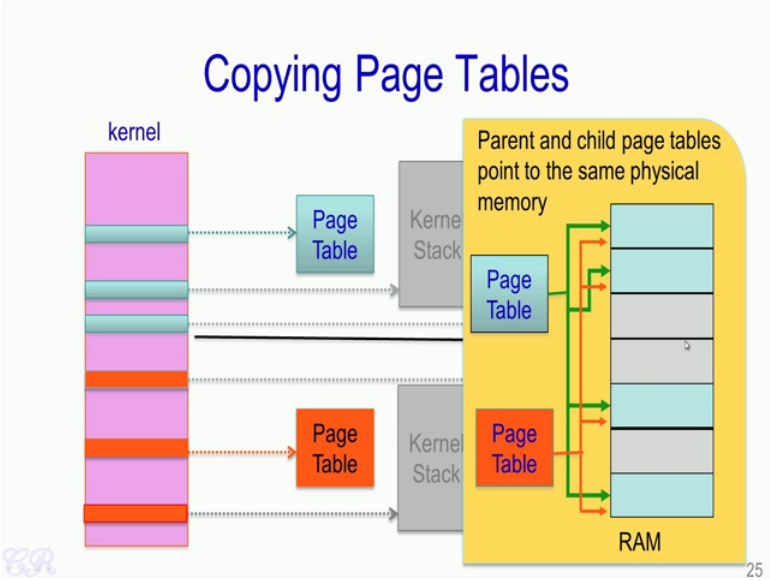


Now one important difference between the parent process and the child process with respect to the fork is that, in the parent process fork returns the child's PID; while in the child's process the fork system call would return 0. So, how does the operating system

make this difference? So, essentially the return from the fork system call is stored as part of the kernel stack. So, when the fork is executed, the OS or the operating system would set the return value in the kernel stack of the parent process as the child's PID.

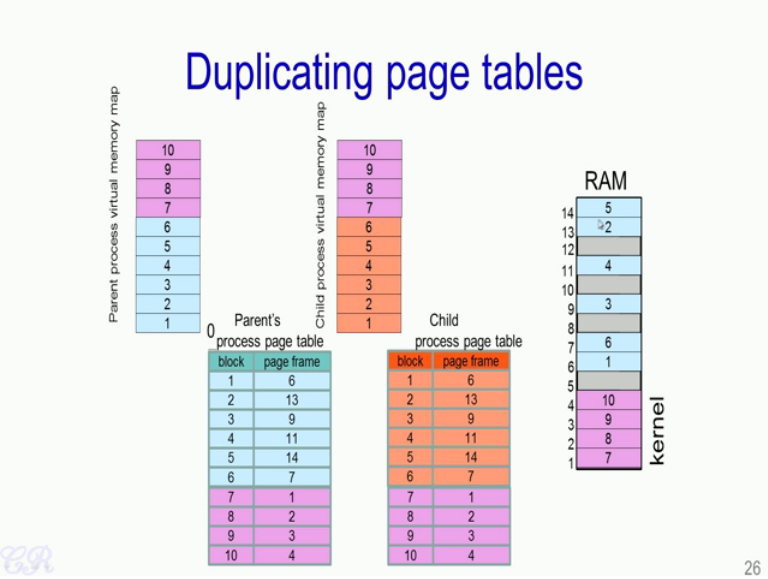
Further in the kernel stack of the child's process the return value is set to 0; thus when the system call returns in each of these cases, each process the parent process as well as the child process will get different values of the return type.

(Refer Slide Time: 06:59)



Now, one of the primary aspects while invoking fork is the duplication of the page table. So, we have the parent page table over here (mentioned in image in blue), and when fork is invoked a duplicate of this page table is created for the child (in orange). So, what does this mean to have a duplicate page table? So, when we actually look at this particular figure, we see that we have two page tables which are exactly the replica of each other and have exactly the same entries. So, what this means is that, both the parent as well as the child page table are pointing to the same page frames (mentioned in above image) in the RAM. So let us look at this in more detail.

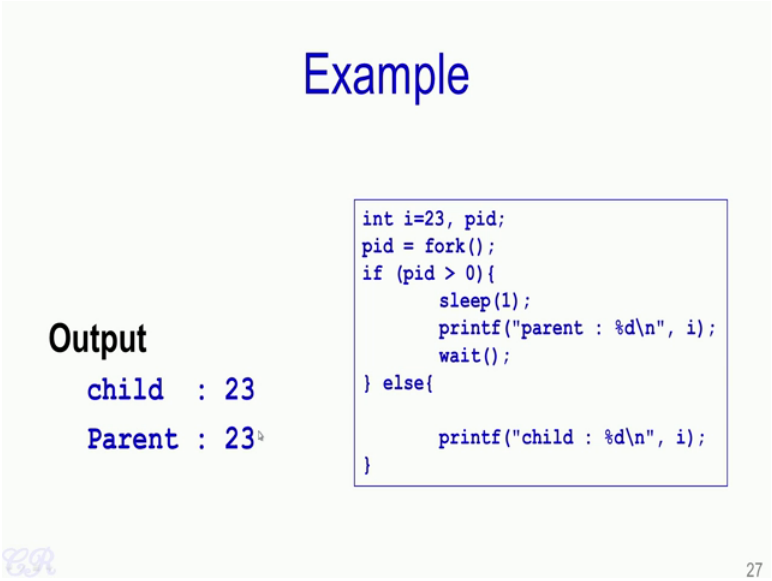
(Refer Slide Time: 07:46)



So we have the parent process with its virtual memory map, and the corresponding child process with its own virtual memory map; and each of these processes have their own page table. So, the Child process page table is an exact replica of the Parents process page table. So, what it means is that block 1 in the parent as well as block 1 in the child point to the same page frame that is 6.

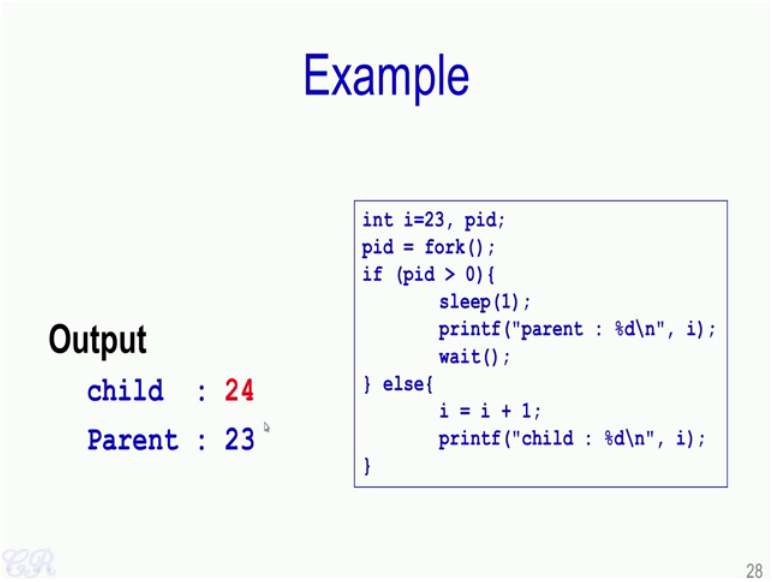
Similarly, block 2 in the parent as well as block 2 in the child point to the same page frame 13 as seen over here as well as in the child's process page table as seen over here. So, essentially what we are achieving over here (mentioned in above image) is that we have two virtual memory maps one for the parent process and one for the child. However, in RAM, we have just one copy of the code and data corresponding to the parent process, both the parent as well as the child page tables point to the same page frames in RAM.

(Refer Slide Time: 08:58)



So let us look at this with an example. Suppose, we have written this following program which invokes the fork and fork returns a pid; and in the parent process, we sleep for some time and then invoke printf which prints the value of I (mentioned in above image). So, 'i' is defined as an integer which has the value of 23. Similarly, in the child process which gets executed in this else part, we again print the value of i, now since the child is an exact replica of the parent and the value of i in the parent as well as the child would be the same. Thus when we execute this program, we get something; both the child as well as the parent prints the value of i as 23.

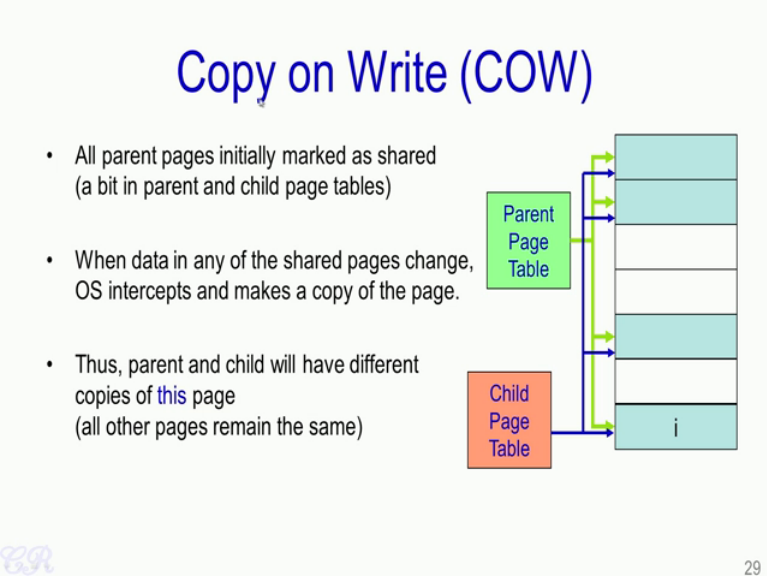
(Refer Slide Time: 09:49)



Now, let us modify this particular example and see what happens when we add this particular line (mentioned in above image). So, what we have done over here is that we have incremented i i = i + 1; only in the child. Note that in the parent process, there is a sleep(1) over here, so which means that we are giving sufficient time for this i = i + 1 to be executed by the child and only then with this printf of the parent be executed.

So, what do you think would be the output of this particular program? So, one would expect that the child would print the value of 24, since we have incremented i = i + 1, so that is 23 + 1; and also the parent would also print the value of 24. However, when we execute this program we get the child has the updated value of 24; however, the parent still has the old value of 23. Now given that both the parent and child point to the same page frames in the RAM. So, how is this possible? So, we will look into why such phenomena is happening.

(Refer Slide Time: 11:04)

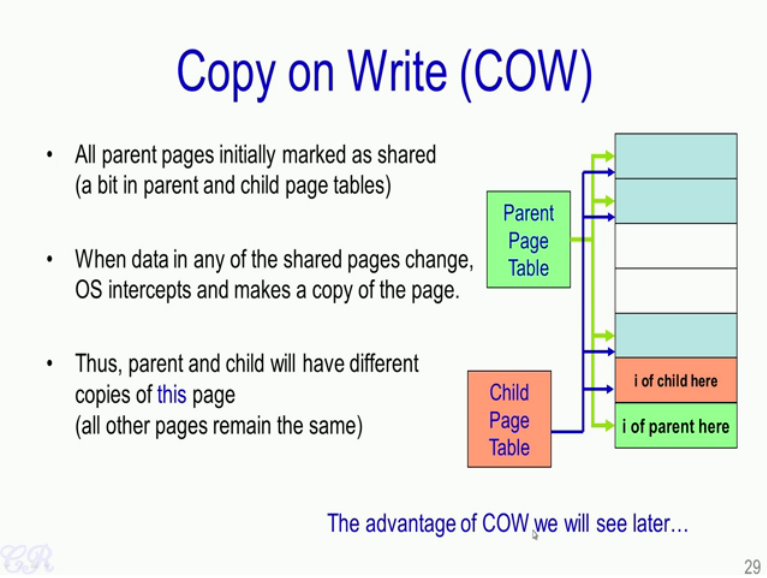


So, this phenomena occurs due to a technique known as Copy on Write that is implemented in the operating systems. So, when a fork system call is executed in the OS, all parent pages are initially marked as shared. The shared bit is present in both the parent and child page tables. When data in any of the shared pages change, the OS intercepts and makes a copy of that page. Thus the parent and child will have different copies of that page and note that 'this' page is highlighted over here (mentioned in above

image), so that; that means, only that page would be different in the parent and child while all other pages would still remain the same.

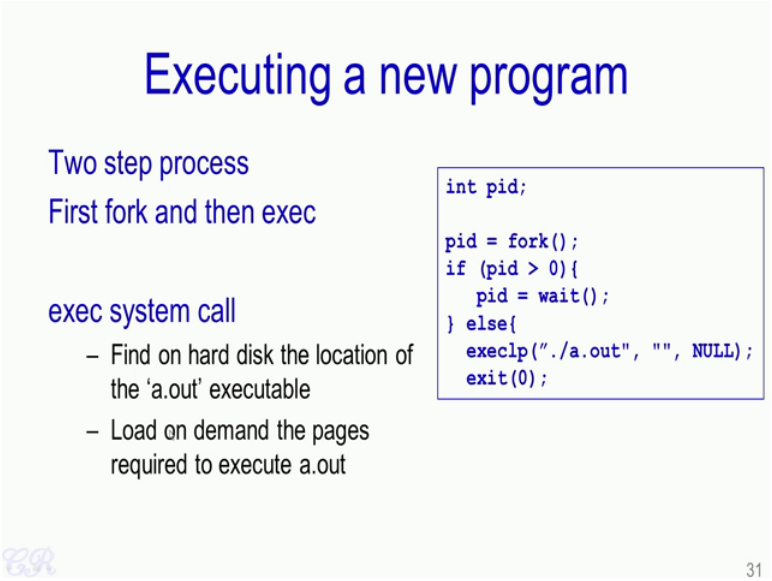
Let us see how Copy on Write works with our example. Let us say that the value of i is stored in this particular page frame (mentioned in above image), which is pointed to by the parent page table as well as the child page table. When the child process executes and increments the value of i to i + 1 that is the value of 23 is incremented to 24 and the new value is returned back, the OS would intercept the write back and create a new page for the child (mentioned in below image).

(Refer Slide Time: 12:25)



So, the i of the child process would be present here (mentioned in above image in orange color) and it would have the new updated value of 24 while the original i value of the parent would be present over here (mentioned in above image in green color), and would have the old value of i that is 23. Further, the corresponding page entry in the child's page table would then be updated. So, what is the advantage of COW (Copy of Write), we will see in a later slide. Now that we have seen how to clone a program, we shall now look how to execute a completely new program.

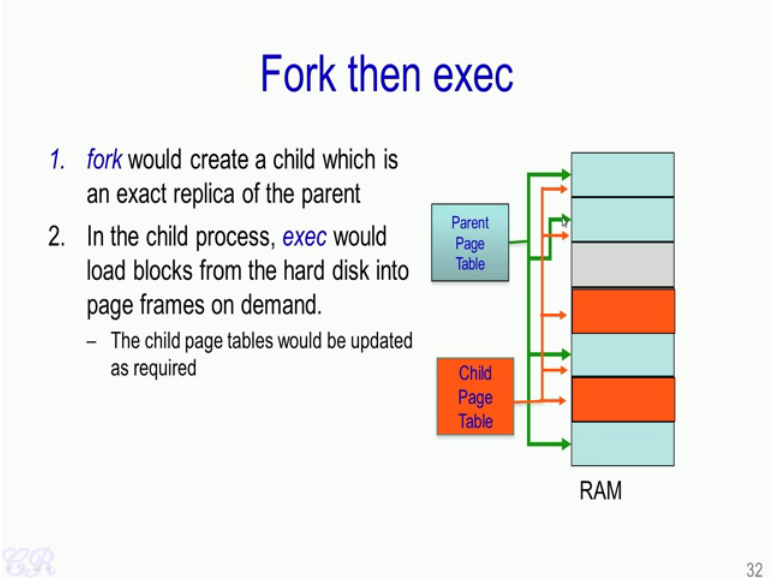
(Refer Slide Time: 13:05)



Executing a new program comprises of 2 steps; 1st a fork() system call needs to be invoked which would result in a child process created which is an exact replica of the parent process, and then an exec() system call needs to be invoked which causes the new program to be executed. So, let us take this particular small C code (mentioned in above slide). So, what we see is that initially a fork system call is invoked which should return a PID value and create a copy of the parent process called the child process.

Now in the child process, the PID has a value 0 therefore, execution will enter this else part. Now in the child process, we invoke a system call exec in this case it is a variant of exec known as execlp() and pass several arguments, of these the most important one of course is the executable file name. In our case, we are trying to execute the file a.out. So, when this exec system call is executed it triggers the operating system functionality. So, what the OS does is it finds on the hard disk the exact location of the – a.out executable then it loads on demand the pages required to execute a.out.

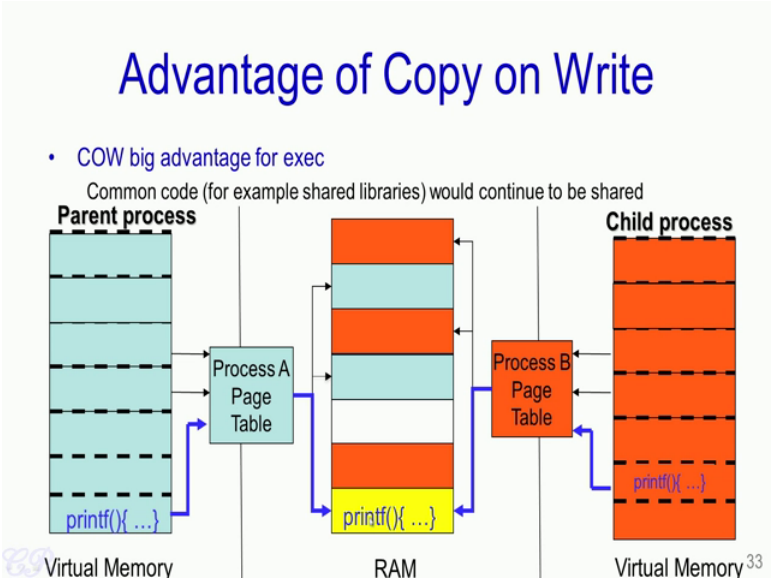
(Refer Slide Time: 14:28)



So let us actually look at this more pictorial. So, what happens when we invoke the fork? So, fork as we know would create a child process, which is an exact replica of the parent. So, what we have seen is the child has its own page tables, but all entries in the page table identically map to the same page frames as the parent. Now let us see what happens when the exec system call gets invoked. So, when the exec system call gets invoke, the operating system would find the program or the executable location in the hard disk and load blocks from the hard disk into page frames on demand.

Thus for instance, when we start to execute the new child process, it would load for instance the first block of the new child; and then on demand whenever required new blocks would be loaded into the RAM. Subsequently, the child's process's page table would also be updated as required. So, we see 2 things that occur, first we notice that whenever a new block or whenever the child program that is being executed has a new functionality a new page frame gets allocated to the child process (mentioned in above image in orange color). However, the functionality which is common to both the parent as well as the child still share the common pages (mentioned in above image in blue color).

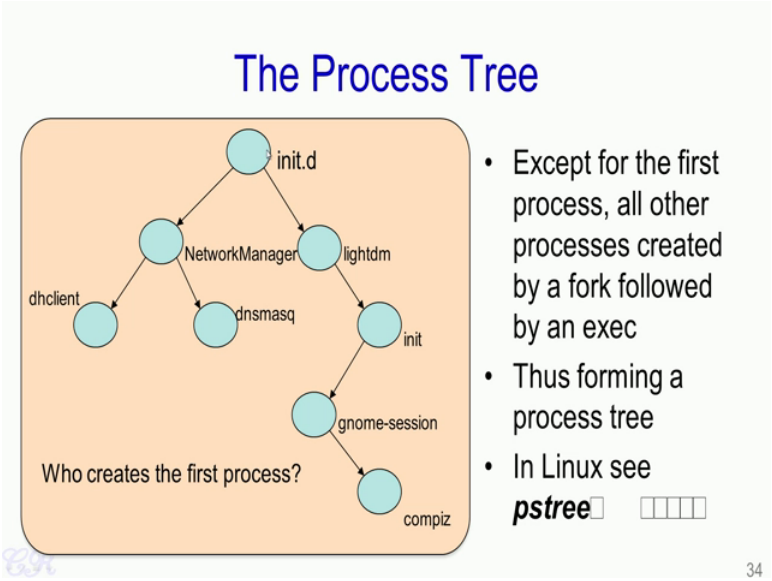
(Refer Slide Time: 15:54)



So let us see what the advantage of Copy on Write is. So, as we know most programs are written with a large number of functionality which is common. For instance, many programs we know would have would use printf or scanf or standard library function calls like freadf(), fscanf() and fprintf() and so on. Since, a new process is created from a parent by first cloning the parent and then executing the new program, thus a lot of the functionality of the parent will also be present in the child process. Now since pages are replaced only on demand in the child process, the common functionality or the common code which is present in both the parent as well as the child process is still shared between the two processes.

For instance, printf() which is present in the parent process and the printf() which is present in the child process, still point to exactly the same page frame in the RAM or in the physical memory. So, what this means is that although you may have like 100 different processes running on your system, and all these processes may use a common function such as printf(), in RAM as such there would be only one copy of printf() present. All processes would then use their page table to point to this particular page frame, which contains the printf().

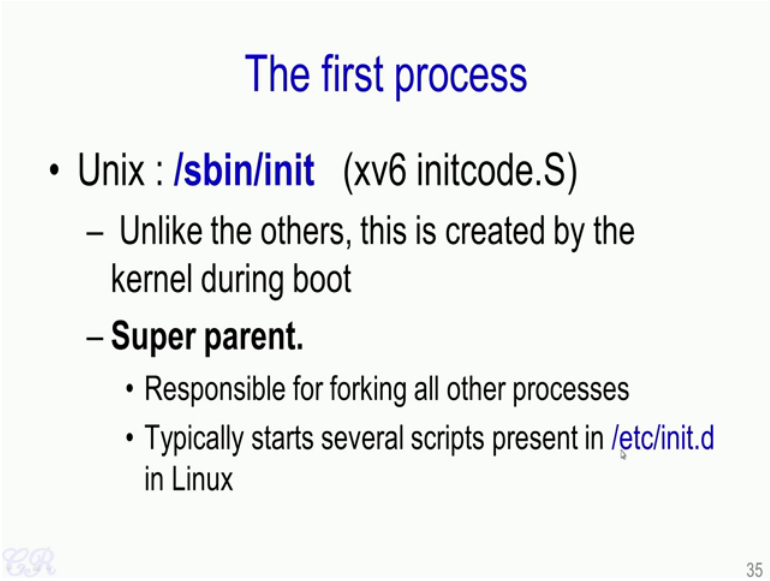
(Refer Slide Time: 17:21)



So we have seen that creating a new process first requires a fork, and which is then followed to by an exec system call. So, every process in the system is created in such a way. Therefore we get a tree like structure where you have a root node known as init.d, and every subsequent node represents a process running in the system and is created from a previous process. For instance, the process compiz (mentioned in above image last circle) is the child process of gnome-session. Gnome-session in turn is a child process of init and so on.

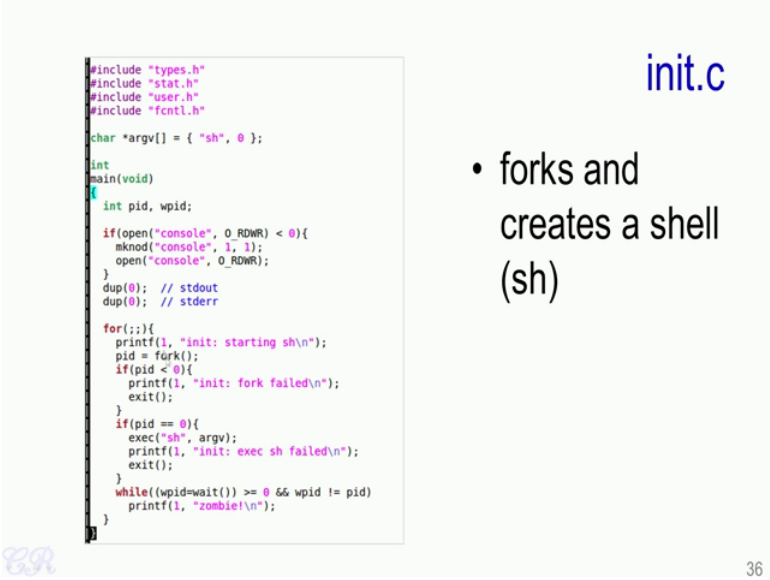
So, eventually we reach the root process which in this case is known as inti.d. If you are interested, you can actually look up or execute this particular command that is pstree from your bash prompt which lists all the processes in your system in a tree like structure. So, we have seeing that every process in the system that is executing has a parent process. So, what about the first process? So, it is the only process in the system, which does not have a parent. So, who creates this process?

(Refer Slide Time: 18:33)



The first process in xv6, the first process is present in initcode.S is unlike the other processes, because this particular process is created by the kernel itself during the booting process. So, in Unix operating systems, the first process is known is present in /sbin/init. So, when you turn on your system, and the operating system starts to boot, it initializes various components in your system and finally, it creates the first user process which is executed. So, this is sometimes known as a Super parent, and its main task is to actually fork other processes. Typically in Linux the first process would start several scripts present in /etc/init.d.

(Refer Slide Time: 19:29)

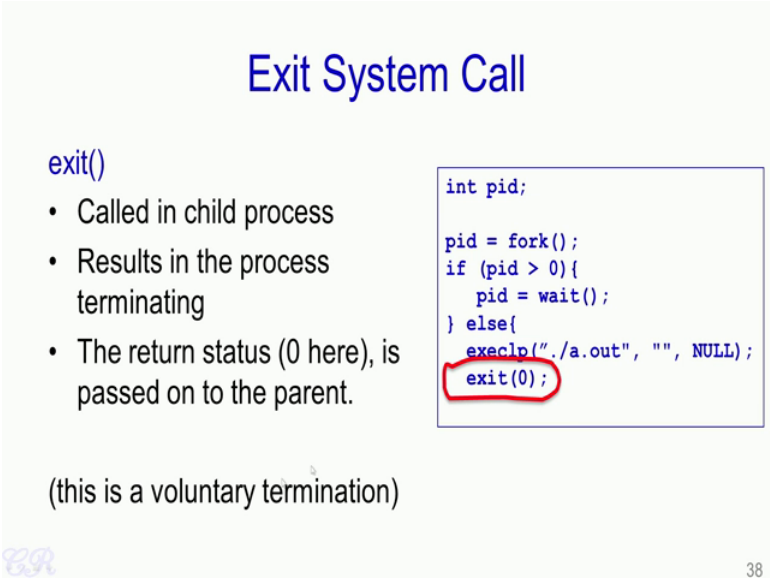


Now let us look at the code init.c, which is part of xv6. So, essentially this particular code forks a process and creates a shell (sh). So, this is a snippet of the code in particular I would like you to notice this particular for loop for(; ;), which is an infinite for loop. So, it runs infinitely; its only task is to fork a process creating a child process; and in the child process (inside if condition), it runs exec("sh", argv); which is a shell with some argument. Then it waits until that particular forked process completes, and then this thing continues forever.

(Refer Slide Time: 20:16)

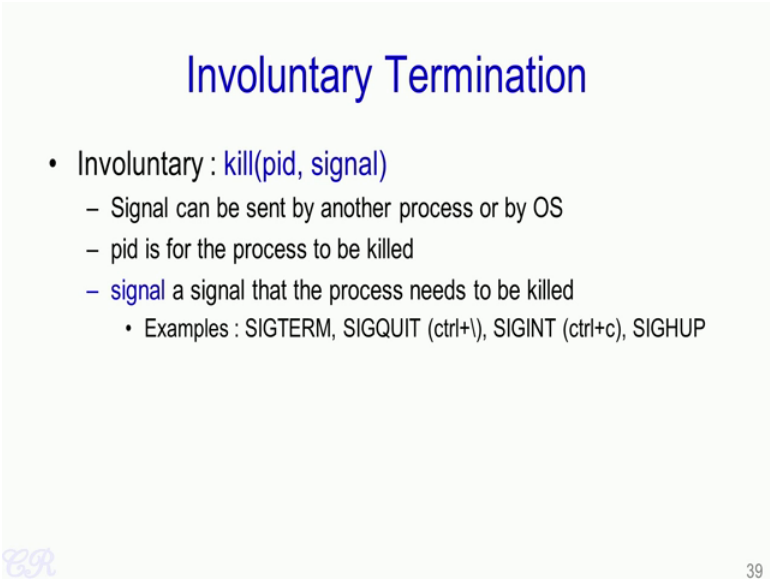
Now that we have seen how processes are cloned, how a new program is executed. Let us see how to terminate a process.

(Refer Slide Time: 20:27)



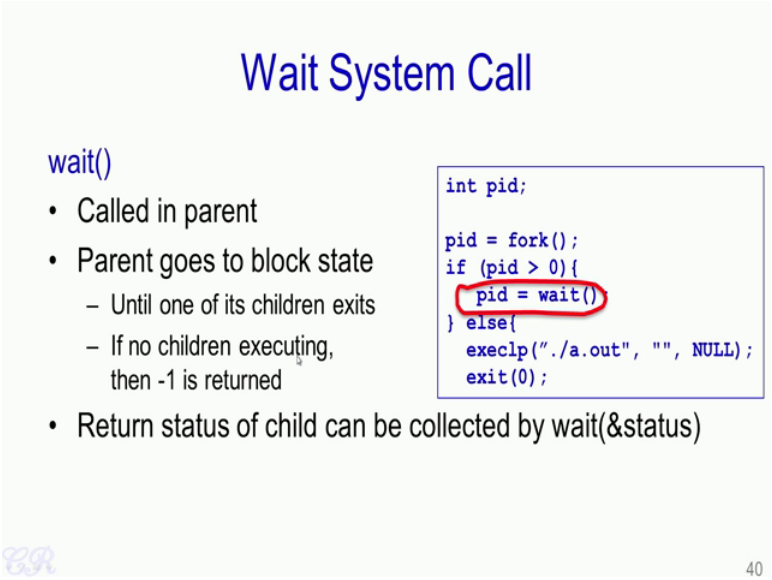
A process gets terminated by something known as an exit call. So, for instance, let us go back to our example (mentioned in above image). And in the child process, we actually run the executable a.out or execute, which then gets executed and this is followed by an exit(0) (mentioned in red circle). So, this exit is invoked in the child process, and the parameter 0 is a status which is passed on to the parent process. So, this particular way of terminating a process is known as a voluntary termination.

(Refer Slide Time: 21:03)



In addition to the voluntary termination, there is also something known as an Involuntary termination. In such a termination, the process is terminated forcefully. So, a system call which actually does that is the kill system call kill(pid, signal), which takes two parameters pid that is the pid of the process which needs to be killed and a signal. A signal is an asynchronous message which is sent from the operating system or by another process running in the system. There are various types of signals such as SIGTERM, SIGQUIT, SIGINT and SIGHUP. When a process receives a signal such as SIGQUIT, the process is going to terminate irrespective of what operation is being done.

(Refer Slide Time: 21:48)

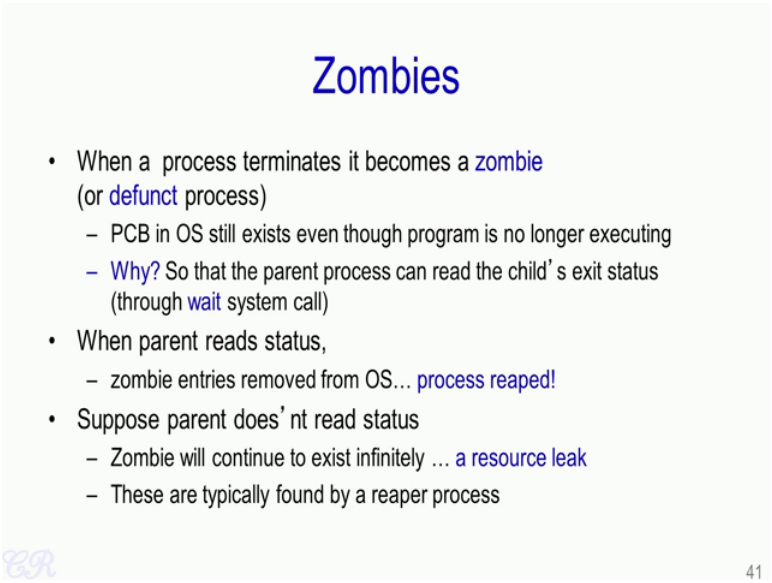


So, another important system call with respect to process termination is the Wait system call. The wait system call is invoked by the parent; it causes the parent to go to a blocked state until one of its children exits. If the parent does not have any children running then a -1 is returned. So, let us go back to our example over here (mentioned in above image), where the parent has forked a child process; and in the parent process, it obtains a PID which is equal to the child process's PID.

So, this would result in the parent actually executing the statement wait(); and it would cause the parent process to be blocked until the child process has exited. When the child process exits i.e exit(0), it would cause the parent process to wake up and the wait function to return with a return value of pid, which is the child process's pid. So, in order to obtain the return status of the child that is in this particular example 0, the parent

process can invoke a variant of wait i.e wait(&status). So, in this particular variant, a pointer is passed to status in which the operating system will put the exit status of the child process.

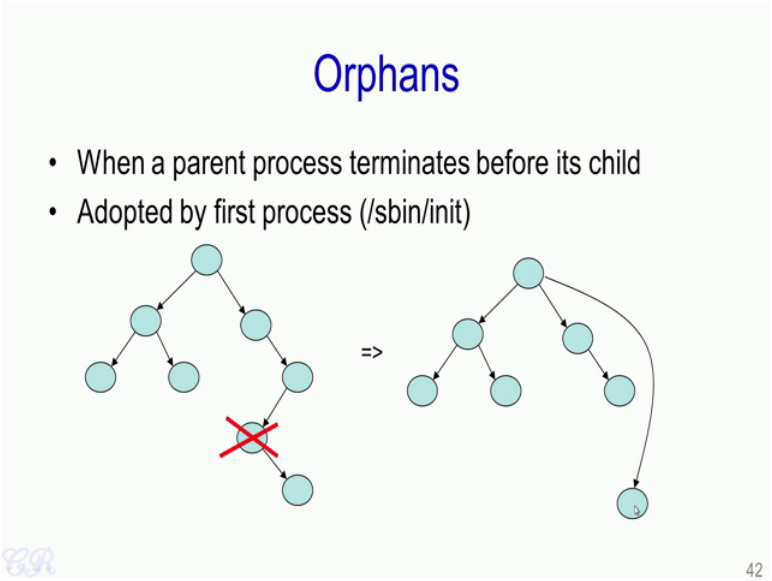
(Refer Slide Time: 23:15)



When a process terminates, it becomes what is known as a Zombie or a defunct process. What is so special about a zombie is that particular process is no longer executing; however, the process control block(PCB) in the operating system will still continue to exist. So, why do we have this concept of zombies in operating system? So, zombies are present, so that the parent process can read the child's exit status through the wait system call. When a particular program exits, its exit status is stored in the PCB present in the operating system. So, when the wait system call is invoked by the parent process, the PCB of the exiting child process would be read and its exit status would be taken from there. When the wait system call actually is invoked by the parent, and the extra zombie entries present in the OS will be removed. So, this is known as the process reaped.

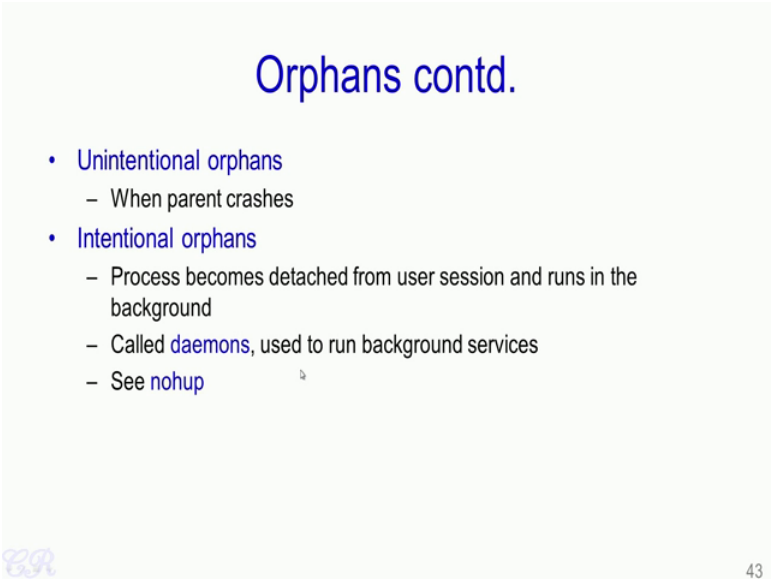
So, what happens if the parent does not read the child's status? In such a case, it will result in a resource leakage and the zombie entries in the operating system will continue to exist infinitely. So, these are eventually removed by a special process in the OS known as the Reaper process. The reaper process periodically runs and recovers any such zombie process states present in the OS.

(Refer Slide Time: 24:45)



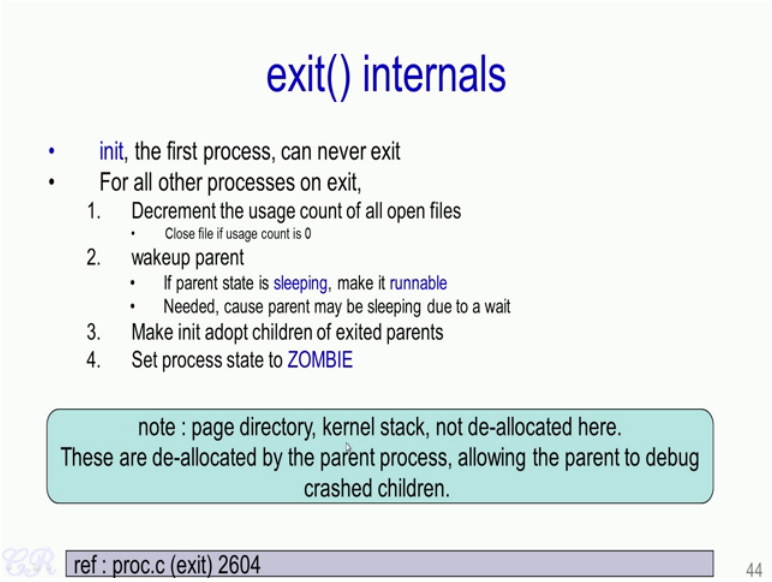
When a parent process terminates before its child, it results in what is known as an Orphan. For instance, let us consider this as the process tree (mentioned in above image) and in particular this is a process with a parent over here (crossed circle). Now suppose this particular parent exits while the child continues to execute, then the child is known as an orphan (circle below crossed circle). In such a case, the first process of the /sbin/init process will adopt the orphan child.

(Refer Slide Time: 25:20)



There are 2 types of orphans; one is the Unintentional orphan which occurs when the parent crashes. The other is the Intentional orphan sometimes called daemons. So, an intentional orphan or a daemon, are processes which become detached from the user session and run in the background. So, these are typically used to run background services.

(Refer Slide Time: 25:42)

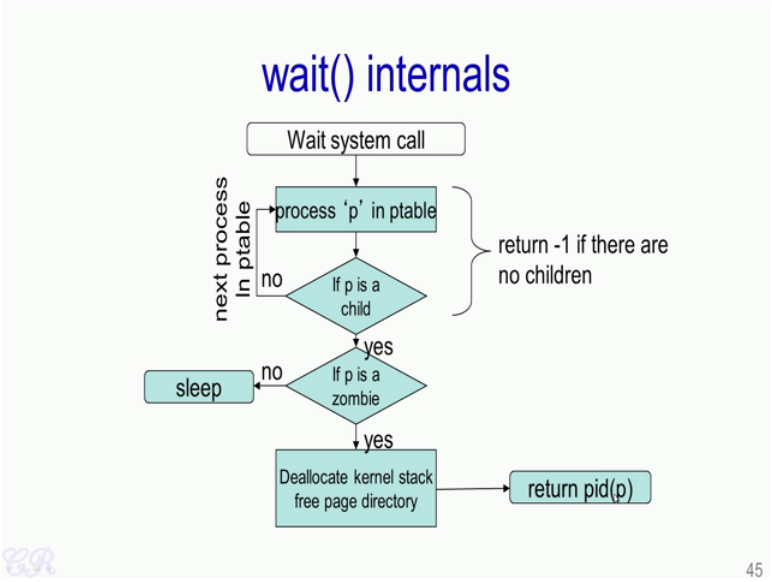


So now let us look at the exit() system call from an operating system perspective. First we notice that the sbin/init that is the first process in the system can never exit. For all other processes, they can exit by either invoking the exit system call or can exit involuntarily by a signal. So, when such a process exits, the operating system will do the following operations. First, it would decrement the usage count of all open files. Now if the usage count happens to be 0, then the file is closed. Then it would wake up parent that is if the parent is currently in a sleeping state then the parent would be made in a runnable state. So, remember that runnable state is also known as the ready state.

So, why do we need to wake up the parent? We need to wake up the parent, because the parent may be waiting for the child due to the wait system call. So, we need to wake up the parent, so that the parent could continue running. Then for all the children that the exiting process has, the operating system will make the init process adopt all the children.

And lastly, it would set the exiting process into the zombie state. So, note that certain aspects of the process such as the page directory and the kernel stack are not de-allocated during the exit. These metadata in the operating system are de-allocated by the parent process, allowing the parent to debug crashed children that is suppose the particular process has crashed, the page directory and the kernel stack will continue to be present in the operating system. Thus, allowing the parent process to read the contents of the crashed child's page directory and kernel stack, thus allowing debugging to happen.

(Refer Slide Time: 27:48)



Now, let us see about the internals of the wait() system call, so this particular flowchart in done with respect to the xv6 operating system. When a parent process in the user space invokes the wait system call, these following operations in green (mentioned in above image), occurs in the operating system. So, first there is a loop i.e "if p is a child" which iterates to every process in the p table and checks whether the process p is a child of the current process. So, if it is not a child, then you take the next process in the p table; however, if it happens to be a child, then we do an additional check to find out if it is a zombie i.e "if p is a zombie".

So, if the child is not a zombie then the parent process will sleep; however, if the child is a zombie, then we de-allocate the kernel stack and free the page directory, and the wait system call will return the pid of p (process). So, in this video, we had seen about the

process, how it is created, how it exits and about system calls such as the wait(), fork(), exec(), exit() system call.

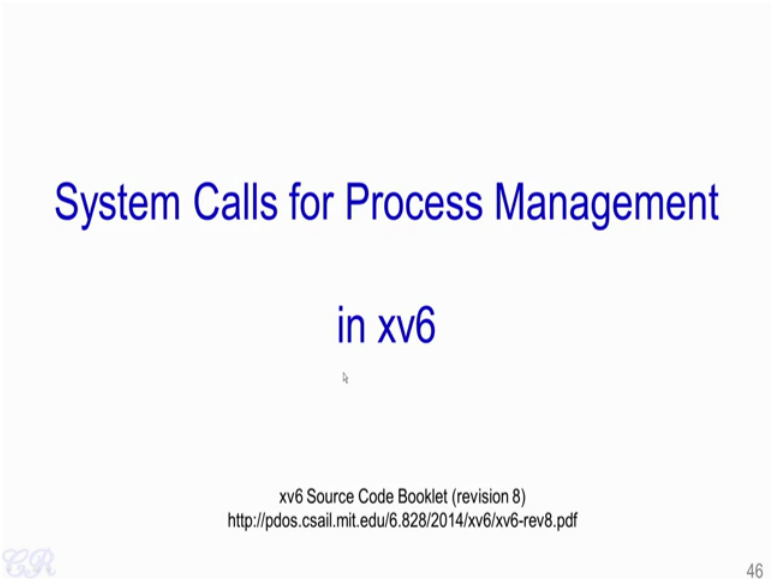
Thank you.

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

Week – 03
Lecture – 13
System Calls for Process Management in xv6

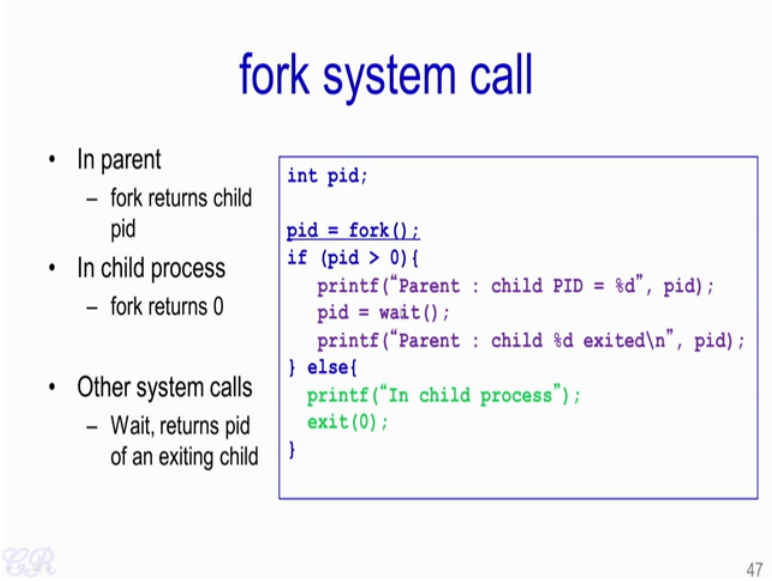
In a previous video, we had seen about several important system calls related to process management. So, we had seen these system calls such as the fork(), exec(), wait() and exit(). So in this video, we will look at how these system calls are implemented within the operating system.

(Refer Slide Time: 00:38)



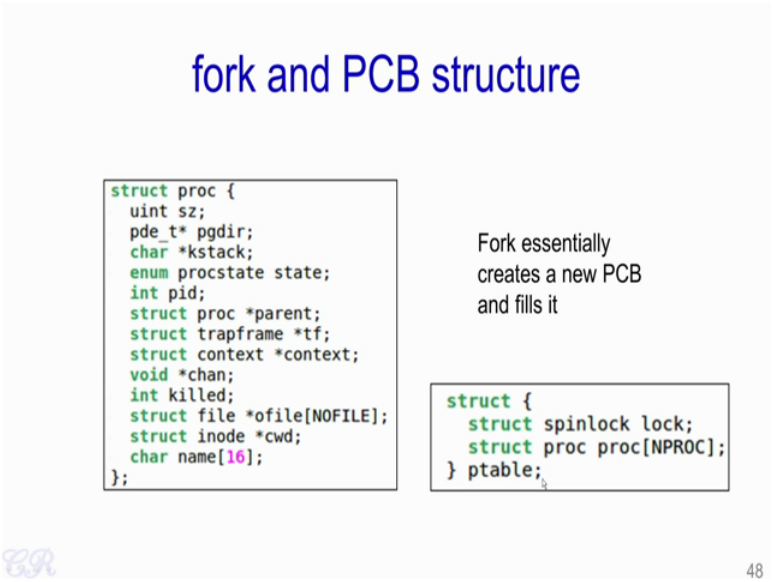
So in particular, we will see about how the system calls are implemented in the OS xv6. So, we will be referring the xv6 source code, which can be downloaded in the form of a booklet from this particular website (mentioned in above image). Please download the revision number 8, so that it matches with the particular video.

(Refer Slide Time: 01:00)



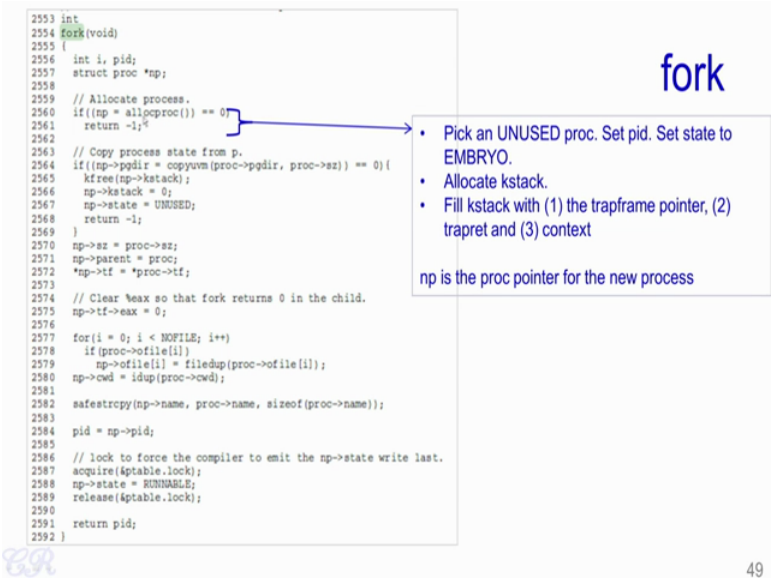
So let us start with the fork system call. In the previous video, we had seen that when the fork() system call gets invoked it creates a child process. The return value of fork that is pid over here will have a value of 0 in the child process; as a result, these green lines are what is executed exclusively by the child process (mentioned in above image). While in the parent process, the value returned by fork will be greater than 0 essentially the value returned by fork will be the child process's pid value, thus these purple lines are what is going to be executed by the parent process.

(Refer Slide Time: 01:42)



Inside the operating system, the fork essentially creates a new process control block and fills it. Recollect that in xv6, the process control block or PCB is defined by a struct proc as shown over here (mentioned in above slide). So, essentially what fork does is that it is going to fill up the various elements of the struct proc corresponding to the new child process created. Recollect also that there is a ptable that is defined in the xv6 operating system. The ptable essentially contains an array of procs, the size of the array is NPROC which is the total number of processes get that, can run at a single time in the xv6 OS. So, every process is allocated one entry in this particular ptable.

(Refer Slide Time: 02:34)

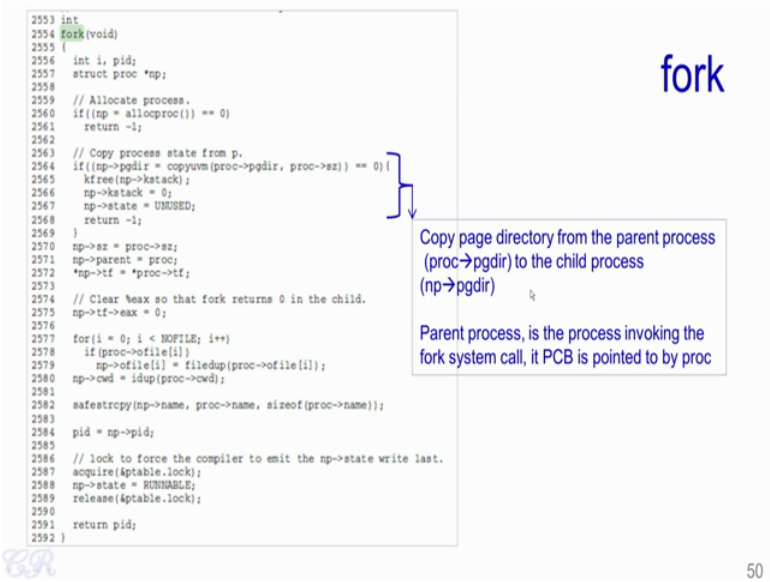


The implementation of fork in xv6 is as shown over here. So, this can be seen in the source code (booklet downloaded from website) listing in line number 2554. So the first thing you would notice over here is that we are declaring a pointer called ‘np’ which is defined as a struct proc *np;. So, this is a pointer corresponding to the new process that is been created or rather the new child process.

Now the first step that fork does is to invoke allocproc(). So, what allocproc function does is that it is going to parse through the ptable and find a proc structure which is unused; once this proc structure is found, then it is going to set the state as EMBRYO. So, recollect that in xv6 EMBRYO means a new process which is not yet ready to be executed, also which is set is the pid for the new child process. Other things which are

done in allocproc is the allocation of a kernel stack and filling the kernel stack of the new child process with a things like the trapframe pointer, the trapret as well as the context.

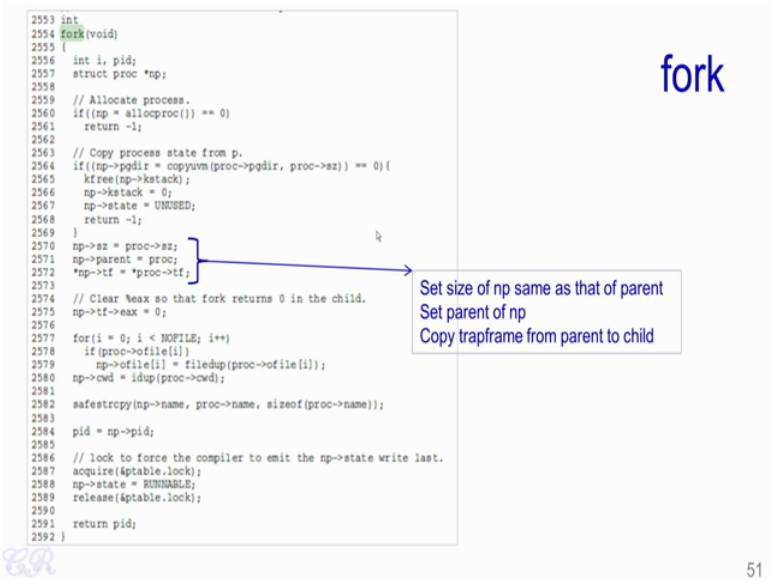
(Refer Slide Time: 03:52)



The next step in the fork implementation is this, call to this function called copyuvm(). So, essentially what the copyuvm function does is that it copies the page directory from the parent process to the child process. This copyuvm takes two parameters; it take the parent page directory which is represented by proc -> pgdir; and proc -> sz, the parent size that is represented by 'sz', and what is returned by this function is a pointer to the new processes page directory i.e np->pgdir.

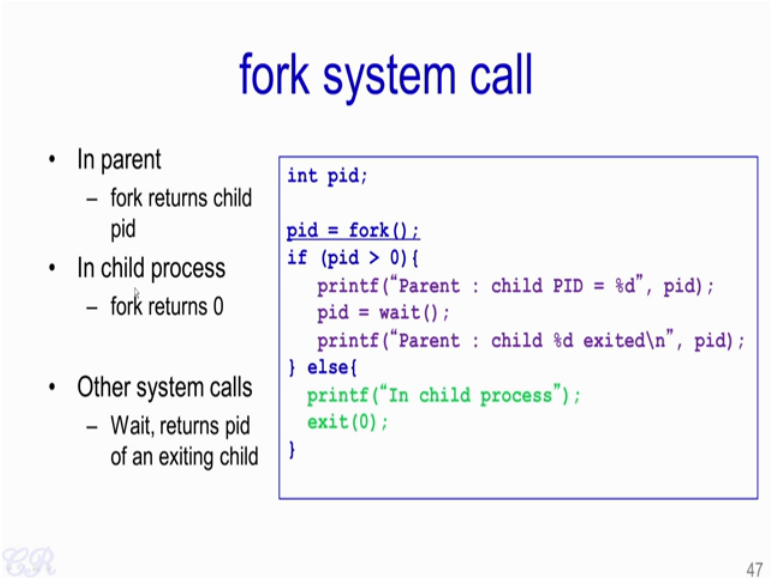
Now we will see this particular function in detail in a later slide, but for now notice that if this function fails then everything is reverted. First, the kernel stack which has been allocated previously in allocproc gets freed i.e kfree(np->kstack), and the value of np->stack = 0; and the state is set back set to UNUSED again i.e np->state = UNUSED;. Remember that allocproc had set the state to EMBRYO, now over here the state is set to unused. Then fork is going to return with -1. So, this -1 is sent back to the user process that is the process which had invoked forked.

(Refer Slide Time: 05:15)



The next step in the fork implementation is to copy some of the parameters of the parent onto the child; of these steps the most important one is the third one where in the entire trapframe of the parent process is copied onto the trapframe of the child process. Now recollect that the trapframe is used or rather recollect that a trapframe is created whenever a hardware interrupt occurs or a system call gets invoked.

(Refer Slide Time: 05:46)

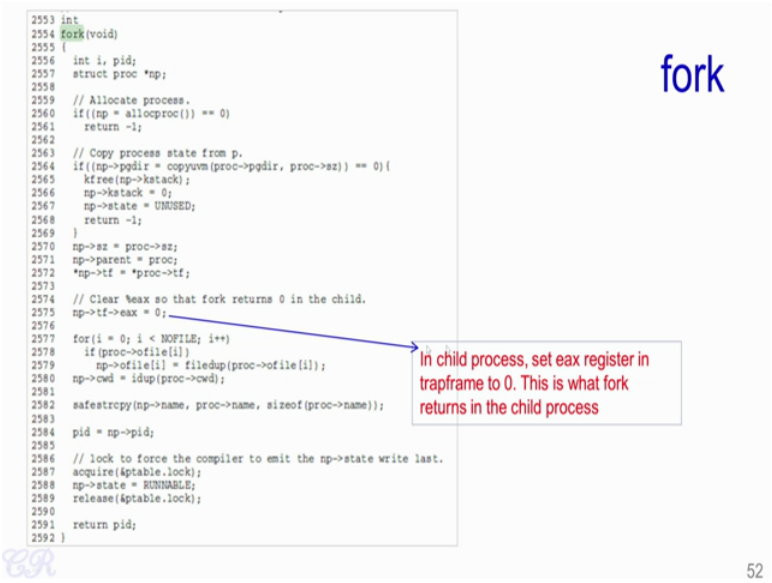


If we go back pid = fork(); when the fork system call gets invoked, it triggers the operating system to execute as well as its going to create a trapframe for this process in

the kernel stack. Now the trapframe is used, so that when the fork system call completes executing in the operating system it will return back to this point i.e pid.

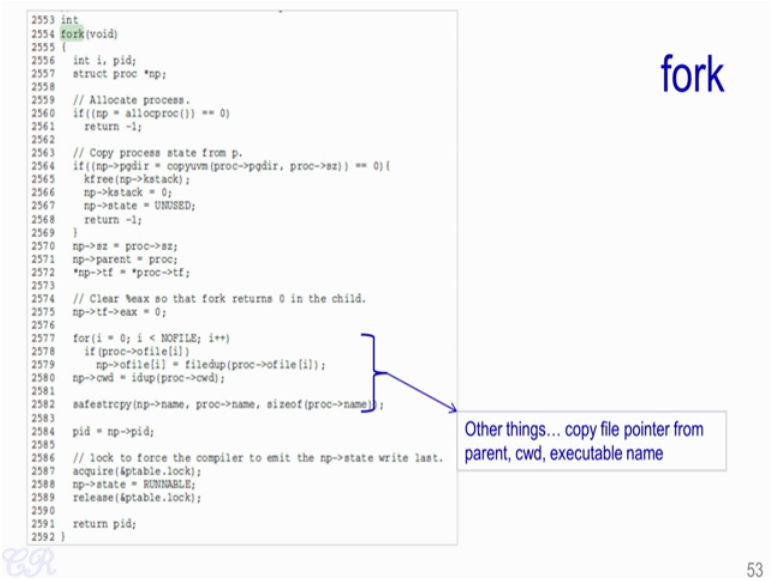
Now by copying the entire trapframe of the parent process to the child process, it will allow that the child process also continues to execute from this point i.e pid = fork(); thus when fork returns in the child process, the child process will execute from this point i.e if(pid > 0). Also, recollect that the difference between the return types of the parent process as well as the child process is that the pid value in the child process is 0; while in the parent process, it has a value that is > 0. So, we will next see how this is achieved.

(Refer Slide Time: 06:49)



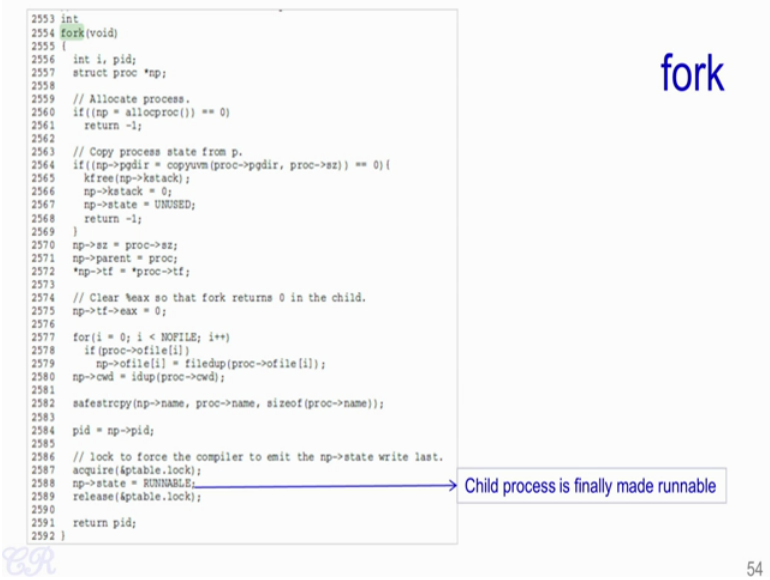
So essentially in the fork implementation, we see that in the trapframe of the new process the value of eax is set to 0 (statement marked with blue arrow in above image). So, this will ensure that in the child process, the return value of fork is set to 0. We will see later how the return value in the parent process is set to the child's pid.

(Refer Slide Time: 07:17)



In this part of the fork implementation (statements mentioned in blue parenthesis), other things are copied from the parent process onto the child process. Other things include the executable name, cwd that is the current working directory and copy of file pointers from the parent.

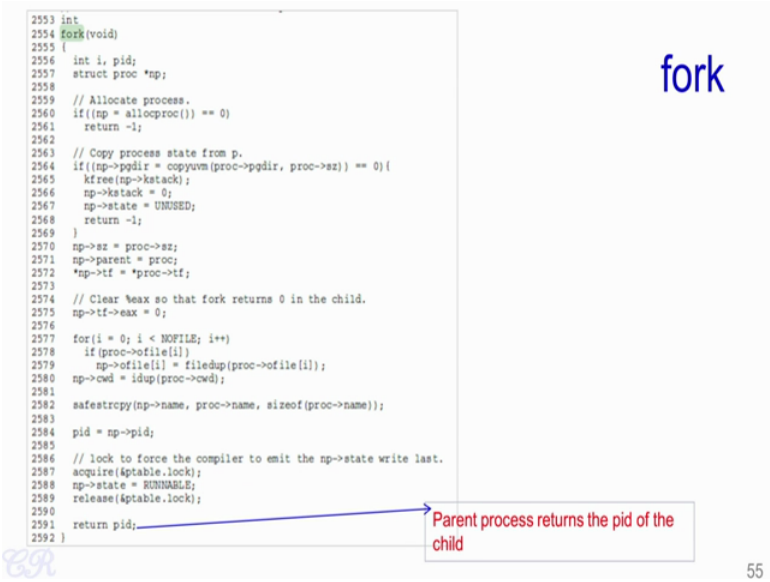
(Refer Slide Time: 07:33)



Now that the proc structure for the child process is completely filled, the state is switched from EMBRYO to RUNNABLE. So, setting the state to runnable would i.e np -> state = RUNNABLE imply that a scheduler could actually select this child process

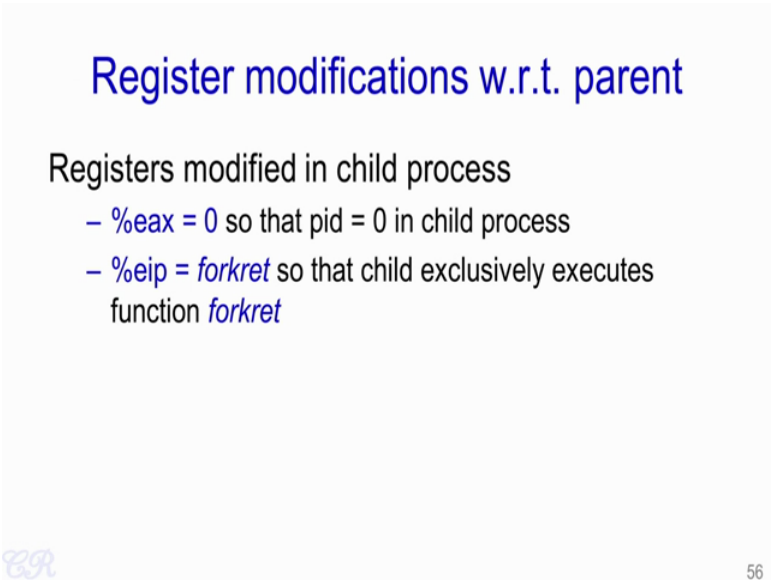
and allocate it to CPU. Thus the child process would be able to run and execute code on the CPU.

(Refer Slide Time: 07:58)



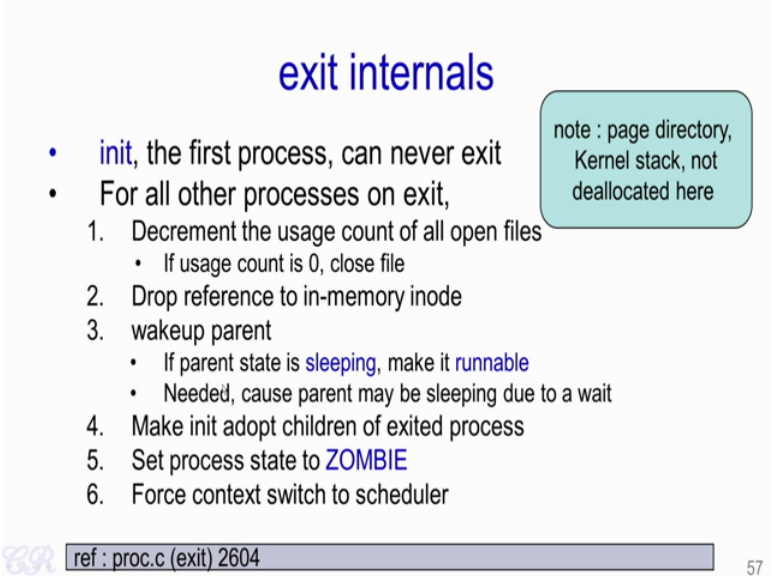
The return from the fork implementation is pid, the value of pid is set over here i.e pid = np -> pid (mentioned in above image). So, essentially the pid is np -> pid that this is the child process's pid value. And this pid value i.e return pid; goes as the return to the fork in the parent process. Thus, in the parent process in the user space, fork would return with the pid value of the child process, while as we have seen in this line over here i.e np -> tf ->eax = 0; in the child process, fork would return with the value of 0.

(Refer Slide Time: 08:35)



When it comes to the CPU registers, all register values in the child process is exactly identical to that of the parent process except for 2 registers. As we have seen before, one is the %eax register. So, in the child process the eax register is set to 0 i.e %eax = 0 so that when fork returns, it returns with the value of 0 in the child process. And other thing which is changed in the child process is the %eip or the instruction pointer. The instruction pointer is set to forkret i.e %eip = forkret which is a function which is exclusively executed by the child process and not by the parent. So, forkret is a function in the xv6 code.

(Refer Slide Time: 09:25)

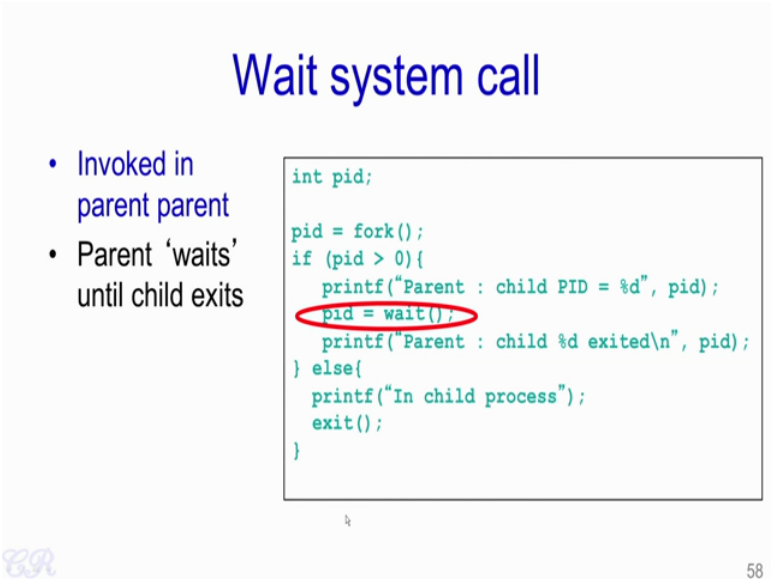


Now, we will recall the exit system call internals. So, when a exit system call gets executed, these are the 6 things (mentioned in above image) which occur inside the operating system. Essentially, there would be a decrement in the usage count of all the open files. Further, if the usage count goes to 0 then the file is closed. 2nd , there is a drop in reference for all in-memory inodes.

3rd, there is a wakeup signal sent to the parent process; essentially, if the parent state is sleeping then the parent is made runnable. Why is this needed? Essentially this is needed because a parent may be sleeping due to a wait system call and therefore, making it runnable would ensure that the wait system call becomes unblocked. The 4th point is that the exiting process will make init, recollect that init is the first process ever created by the OS. So the init process is made to adopt all the children of the exiting process, and the exiting process is going to be set to a state called a ZOMBIE state.

This setting of state to ZOMBIE is used, so that the parent process would then determine that one of its child processes is exiting. So, we will see more on this in the wait system call. And lastly it is going to force a context switch in the scheduler.

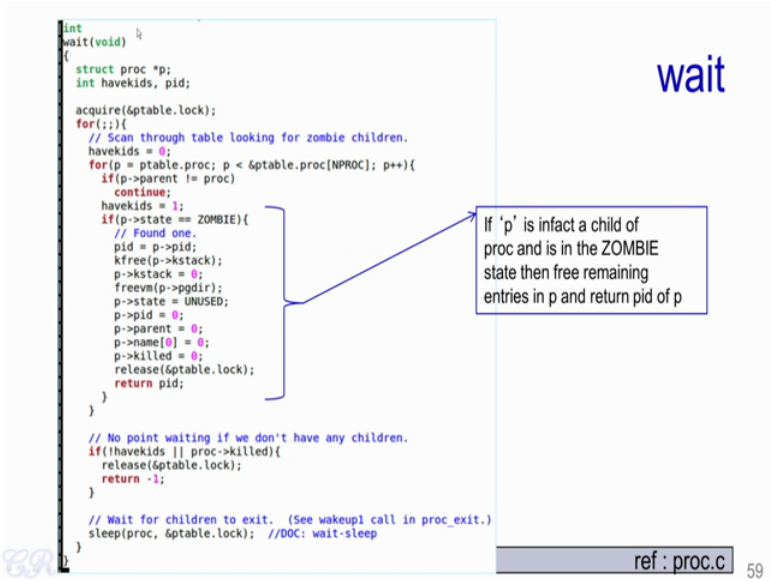
(Refer Slide Time: 11:00)



We will next see the wait system call. Recollect that when the wait() system call is invoked in the parent, it is going to be blocked until one of its child process exits. So, if no child exits then it will continue to be blocked. On the other hand, if the parent process

has no child at all, then it will return with -1. Now we will see the internals of the wait system call.

(Refer Slide Time: 11:30)



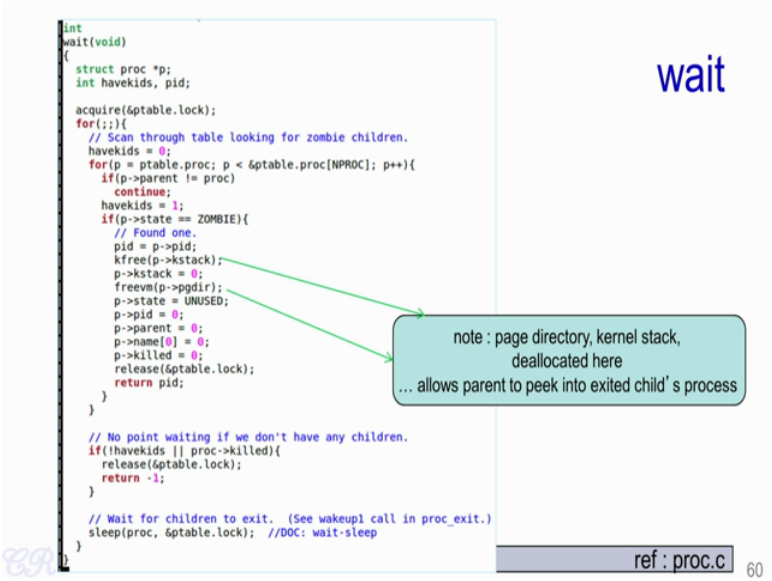
So this is the implementation of the wait system call in the xv6 operating system. So, this listing is obtained from the proc.c file of the xv6 source code. Essentially, you would see that it has an infinite loop for(; ;)which starts from 4th line (mentioned in above image) and ends at last 2nd bracket. So, within this particular infinite loop, there is an inner for loop starts from 6th line and ends with bracket. So, essentially this inner for loop parses through the ptable. So, recollect that the ptable is an array of procs; and each and every process which is in some state in the xv6 OS has an entry in the ptable. So, by parsing through all elements of the ptable, this particular loop (for loop in the code) will be able to check every process that is present in the xv6 OS in this particular instant.

So the first check that it does is to find out whether the current proc is the parent of this particular entry in the ptable, which is p (mentioned in above image). So, it is going to find out if the current proc which has invoked the wait is the parent of p (if(p->parent != proc). If it is not the parent then it just continues; otherwise it comes over here i.e havekids = 1. So, at this particular point, in the implementation, we are ensure that the p is a child of the process which has invoked the fork.

The next check is to determine the state of p i.e if(p->state == ZOMBIE). So, if the state happens to be ZOMBIE, it indicates that the child process has exited, and therefore, it

will enter this particular if condition and it will do various freeing such as it is going to free the kernel stack i.e kfree(p->kstack), set the state to unused i.e p->state = UNUSED, set pid to 0 i.e p->pid = 0, and so on. The return is this point i.e return pid, so this would result in the break of the infinite loop and it will result in the wait to exit and the return would be the pid; essentially pid is the child process's pid value.

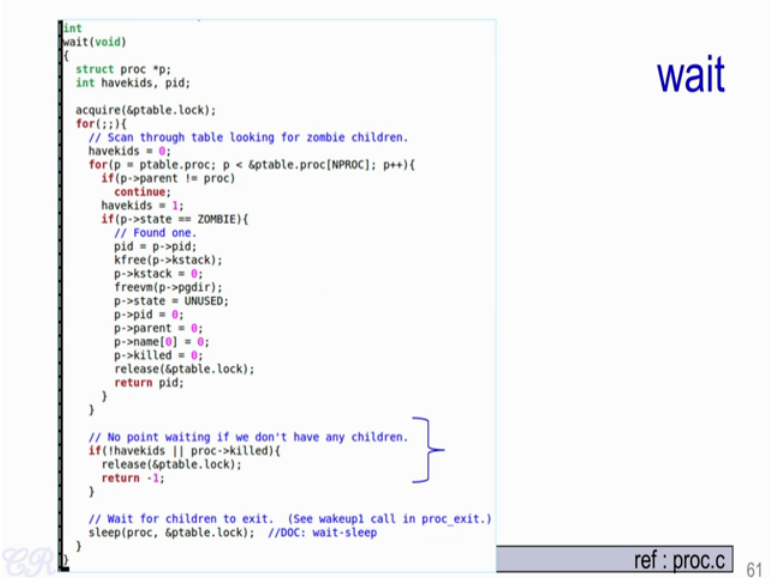
(Refer Slide Time: 14:00)



So, you may have noticed 2 things; 1st the kernel stack of the exiting child is cleared or rather is freed at this particular point i.e kfree(p->kstack), 2nd the page directory corresponding to the exiting child is freed at this particular statement i.e freevm(p->pgdir). So, freeing this child processes stack as well as page directory would allow the parent process to peek into the exited child's process. So, this enables better debugging facilities of the child process.

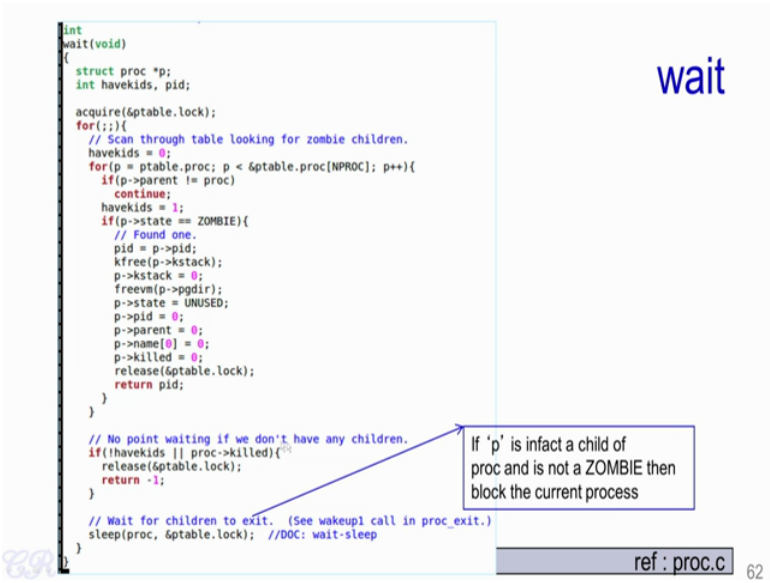
For instance, if the child happens to have crashed then the parent process could look up the stack as well as the page directory, and therefore into the physical pages of the child process and we will be able to get information about why the child process has crashed.

(Refer Slide Time: 14:54)



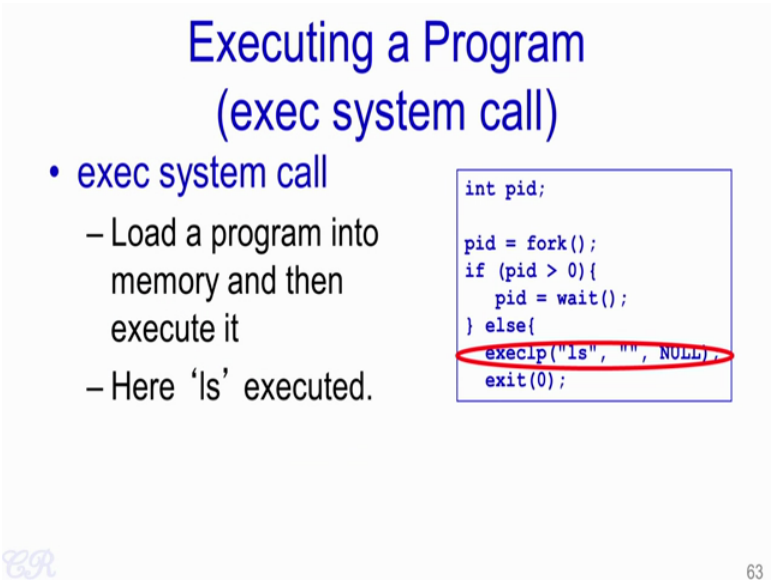
Next during the entire for loop, if we have found no children for the particular process then we just return -1. So, wait will again return to the user process, but with a value of -1.

(Refer Slide Time: 15:10)



So, execution comes to this particular line that is the sleep (mentioned in above image), when we have a child process which is not in a ZOMBIE state. So, in such a case this wait statement is going to sleep until it is woken up by an exiting child.

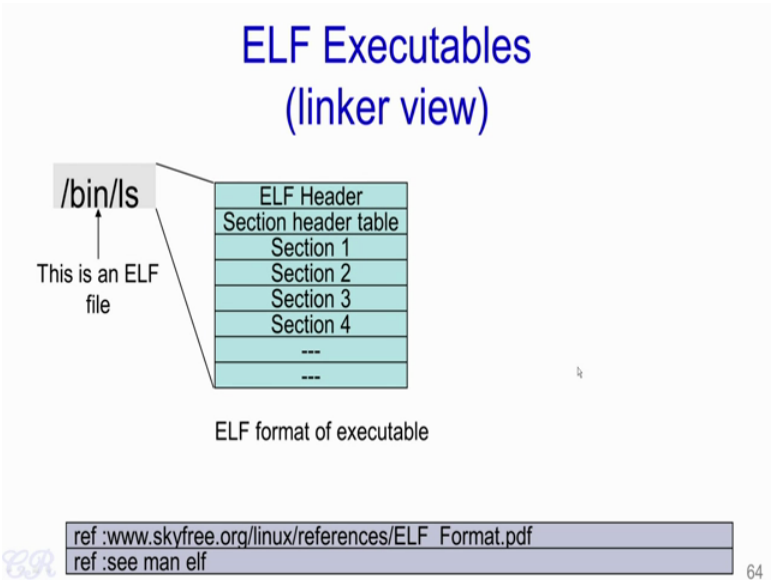
(Refer Slide Time: 15:28)



So we will now look at the internals of the exec system call. Recollect that the exec system call would load a program into memory and then execute it. In this particular code distinct (mentioned in above image), the parent process would invoke the fork system call which would result in a child process being created. The child process in this particular example would execute the exec system call; and as a result, it would cause a new program to be executed in the system. In this particular example, the new program is the 'ls' program, which lists all the files in the current directory.

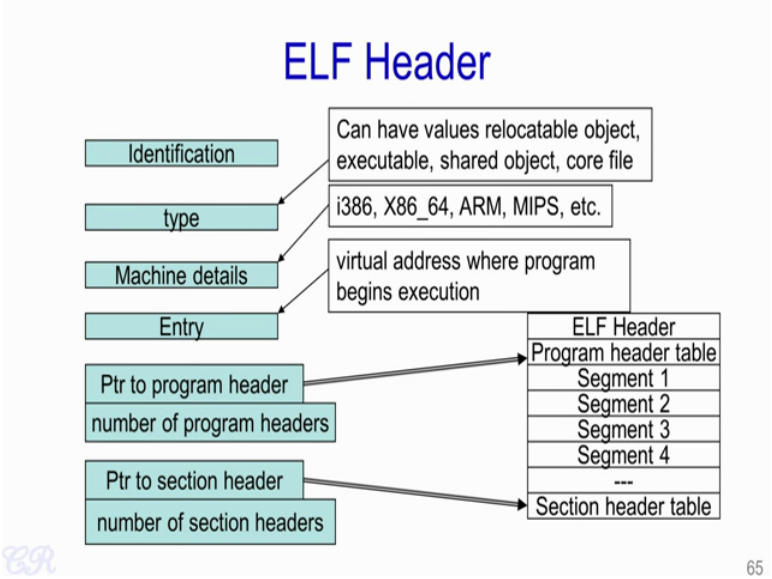
Now, 'ls' is an executable. It has a current a particular format known as the ELF format or Executable Linker Format. So, this particular format is what is interpreted internally by the exec system call within the operating system and this format is essentially understood and used to load 'ls' from the hard disk into the RAM.

(Refer Slide Time: 16:41)



Let us see what actually is present in the ELF format. So, every time we actually compile or link a program, it creates an ELF executable or an ELF object. For instance, with the example that we took that is - /bin/ls, this is an ELF executable. So, it has a format has shown over here (mentioned in above image in blue box), so at least this is part of the format, and more details about the format can be obtained from these particular references (mentioned below in above image). What we will do now is we will go through some important components of this ELF executable, and see how this is going to be useful for us that is, from an OS perspective.

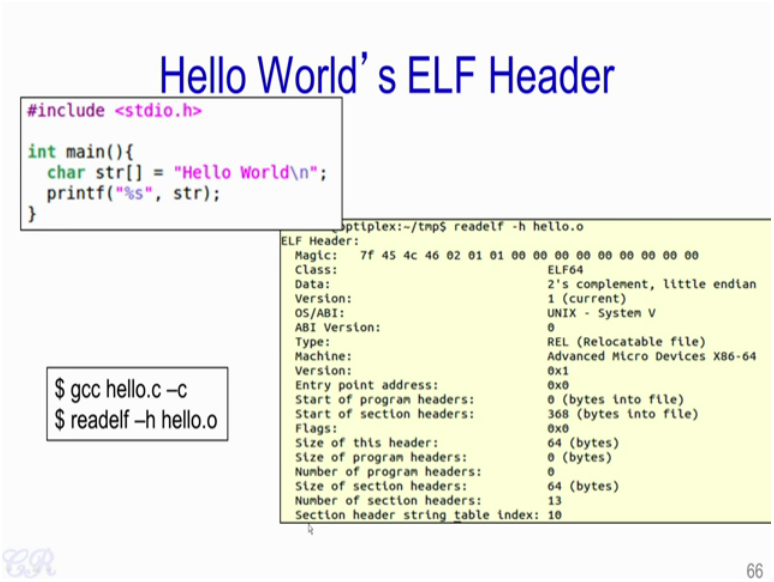
(Refer Slide Time: 17:19)



So, we will start with the ELF header. The ELF header contains various parameters and these are just a few of the parameters which are present (mentioned in above image). The ELF header starts with an Identifier. So, this identifier essentially is a magic number which is used to identify whether this file is indeed an ELF file. 2nd, there is the type so a type would tell that the type of this ELF file, so the type would have a value of executable or relocatable object or shared object or core file and so on.

Another important entry is the Machine Details. So it would tell information about whether this ELF executable or the ELF object could run on a certain machine. For instance, ELF this machine details could have values such as i386, x86_64, ARM, MIPS and so on. Then we have an Entry value in the ELF header, this entry value will tell the virtual address at which the program should begin to execute. So, beside this we have a pointer to the program header, number of program headers which are present, pointer to section headers and the number of section headers which are present. So, we will see more about this in the later slides.

(Refer Slide Time: 18:44)



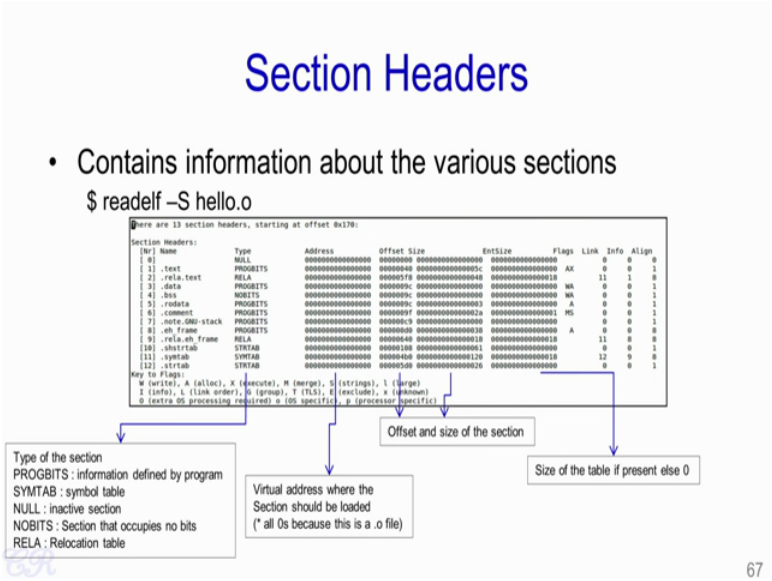
So let us first start with an example. So let us take this particular example of our hello world program written in C, and we compile it with the - c option. So, when you say $gcc hello.c -c, it creates the object file hello.o. So, this is an ELF object, we then use a utility readelf with an option - h with the object file hello.o. So the - h option would print the ELF header. So, this is the ELF header (mentioned in above image), so it has a magic

number which essentially is the identifier and it is used to distinguish this particular object file from any other file. So, it is used to say that this particular object file is indeed an ELF file.

Then another thing which we have seen is the Type of the ELF header which it could be for instance a relocatable object and so on. So, in this particular example (mentioned in above image), since we used a dot using a hello.o that is a object file, it is a relocatable object file and you can actually see it over here REL which is a relocatable object file.

And another thing is the machine details, which over here (mentioned in above image) which specified as Machine in this case it is the AMD x86-64. So, which indicates that this object file is for x86-64 bit machines, then we have seen the Entry point in this case is 0. So, other things we have seen is the Start of the program headers which is 0 (bytes into the file), this one (refer ELF header image) pointer to program headers. And we have seen the Number of program headers in this particular object file is 0. In the other two aspects are the pointer to the section headers and the number of section headers (refer ELF header image). So the pointer to section headers is the Start of section headers it is 368, and the Number of section headers is 13. So, in this way, we can actually see the various contents of the ELF header.

(Refer Slide Time: 20:49)

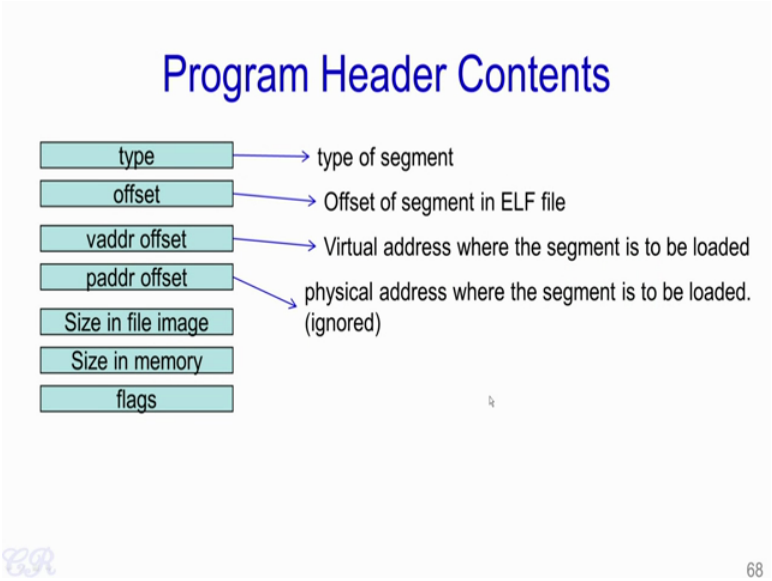


Now, we will look more into detail about the Section Headers, so in order to get a listing of the section headers you could use $readelf –S hello.o, which will print all the section

headers present in the relocatable object hello.o. So, this is actually shown over here (mentioned in above image in square box). So, we will not go into too much details of this because not much is applicable for us, but just give an explanation about the various columns. So, this particular column that is the 2nd column over here is the name of the sections, while the 3rd column gives you the type of the sections.

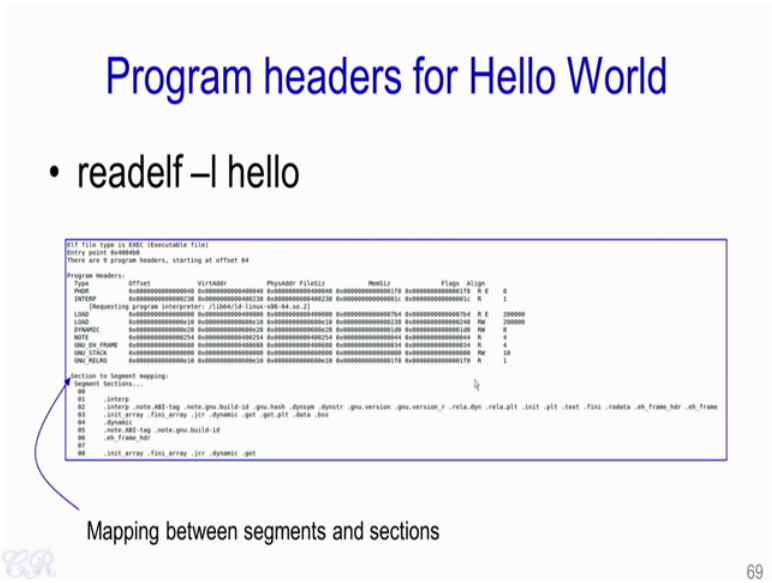
So the type could be one of these (refer first square in above image), these are the PROGBITS which is information defined by the program, SYMTAB which is a symbol table, NULL or NOBITSs essentially is the section which occupies no bits, and RELA which is a relocatable table. Then we have here the address (refer second square in above image), which is a virtual address for where the section should be loaded. In this particular case, it is all 0's because it is a .o file, it is an object file and it could be relocated. Then here you have the offset and the size of the sections (refer third square in above image), while here (refer last square in above image) you have a table size if there is a table then you have a non-zero value; however, if no table is present then you have a 0 value.

(Refer Slide Time: 22:11)



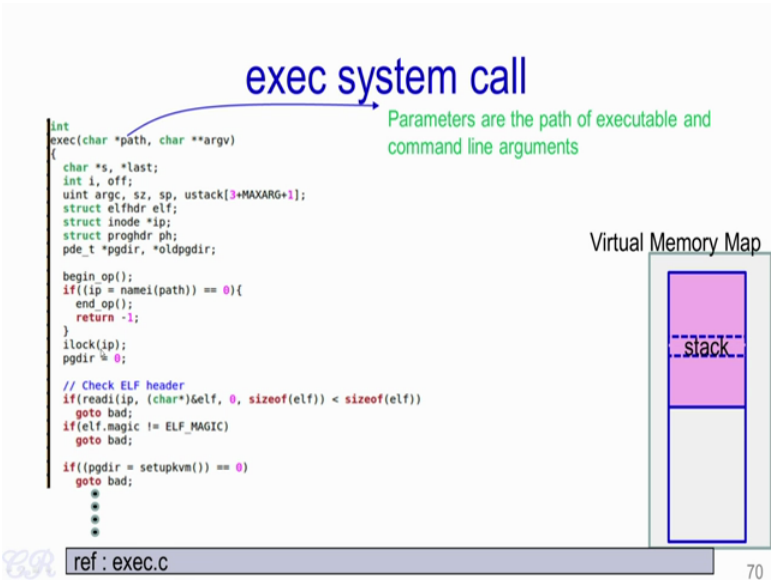
Next, we will look at the Program Header Contents So, program header also has several parameters such as the type of the program header, the offset, the virtual address essentially the virtual address where the segment needs to be loaded, and you have a p address offset which is essentially ignored.

(Refer Slide Time: 22:34)



So, in order to get the program headers for our hello world program, so we use readelf –l hello. So note that, we are giving the executable over here and not the object. So these are the parameters which are printed related to the program headers. So, essentially we have the type of the header, the offset, the virtual address where it needs to be loaded, physical address, the memory size and the flags.

(Refer Slide Time: 23:05)



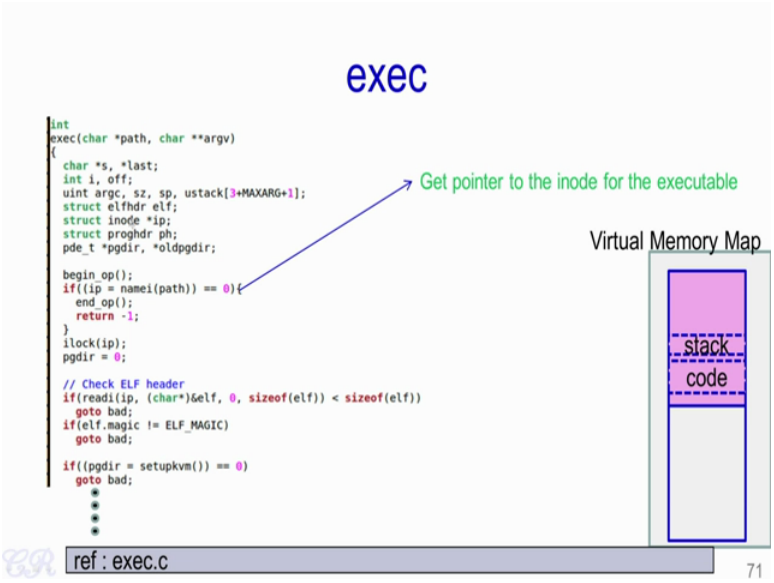
Now that we have some idea about ELF executable files and exec elf objects. Essentially how they are stored and the various components in the ELF images, we are equipped to

see how exec system call is implemented in the xv6 OS. So, this is actually listed over here (refer above image). And you could also look it up in the xv6 source code in this particular file, that exec.c. So, also shown in the slide is the virtual memory map. So, recollect that the exec system call gets invoked by the child process.

So first, there is a parent process which forks and in the child process, the exec system call that gets invoked. So, when the child process is created by forking, there is a virtual memory map which is created for the child process. And we as we have seen the top half or the top area regions of this virtual memory map comprises of the kernel code, while the lowest half comprises of the user code and data. Also we have seen that, during the process of forking there is the kernel stack which gets created. So, this kernel stack is specific for this child process (refer above image).

Now, we will see how as exec() function executes this virtual memory map changes. So, to start with let us look at the exec parameters which are taken (refer above image). So, in this case there are 2 parameters which are taken that is the path and the argv, so path specifies the path to the executable. So, in our example we have used /bin/ls. So, this would be specified in the path while argv is the parameter which you are passing to the particular program. So, for instance ls if you are taking could have arguments such as ls -l or ls -t and so on. So the -l and -t are arguments which are passed over here.

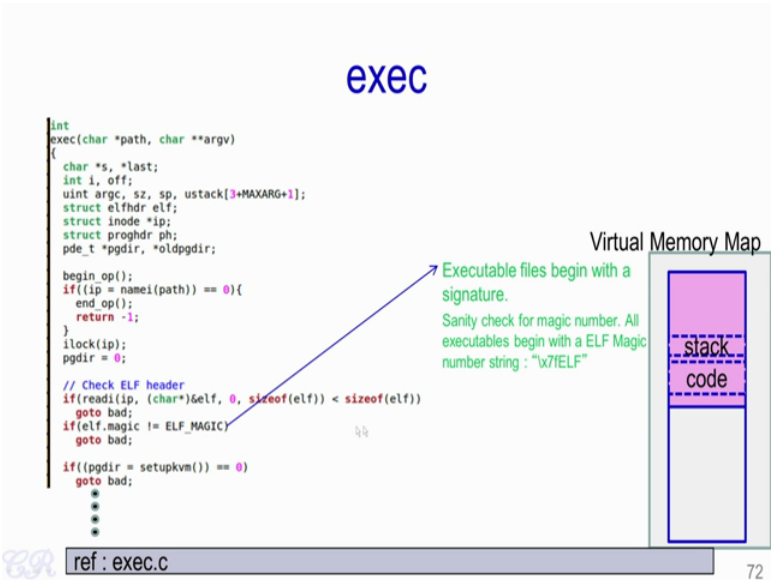
(Refer Slide Time: 25:16)



So the first thing that is done is that we get a pointer to the inode of the executable (refer above image if condition). Now inode is a metadata which has information about where the particular executable is stored on the secondary storage device such as the hard disk. So, for instance, in our particular case, where we are using /bin/ls, so this particular function namei() would return the inode for the ls executable.

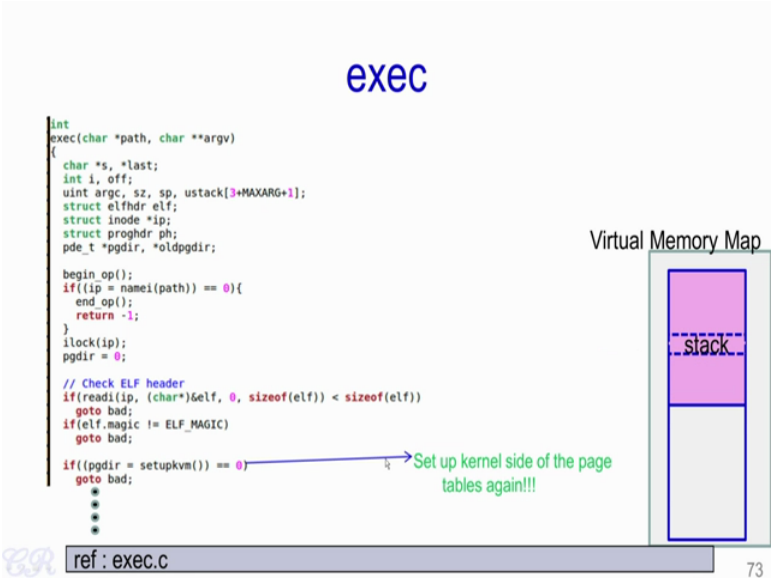
Next, what we are doing is we are using this function called readi(), which reads from that inode that is from the secondary storage device, the ELF header. So, we are reading the ELF header from this particular inode i.e ip. So, elf header or just mention as elf over here is defined over here as elf header (Check ELF header condition in above image). So, we are then looking up into this ELF header and checking the identifier, we are checking the magic number and we are verifying whether this magic number is indeed correct (refer above image).

(Refer Slide Time: 26:23)



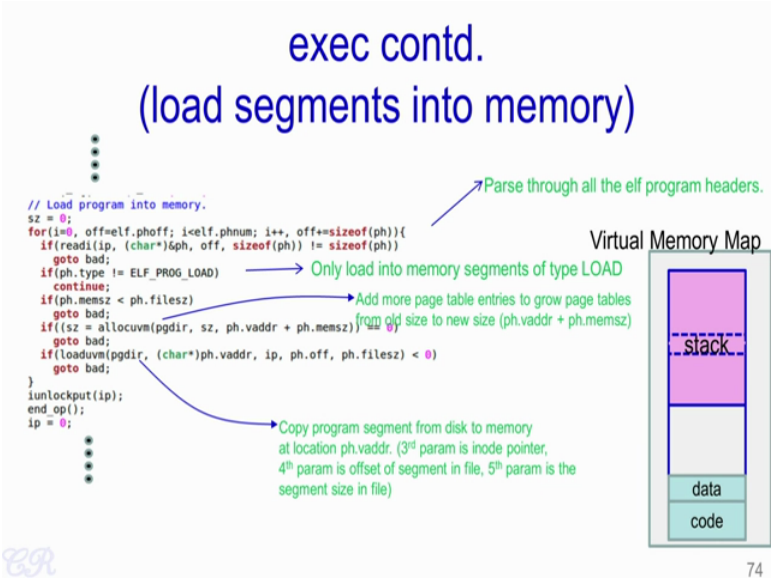
So, this essentially is a sanity check and this magic number for ELF should have this particular value that is, it should have \x7fELF (mentioned in above image). So, this is the 7f is the hexadecimal value, while ELF is the alphabets.

(Refer Slide Time: 26:46)



The next step is a call to the setupkvm() (refer above image), where we setup kernel side page tables again. So this essentially may not be required, but it is done in xv6, though it is not a necessity since, we have already done it during the fork process.

(Refer Slide Time: 27:07)

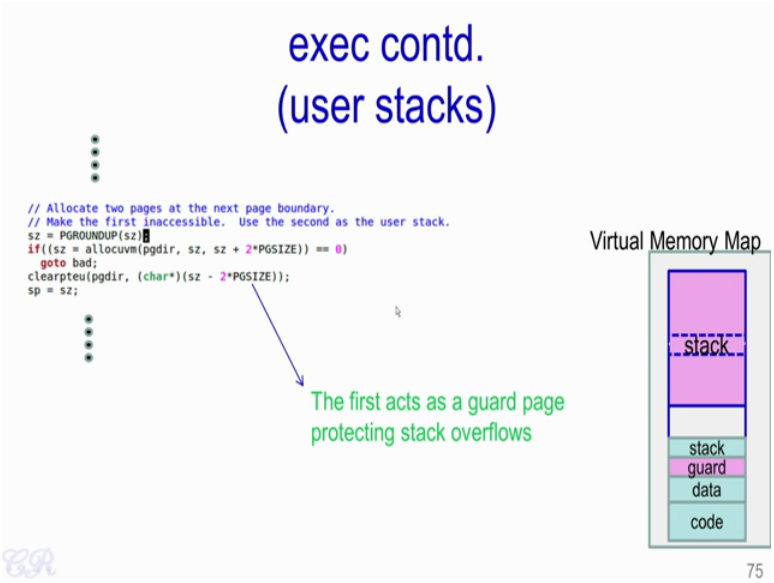


So this particular slide shows a continuation of the exec implementation (mentioned above image). So the next thing is we continue to read from the inode. We read various things like the program headers and we begin to load code and data from the ELF image which is present on the hard disk into RAM, and consequently we are actually filling up

the virtual address space corresponding to the code and data. So, this is done over here (refer all if conditions in above image).

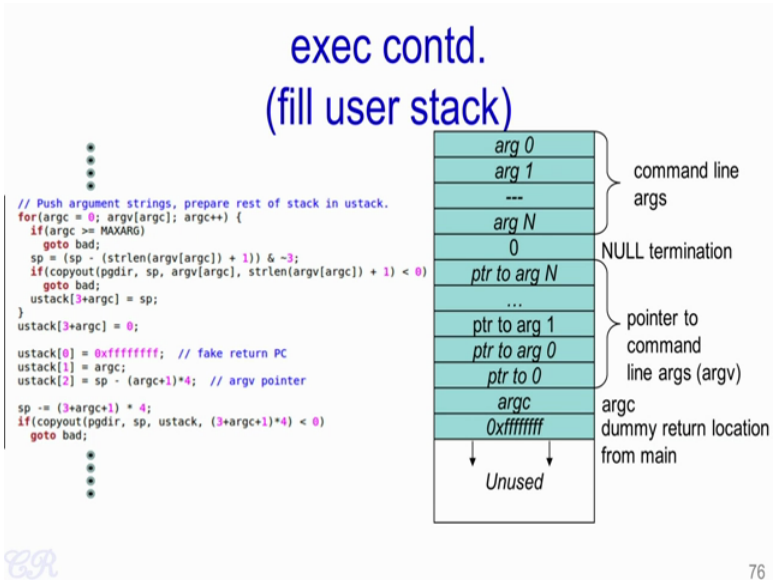
So I will not go more into details about this about how the various functions are used. But the basic idea is that, we are going to the hard disk looking into the inode corresponding to that particular executable and loading the code and the read only data into the physical memory map and we are also creating page table and page directory entries corresponding to that code and data. So, this is actually present, so therefore, you get this mapping present over here (refer above image).

(Refer Slide Time: 28:12)



The next step in the exec implementation is to create the stack for the user process. So, this stack (mentioned in above image) as opposed to the kernel stack is used by the code for storing of local variables as well as for function calls. So, in order to create this stack, we rather the exec implementation allocates 2 contiguous pages. So, it one is used for the stack while the other one is used as a guard page. So the guard page is made inaccessible; essentially this is used to protect against stack overflows. So, what does it means is that as we keep using the stack, the stack size keeps increasing and it would eventually hit the guard page; and as a result, we would know that the stack overflow has occurred.

(Refer Slide Time: 29:07)

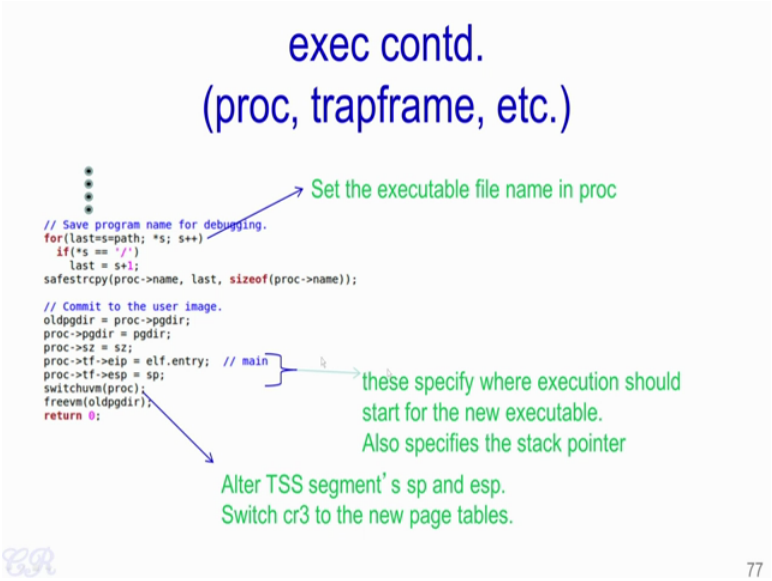


The next step in the exec implementation is to fill the user stack. So, essentially we have created the stack over here (refer above image) and now we are actually filling the stack. So we fill the stack with command line arguments. So, we know that any program that we write could take command line arguments and these arguments are actually filled into the stack.

So, we have like command line argument 0 to N, followed by a null termination string and then we have pointers to these arguments like pointer to argument 0, pointer to argument 1, and so on (mentioned in above image). So these pointer to arguments, forms the argv of your program. So, you know that a main function takes argc and argv, so these pointers from the argv of your programs input and after these argv is the argc, which is the number of such parameters and followed by 0xffffffff, there is a dummy return location for main.

Now, we have seen that we have actually created the code for the user process, we have the data for the user process and these two have actually been taken from the secondary storage device and loaded into physical RAM. And also a page directory and page table entries have been created thus we are able to see it in the virtual main memory map. And also we have created a stack for this user process and we have filled the stack with command line arguments, creating the argc and argv. The only thing next to do is to actually start executing the process.

(Refer Slide Time: 30:42)



So this is done by filling the trapframe for this current process with elf entry. So, essentially the tf eip that is the instruction pointer in the trapframe is stored with elf.entry i.e tp->eip = elf.entry, which essentially is a pointer to the main of the user program or main function of the user program. Similarly, the stack pointer is set to sp, so we are creating the trapframe esp is set to sp i.e tf->esp = sp. Now when this particular exec system call returns to the user process, the trapframe gets restored into the registers as a result the eip gets the value of the main address while the stack pointer the esp gets the value of the user space stack and therefore, execution will start from the main program. The stack pointer will have the pointer to the various arguments for argc and argv, which will then be used in the main program.

So with this, we come to the end of this video lecture. We had seen quite in detail about how xv6 operating system implements various system calls related to the process management such as the fork, wait, exec and the exit system call. So, in particular with the exec system call, we had seen how the various user space sections such as the code data and the stack gets created, and how the stack gets filled with command line arguments, and how execution of the user space process gets initiated.

Thank you.