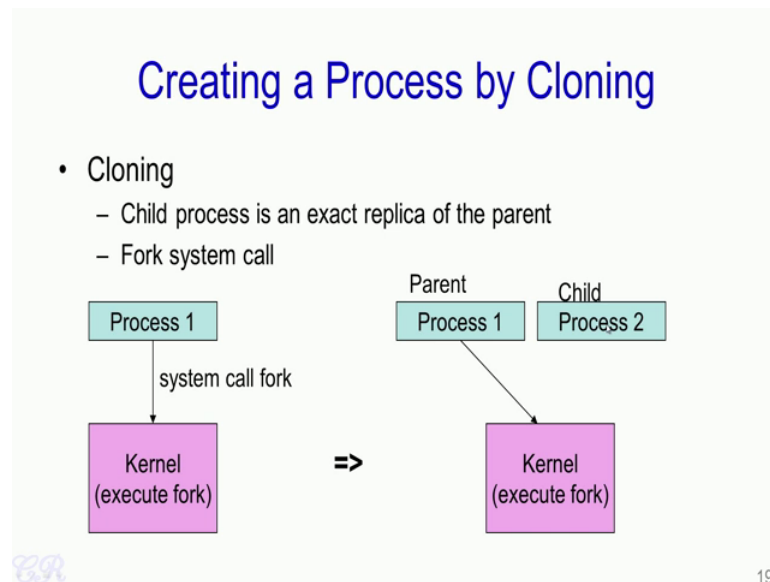


Introduction to Operating Systems
Prof. Chester Rebeiro
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

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.

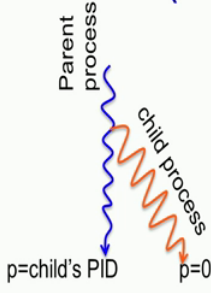
(Refer Slide Time: 00:27)



In order to create a process, operating systems use a technique called Cloning. When a process coincidentally 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)

Creating a Process by Cloning (using fork system call)



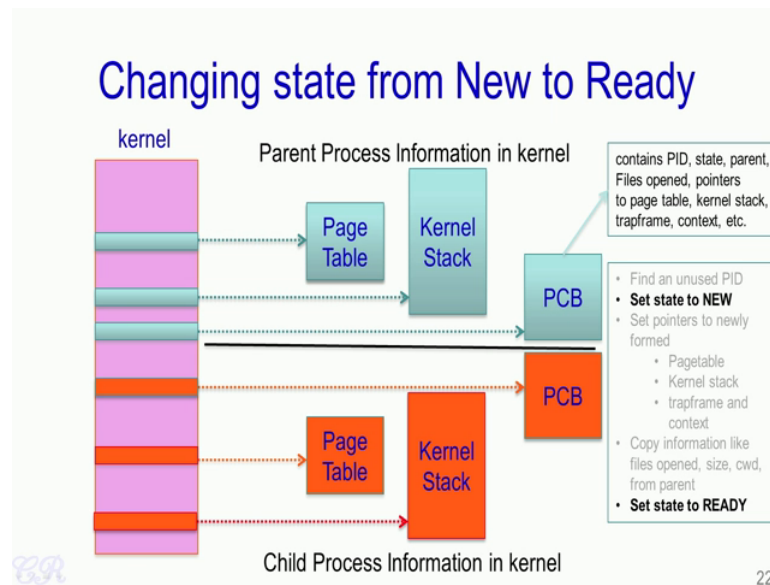
```
int p;
p = fork();
if (p > 0){
    printf("Parent : child PID = %d", p);
    p = wait();
    printf("Parent : child %d exited\n", p);
} else if (p == 0){
    printf("In child process");
    exit(0);
} else{
    printf("Error\n ");
}
```

20

Let us say that we have written this particular program which invokes the fork system call. The return from the fork system call is p , and p is defined as an integer, and it can take values of minus 1, 0 or something greater than 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 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 greater than 0; however, in the child process the value of p equal to 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 greater than 0, therefore the parent process would execute at these particular statements. However, in the child process, since the value of p equal to 0 the else if part of that is this green statements would be executed. Now if by chance the fork system call fails a value of minus 1 would be returned which would result in this printf error being printed onto the screen.

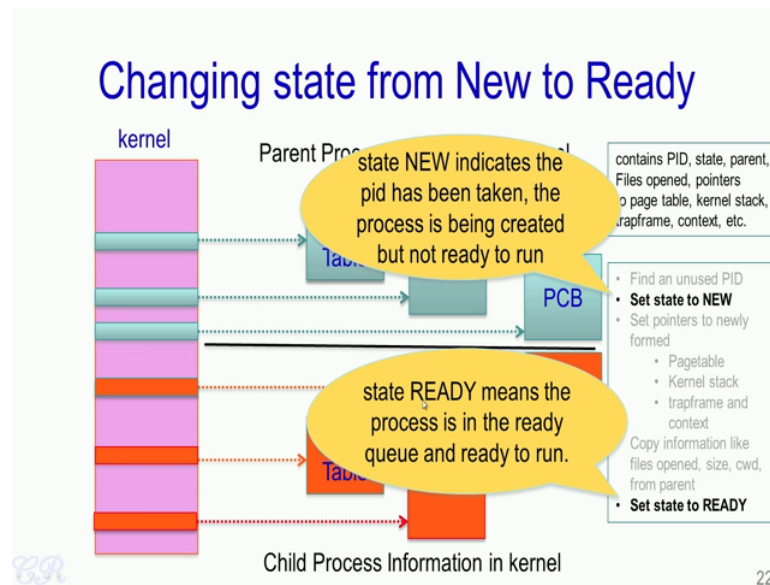
(Refer Slide Time: 02:48)



Now, let us see how this fork system call works inside the operating system. So, as we mentioned before, each process has three metadata stored in the kernel. So, one is the page table, second is the kernel stack, and third is an entry in the fork structure which is the process control block - PCB. So, when the fork system call executes, a copy of the page table is made. So, this corresponds to the page table of the new child that is being created.

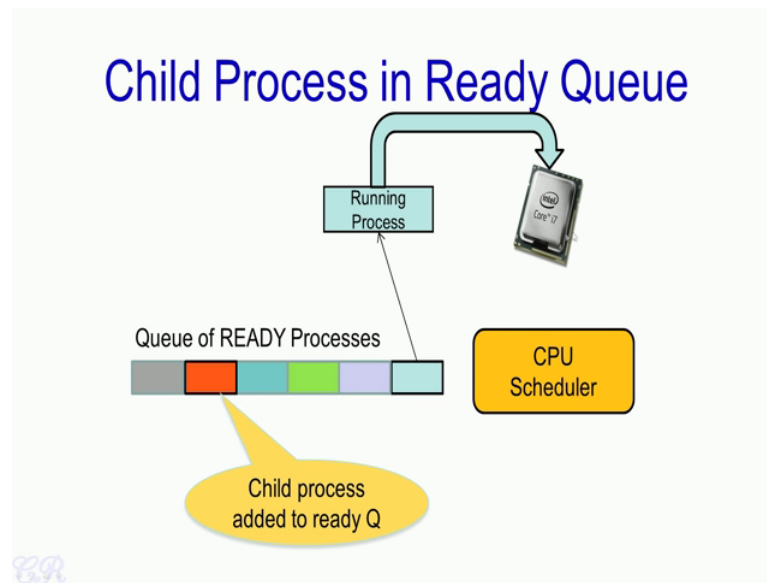
Similarly, the kernel stack of the parent process is duplicated as the child's kernel stack. Further a new PCB entry is created corresponding to the child's PCB entry. In order to do this, first an unused PID needs to be found by the operating system. Then a state of the newly formed process or the state of the child process is set to new. Then several pointers are set in the in the PCB such as the page table pointer that is the page table pointer will point to this particular page table, the kernel stack pointer would point to this one to this newly formed copied entry, and a pointers to trapframe and the context which are present within this kernel stack. Other information such as the files opened, the size of the process, the current working directory etcetera are also copied from the parents PCB into the child's PCB.

(Refer Slide Time: 04:32)



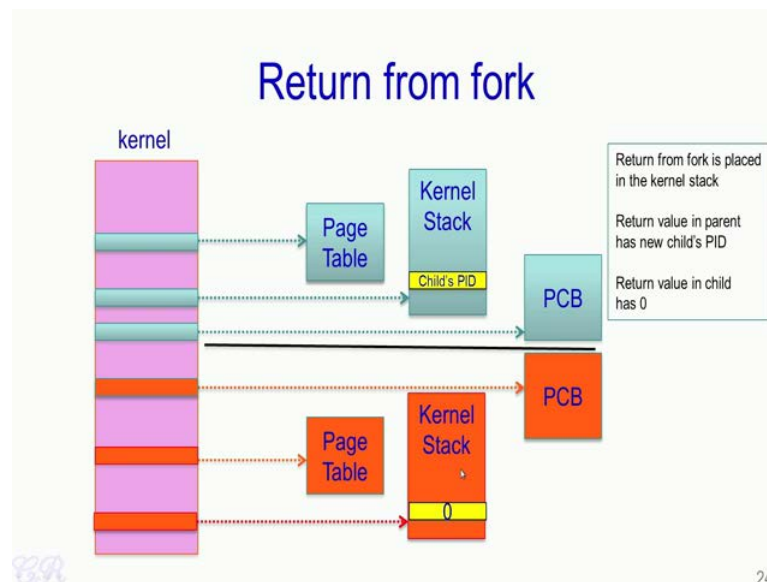
Finally just before returning, the operating system would set the state of the newly created child to ready. So, we see that at the start of execution of the fork, the child's state is set to new while towards the end of execution of the fork, the OS would set the state to ready. So, why do we need this intermediate state new? So, what does new signify and what does the ready signify. So, the new state indicates that the PID, the chosen PID over here 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 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)

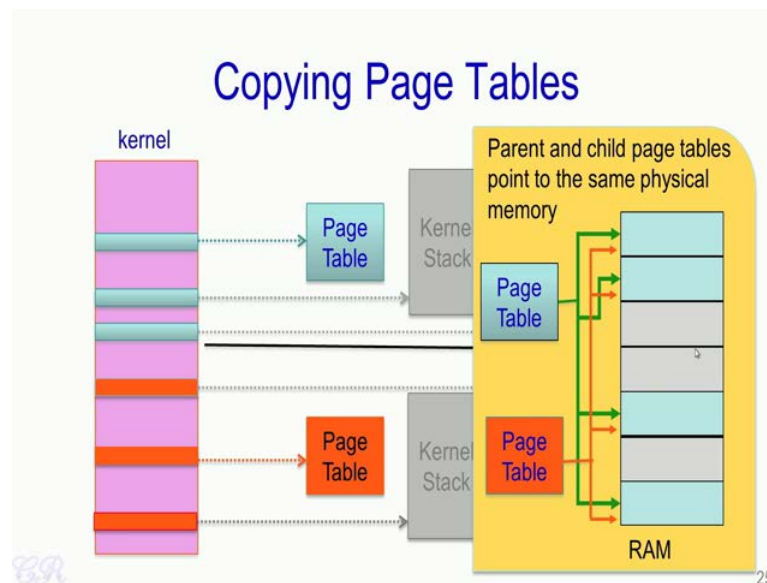


Now one important difference between the parent process and the child process with respect to the fork is that in the parent process fork returns the child's PID; while in the child's process, the fork system call would return 0. So, how does the operating system

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.

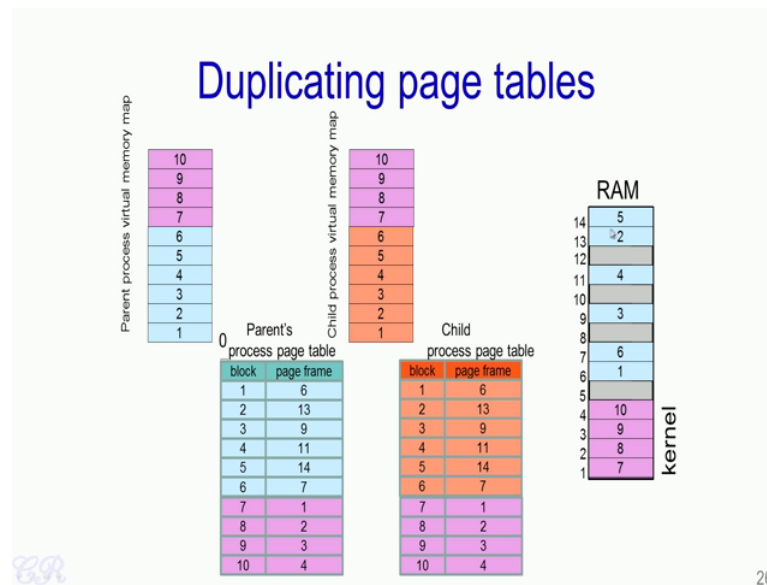
Further in the kernel stack of the child's process the return value is set to 0; thus when the system call returns in each of these cases, each process the parent process as well as the child process will get different values of the return type.

(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, and when fork is invoked a duplicate of this page table is created for the child. 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 the both the parent as well as the child page table are pointing to the same page frames in the ram. Let us look at this in more detail.

(Refer Slide Time: 07:46)



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 has its own page table. So, the child process page table is an exact replica of the parent's 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 is that we have two virtual memory maps, one for the 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)

Example

```
int i=23, pid;
pid = fork();
if (pid > 0){
    sleep(1);
    printf("parent : %d\n", i);
    wait();
} else{
    printf("child : %d\n", i);
}
```

Output

```
child : 23
Parent : 23
```

27

Let us look at this with an example. Suppose, we have return 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. 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 the 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 like this; both the children as well as the parent print the value of i as 23.

(Refer Slide Time: 09:49)

Example

```
int i=23, pid;
pid = fork();
if (pid > 0){
    sleep(1);
    printf("parent : %d\n", i);
    wait();
} else{
    i = i + 1;
    printf("child : %d\n", i);
}
```

Output

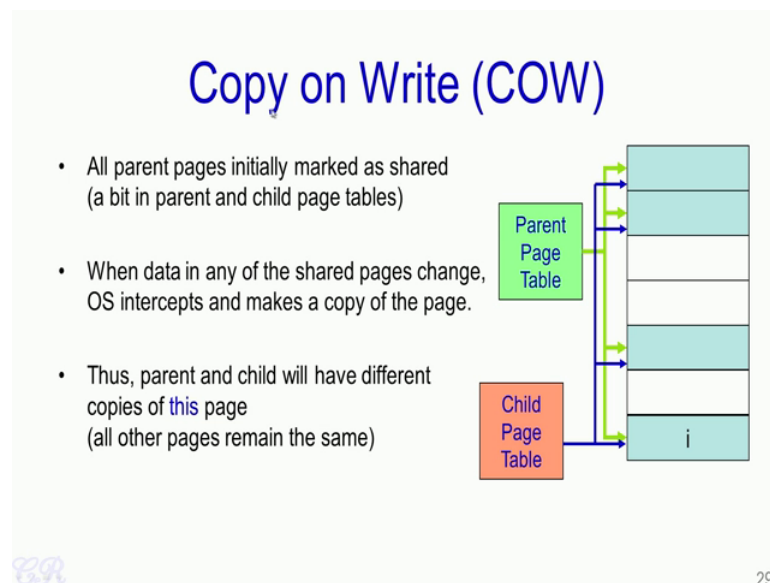
```
child : 24
Parent : 23
```

28

Now, let us modify this particular example and see what happens when we add this particular line. So, what we have done over here is that we have incremented i , i equal to i plus 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 equal to i plus 1 to be executed by the child and only then with this `printf` of the parent d 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 by i plus 1, so that is 23 plus 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 are happening.

(Refer Slide Time: 11:04)

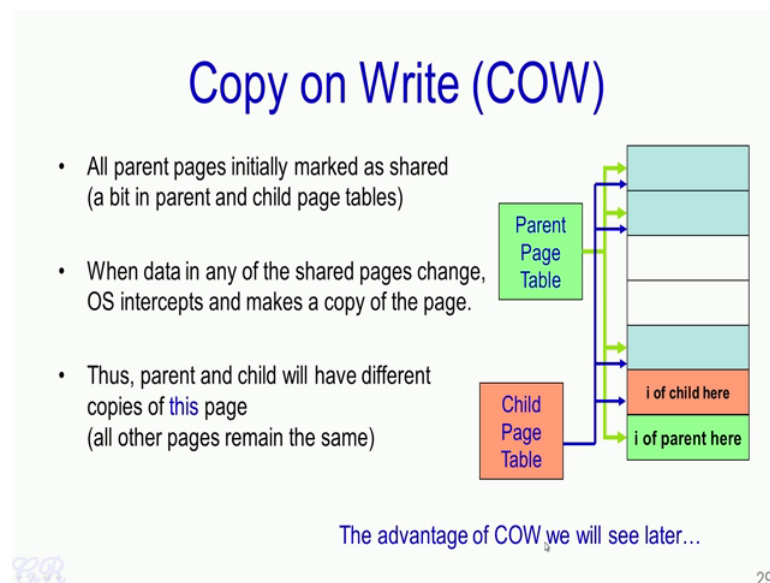


So, this phenomena occurs due to a technique known as copy on write that is implemented in the operating systems. So, when a `fork` system call is executed in the OS all parent pages are initially marked as shared. The shared bit is present in both the parent and child page tables. When data in any of 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, 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, 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 plus 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.

(Refer Slide Time: 12:25)



So, i of the child process would be present here and it would have the new updated value of 24 while the original i value of the parent would be present over here, 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, 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

Two step process
First fork and then exec

exec system call

- Find on hard disk the location of the 'a.out' executable
- Load on demand the pages required to execute a.out

```
int pid;

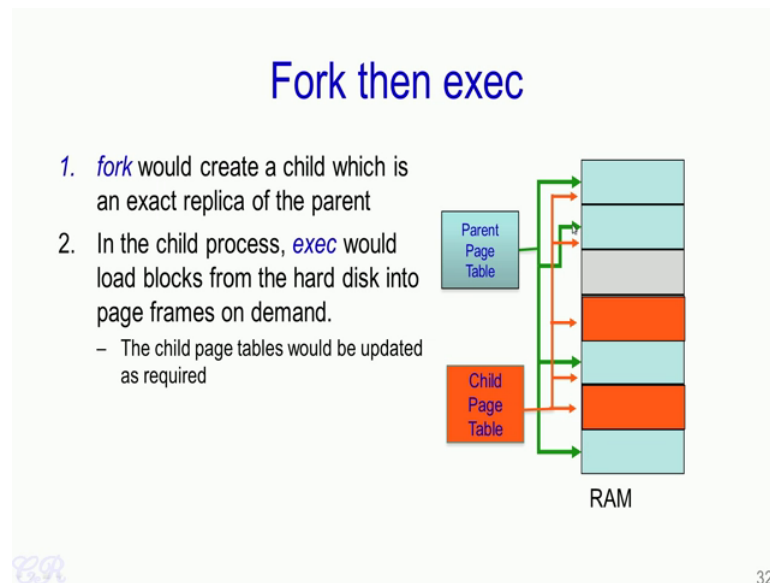
pid = fork();
if (pid > 0){
    pid = wait();
} else{
    execlp("./a.out", "", NULL);
    exit(0);
}
```

31

Executing a new program comprises of two steps; first 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. 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 call 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, the invoke a system call exec in this case it is a variant of exec known as exec l p 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 dot 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 dot out executable then it loads on demand the pages required to execute a dot out.

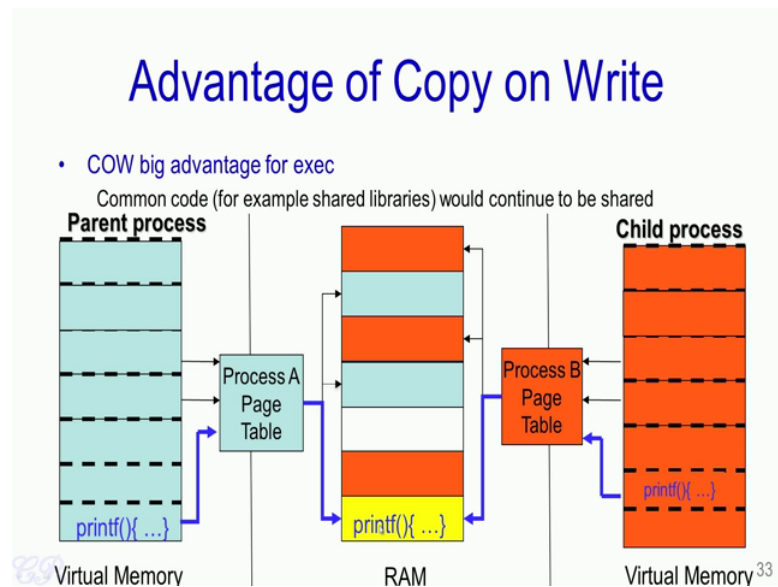
(Refer Slide Time: 14:28)



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 could 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 invoked, 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 processes page table would also be updated as required. So, we see two 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. However, the functionality which gets which is common to both the parent as well as the child still share the common pages, for instance like this.

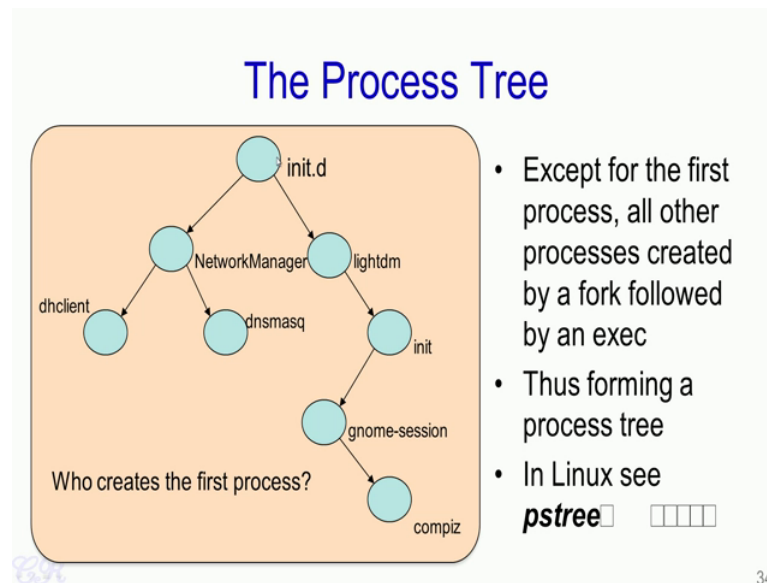
(Refer Slide Time: 15:54)



Let us see what the advantage of copy on write is. So, as we know most programs are returned 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 `f read`, `f scanf` and `printf` and so on. Since, our 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 with the new step page table to point to this particular page frame, which contains the `printf`.

(Refer Slide Time: 17:21)



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 of a node known as init dot d, and every subsequent in the node represents a process running in the system and is created from a previous process. For instance, the process compiz 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 no which in this case is known as inti dot d. If you are interested, you can actually look up or execute this particular command that is pstree from your (Refer Time: 18:13) 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

- Unix : **/sbin/init** (xv6 initcode.S)
 - Unlike the others, this is created by the kernel during boot
 - **Super parent.**
 - Responsible for forking all other processes
 - Typically starts several scripts present in **/etc/init.d** in Linux

BR

35

The first process in xv 6 the first process is present in initcode dot 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 slash sbin slash 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 this 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 this the first process would start several scripts present in slash etc slash init dot d.

(Refer Slide Time: 19:29)

```
#include "types.h"
#include "stat.h"
#include "user.h"
#include "fcntl.h"

char *argv[] = { "sh", 0 };

int
main(void)
{
    int pid, wpid;

    if(open("console", 0_RDWR) < 0){
        mknod("console", 1, 1);
        open("console", 0_RDWR);
    }
    dup(0); // stdout
    dup(0); // stderr

    for(;;){
        printf(1, "init: starting sh\n");
        pid = fork();
        if(pid < 0){
            printf(1, "init: fork failed\n");
            exit();
        }
        if(pid == 0){
            exec("sh", argv);
            printf(1, "init: exec sh failed\n");
            exit();
        }
        while((wpid=wait()) >= 0 && wpid != pid)
            printf(1, "zombie!\n");
    }
}
```

init.c

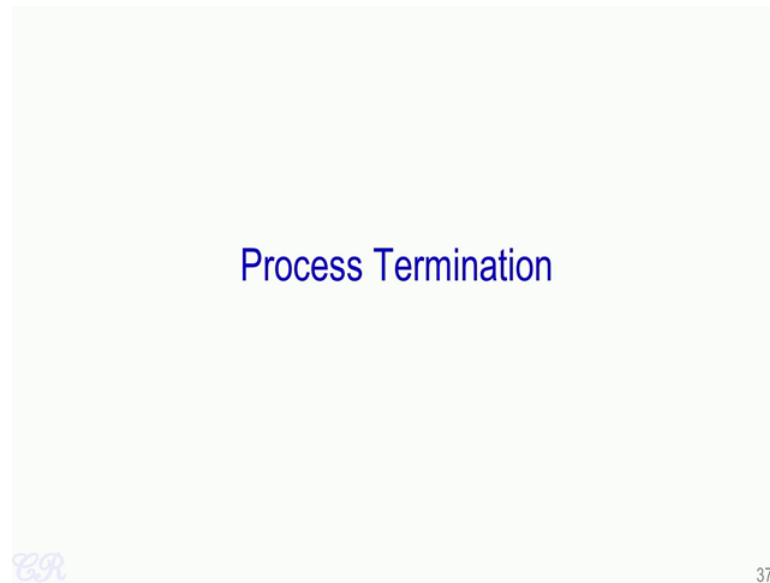
- forks and creates a shell (sh)

BR

36

Now let us look at the code `init dot 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 loop, this particular for loop, 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, it runs `exec sh` 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)

Exit System Call

exit()

- Called in child process
- Results in the process terminating
- The return status (0 here), is passed on to the parent.

```
int pid;

pid = fork();
if (pid > 0){
    pid = wait();
} else{
    execlp("./a.out", "", NULL);
    exit(0);
}
```

(this is a voluntary termination)

38

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

(Refer Slide Time: 21:03)

Involuntary Termination

- Involuntary : **kill(pid, signal)**
 - Signal can be sent by another process or by OS
 - pid is for the process to be killed
 - **signal** a signal that the process needs to be killed
 - Examples : SIGTERM, SIGQUIT (ctrl+), SIGINT (ctrl+c), SIGHUP

39

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, 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 in 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 in irrespective of what operation is being done.

(Refer Slide Time: 21:48)

Wait System Call

`wait()`

- Called in parent
- Parent goes to block state
 - Until one of its children exits
 - If no children executing, then -1 is returned
- Return status of child can be collected by `wait(&status)`

```
int pid;
pid = fork();
if (pid > 0){
    pid = wait();
} else{
    execlp("./a.out", "", NULL);
    exit(0);
}
```

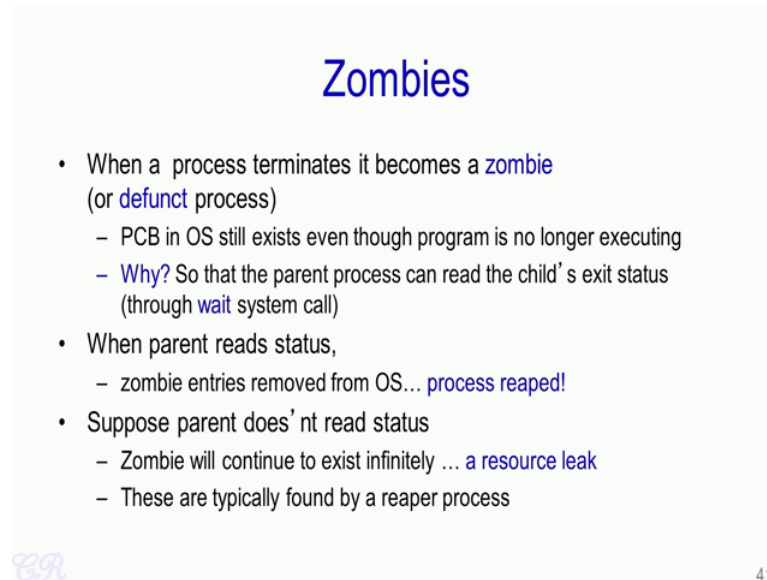
40

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 minus 1 is returned. So, let us go back to our example over here, where the parent has forked a child process; and in the parent process, it obtains a PID which is equal to the child processes PID.

So, this would result in the parent actually executing this statement which is a wait; and it would cause the parent process to be blocked until the child process has exited. When the child process exits, 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 processes 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 such as this. 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)



Zombies

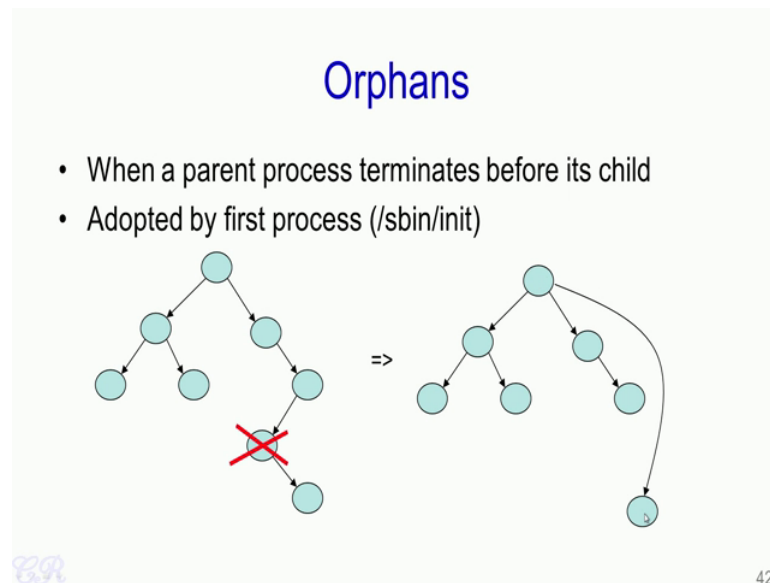
- When a process terminates it becomes a **zombie** (or **defunct** process)
 - PCB in OS still exists even though program is no longer executing
 - **Why?** So that the parent process can read the child's exit status (through **wait** system call)
- When parent reads status,
 - zombie entries removed from OS... **process reaped!**
- Suppose parent does' nt read status
 - Zombie will continue to exist infinitely ... **a resource leak**
 - These are typically found by a reaper process

41

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 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.

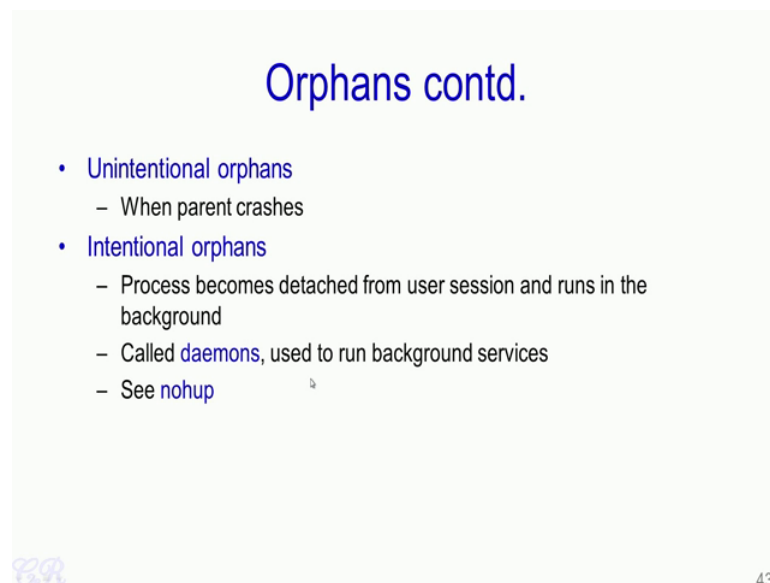
So, what happens if the parent does not read the child's status? In such a case, it will result in a resource leakage and the zombie entries in the operating system will continue to exist infinitely. So, these are eventually removed by a special process in the OS known as the reaper process. The reaper 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 and in particular this is a process with a parent over here. Now suppose this particular parent exits, while the child continues to execute then the child is known as an orphan. In such a case, the first process of the sbin init process will adopt the orphan child.

(Refer Slide Time: 25:20)



There are two 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 daemons 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)

exit() internals

- `init`, the first process, can never exit
- For all other processes on exit,
 1. Decrement the usage count of all open files
 - Close file if usage count is 0
 2. wakeup parent
 - If parent state is `sleeping`, make it `runnable`
 - Needed, cause parent may be sleeping due to a wait
 3. Make `init` adopt children of exited parents
 4. Set process state to `ZOMBIE`

note : page directory, kernel stack, not de-allocated here.
These are de-allocated by the parent process, allowing the parent to debug crashed children.

ref : `proc.c (exit) 2604` 44

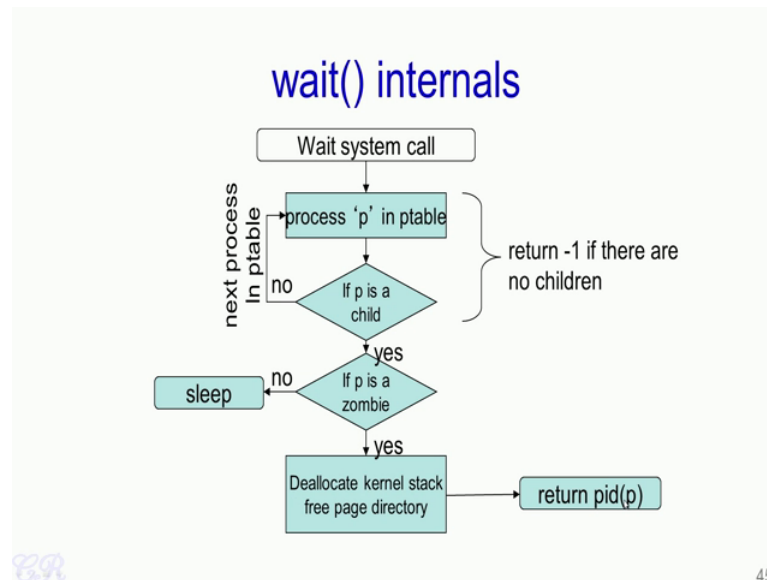
Now let us look at the exit system call from an operating system perspective. First we notice that the `/bin/init` that is the first process in the system can never exit. For all other processes, they can exit by either invoking the exit system call or can exit involuntarily by a signal. So, when such 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 make 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 current the exiting processes have 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 and the user space invoke the wait system call these following operations in green occurs in the operating system. So, first there is a loop 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.

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. 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 for exit system call.

Thank you.