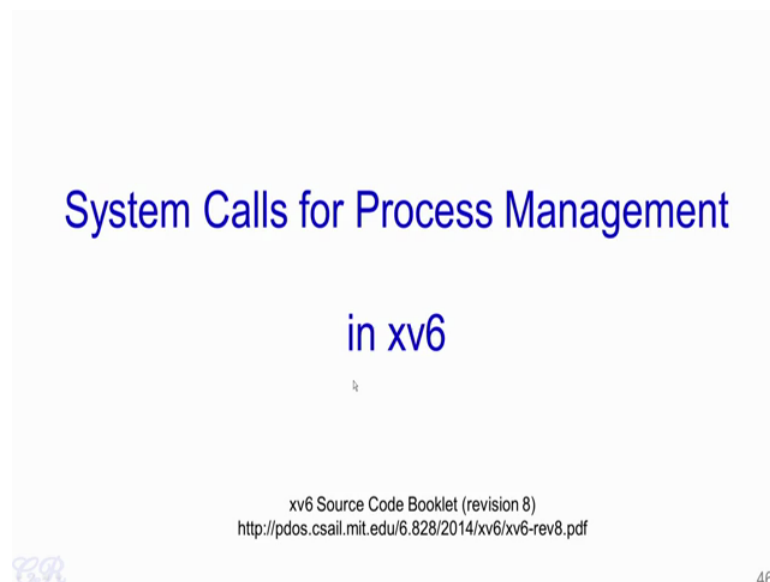


Introduction to Operating Systems
Prof. Chester Rebeiro
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

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. In this video, we will look at how these system calls are implemented within the operating system.

(Refer Slide Time: 00:38)



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. Please download the revision number 8, so that it matches with the particular video.

(Refer Slide Time: 01:00)

fork system call

- In parent
 - fork returns child pid
- In child process
 - fork returns 0
- Other system calls
 - Wait, returns pid of an exiting child

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

47

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. 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 processes pid value, thus these purple lines are what is going to be executed by the parent process.

(Refer Slide Time: 01:42)

fork and PCB structure

```
struct proc {
    uint sz;
    pde_t* pgdir;
    char *kstack;
    enum procstate state;
    int pid;
    struct proc *parent;
    struct trapframe *tf;
    struct context *context;
    void *chan;
    int killed;
    struct file *ofile[NOFILE];
    struct inode *cwd;
    char name[16];
};
```

Fork essentially creates a new PCB and fills it

```
struct {
    struct spinlock lock;
    struct proc proc[NPROC];
} ptable;
```

48

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. 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)

```

2553 int
2554 fork(void)
2555 {
2556     int i, pid;
2557     struct proc *np;
2558
2559     // Allocate process.
2560     if((np = allocproc()) == 0)
2561         return -1;
2562
2563     // Copy process state from p.
2564     if((np->pgdir = copyvm(proc->pgdir, proc->sz)) == 0){
2565         kfree(np->kstack);
2566         np->kstack = 0;
2567         np->state = UNUSED;
2568         return -1;
2569     }
2570     np->sz = proc->sz;
2571     np->parent = proc;
2572     *np->tf = *proc->tf;
2573
2574     // Clear %eax so that fork returns 0 in the child.
2575     np->tf->eax = 0;
2576
2577     for(i = 0; i < NOFILE; i++)
2578         if(proc->ofile[i])
2579             np->ofile[i] = filedup(proc->ofile[i]);
2580     np->cwd = idup(proc->cwd);
2581     safestrcpy(np->name, proc->name, sizeof(proc->name));
2582
2583     pid = np->pid;
2584
2585     // lock to force the compiler to emit the np->state write last.
2586     acquire(&ptable.lock);
2587     np->state = EMBRYO;
2588     release(&ptable.lock);
2589
2590     return pid;
2591 }
2592

```

fork

- Pick an UNUSED proc. Set pid. Set state to EMBRYO.
- Allocate kstack.
- Fill kstack with (1) the trapframe pointer, (2) trapret and (3) context

np is the proc pointer for the new process

49

The implementations of fork in xv6 are shown over here. So, this can be seen in the source code listing in line number 2554. The first thing you would notice over here is that we are declaring a pointer called np which is defined as a struct proc pointer. So, this is our 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 does that is what this function does is that it is going to parse through the p table 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)

```
2553 int
2554 fork(void)
2555 {
2556     int i, pid;
2557     struct proc *np;
2558
2559     // Allocate process.
2560     if((np = allocproc()) == 0)
2561         return -1;
2562
2563     // Copy process state from p.
2564     if((np->pgdir = copyvm(proc->pgdir, proc->sz)) == 0){
2565         kfree(np->kstack);
2566         np->kstack = 0;
2567         np->state = UNUSED;
2568         return -1;
2569     }
2570     np->sz = proc->sz;
2571     np->parent = proc;
2572     *np->tf = *proc->tf;
2573
2574     // Clear %eax so that fork returns 0 in the child.
2575     np->tf->eax = 0;
2576
2577     for(i = 0; i < NOFILE; i++)
2578         if(proc->ofile[i])
2579             np->ofile[i] = filedup(proc->ofile[i]);
2580     np->cwd = idup(proc->cwd);
2581
2582     safestrcpy(np->name, proc->name, sizeof(proc->name));
2583
2584     pid = np->pid;
2585
2586     // lock to force the compiler to emit the np->state write last.
2587     acquire(&table.lock);
2588     np->state = EUNBABLE;
2589     release(&table.lock);
2590
2591     return pid;
2592 }
```

fork

Copy page directory from the parent process (proc->pgdir) to the child process (np->pgdir)

Parent process, is the process invoking the fork system call, it PCB is pointed to by proc

50

The next step in the fork implementation is this call to this function called copy u v m. So, essentially what the copy u v m function does is that it copies the page directory from the parent process to the child process. This copy u v m takes two parameters; it take the parent page directory which is represented by proc point pgdir; and the parent size that is represented by s z, and what is returned by this function is a pointer to the new processes page directory.

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, and the value of n p k stack is set to 0 and the state is set back set to unused again. Remember that allocproc had set the state to EMBRYO, now over here the state is set to unused, and then fork is going to return with minus 1. So, this minus 1 is sent back to the user process that is the process which had invoked forked.

(Refer Slide Time: 05:15)

```
2553 int
2554 fork(void)
2555 {
2556     int i, pid;
2557     struct proc *np;
2558
2559     // Allocate process.
2560     if((np = allocproc()) == 0)
2561         return -1;
2562
2563     // Copy process state from p.
2564     if((np->ppdir = copyvm(proc->ppdir, proc->sz)) == 0){
2565         kfree(np->kstack);
2566         np->kstack = 0;
2567         np->state = UNUSED;
2568         return -1;
2569     }
2570     np->sz = proc->sz;
2571     np->parent = proc;
2572     *np->tf = *proc->tf;
2573
2574     // Clear %eax so that fork returns 0 in the child.
2575     np->tf->eax = 0;
2576
2577     for(i = 0; i < NOFILE; i++)
2578         if(proc->ofile[i])
2579             np->ofile[i] = filedup(proc->ofile[i]);
2580     np->cwd = idup(proc->cwd);
2581
2582     safestrcpy(np->name, proc->name, sizeof(proc->name));
2583
2584     pid = np->pid;
2585
2586     // lock to force the compiler to emit the np->state write last.
2587     acquire(&table.lock);
2588     np->state = EUNNABLE;
2589     release(&table.lock);
2590
2591     return pid;
2592 }
```

fork

Set size of np same as that of parent
Set parent of np
Copy trapframe from parent to child

BR

51

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 on to 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)

fork system call

- In parent
 - fork returns child pid
- In child process
 - fork returns 0
- Other system calls
 - Wait, returns pid of an exiting child

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

BR

47

If we go back over here 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.

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; thus when fork returns in the child process, the child process will execute from this point. 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 greater than 0. So, we will next see how this is achieved.

(Refer Slide Time: 06:49)

```
2553 int
2554 fork(void)
2555 {
2556     int i, pid;
2557     struct proc *np;
2558
2559     // Allocate process.
2560     if((np = allocproc()) == 0)
2561         return -1;
2562
2563     // Copy process state from p.
2564     if((np->pgdir = copyvm(proc->pgdir, proc->vaz)) == 0){
2565         kfree(np->kstack);
2566         np->kstack = 0;
2567         np->state = UNUSED;
2568         return -1;
2569     }
2570     np->sz = proc->sz;
2571     np->parent = proc;
2572     *np->tf = *proc->tf;
2573
2574     // Clear %eax so that fork returns 0 in the child.
2575     np->tf->eax = 0;
2576
2577     for(i = 0; i < NOFILE; i++)
2578         if(proc->ofile[i])
2579             np->ofile[i] = filedup(proc->ofile[i]);
2580     np->cwd = idup(proc->cwd);
2581
2582     safestrcpy(np->name, proc->name, sizeof(proc->name));
2583
2584     pid = np->pid;
2585
2586     // lock to force the compiler to emit the np->state write last.
2587     acquire(&table.lock);
2588     np->state = RUNNABLE;
2589     release(&table.lock);
2590
2591     return pid;
2592 }
```

fork

In child process, set eax register in trapframe to 0. This is what fork returns in the child process

52

Essentially in the folk implementation, we see that in the trapframe of the new process the value of eax is set to 0. So, this will ensure that in the child process, the return value of folk 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)

```
2553 int
2554 fork(void)
2555 {
2556     int i, pid;
2557     struct proc *np;
2558
2559     // Allocate process.
2560     if((np = allocproc()) == 0)
2561         return -1;
2562
2563     // Copy process state from p.
2564     if((np->pgdir = copyvm(proc->pgdir, proc->sz)) == 0){
2565         kfree(np->kstack);
2566         np->kstack = 0;
2567         np->state = UNUSED;
2568         return -1;
2569     }
2570     np->sz = proc->sz;
2571     np->parent = proc;
2572     *np->tf = *proc->tf;
2573
2574     // Clear %eax so that fork returns 0 in the child.
2575     np->tf->eax = 0;
2576
2577     for(i = 0; i < NOFILE; i++)
2578         if(proc->ofile[i])
2579             np->ofile[i] = filedup(proc->ofile[i]);
2580     np->cwd = idup(proc->cwd);
2581     safestrcpy(np->name, proc->name, sizeof(proc->name));
2582
2583     pid = np->pid;
2584
2585     // lock to force the compiler to emit the np->state write last.
2586     acquire(&table.lock);
2587     np->state = EMMBRYO;
2588     release(&table.lock);
2589
2590     return pid;
2591 }
2592 }
```

fork

Other things... copy file pointer from parent, cwd, executable name

53

In this part of the folk implementation, 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)

```
2553 int
2554 fork(void)
2555 {
2556     int i, pid;
2557     struct proc *np;
2558
2559     // Allocate process.
2560     if((np = allocproc()) == 0)
2561         return -1;
2562
2563     // Copy process state from p.
2564     if((np->pgdir = copyvm(proc->pgdir, proc->sz)) == 0){
2565         kfree(np->kstack);
2566         np->kstack = 0;
2567         np->state = UNUSED;
2568         return -1;
2569     }
2570     np->sz = proc->sz;
2571     np->parent = proc;
2572     *np->tf = *proc->tf;
2573
2574     // Clear %eax so that fork returns 0 in the child.
2575     np->tf->eax = 0;
2576
2577     for(i = 0; i < NOFILE; i++)
2578         if(proc->ofile[i])
2579             np->ofile[i] = filedup(proc->ofile[i]);
2580     np->cwd = idup(proc->cwd);
2581     safestrcpy(np->name, proc->name, sizeof(proc->name));
2582
2583     pid = np->pid;
2584
2585     // lock to force the compiler to emit the np->state write last.
2586     acquire(&table.lock);
2587     np->state = EMMBRYO;
2588     release(&table.lock);
2589
2590     return pid;
2591 }
2592 }
```

fork

Child process is finally made runnable

54

Now that the proc structure for the child process is completely filled, the state is switched from EMMBRYO to runnable. So, setting the state to runnable would imply that a scheduler could actually select this child process and allocate it with the CPU. Thus the child process would be able to run and execute code on the CPU.

(Refer Slide Time: 07:58)

```
2553 int
2554 fork(void)
2555 {
2556     int i, pid;
2557     struct proc *np;
2558
2559     // Allocate process.
2560     if((np = allocproc()) == 0)
2561         return -1;
2562
2563     // Copy process state from p.
2564     if((np->ppdir = copyvm(proc->ppdir, proc->sz)) == 0){
2565         kfree(np->kstack);
2566         np->kstack = 0;
2567         np->state = UNUSED;
2568         return -1;
2569     }
2570     np->sz = proc->sz;
2571     np->parent = proc;
2572     *np->tf = *proc->tf;
2573
2574     // Clear %eax so that fork returns 0 in the child.
2575     np->tf->eax = 0;
2576
2577     for(i = 0; i < NOFILE; i++)
2578         if(proc->ofile[i])
2579             np->ofile[i] = filedup(proc->ofile[i]);
2580     np->cwd = idup(proc->cwd);
2581
2582     safestrcpy(np->name, proc->name, sizeof(proc->name));
2583
2584     pid = np->pid;
2585
2586     // lock to force the compiler to emit the np->state write last.
2587     acquire(&table.lock);
2588     np->state = EUNBABLE;
2589     release(&table.lock);
2590
2591     return pid;
2592 }
```

fork

Parent process returns the pid of the child

55

The return from the fork implementation is pid, the value of pid is set over here. So, essentially the pid is np pid that this is the child processes pid value. And this pid value 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 in the child process, fork would return with the value of 0.

(Refer Slide Time: 08:35)

Register modifications w.r.t. parent

Registers modified in child process

- %eax = 0 so that pid = 0 in child process
- %eip = *forkret* so that child exclusively executes function *forkret*

56

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 two registers. As we have seen before,

one is the eax register. So, in the child process, the eax register is set to 0, so that when fork returns, it returns with the value of zero 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 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 as you can see.

(Refer Slide Time: 09:25)

exit internals

- **init**, the first process, can never exit
- For all other processes on exit,
 1. Decrement the usage count of all open files
 - If usage count is 0, close file
 2. Drop reference to in-memory inode
 3. wakeup parent
 - If parent state is **sleeping**, make it **runnable**
 - Needed, cause parent may be sleeping due to a wait
 4. Make **init** adopt children of exited process
 5. Set process state to **ZOMBIE**
 6. Force context switch to scheduler

note : page directory, Kernel stack, not deallocated here

ref : proc.c (exit) 2604

57

Now, we will recall the exit system call internals. So, when a exit system call gets executed, these are the six things 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. Second, there is a drop in reference for all in memory i nodes.

Third, there is a wake up 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 fourth 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)

Wait system call

- Invoked in parent parent
- Parent 'waits' until child exits

```
int pid;

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

58

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 minus 1. Now we will see the internals of the wait system call.

(Refer Slide Time: 11:30)

```
int
wait(void)
{
    struct proc *p;
    int havekids, pid;

    acquire(&ptable.lock);
    for(;;){
        // Scan through table looking for zombie children.
        havekids = 0;
        for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
            if(p->parent != proc)
                continue;
            havekids = 1;
            if(p->state == ZOMBIE){
                // Found one.
                pid = p->pid;
                kfree(p->kstack);
                p->kstack = 0;
                freevm(p->pgdir);
                p->state = UNUSED;
                p->pid = 0;
                p->parent = 0;
                p->name[0] = 0;
                p->killed = 0;
                release(&ptable.lock);
                return pid;
            }
        }
        // No point waiting if we don't have any children.
        if(!havekids || proc->killed){
            release(&ptable.lock);
            return -1;
        }
        // Wait for children to exit. (See wakeup call in proc_exit.)
        sleep(proc, &ptable.lock); //DOC: wait-sleep
    }
}

```

wait

ref: proc.c 59

This is the implementation of the wait system call in the xv6 operating system. So, this listing is obtained from the proc dot c file of the xv6 source code. Essentially, you would see that it has an infinite loop by this that is for semicolon semicolon which starts over here, and ends at this point. So, within this particular infinite loop, there is an inner for loop which is from here to here. So, essentially this inner for loop passes through the p table. So, recollect that the p table is an array of procs; and each and every process which is in some state in the xv6 OS has an entry in the p table. So, by passing through all elements of the p table, this particular loop will be able to check every process that is present in the xv6 OS in this particular instant.

The first check that it does is to find out whether the current proc is the parent of this particular entry in the p table, which is p. So, it is going to find out if the current proc which has invoked the wait is the parent of p. If it is not the parent then it just continues; otherwise it comes over here. So, at this particular point, in the implementation, we 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. 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 set the state to unused set pid to 0 and so on. The return is that this point, 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 processes pid value.

(Refer Slide Time: 14:00)

The image shows a slide titled "wait" with the C code for the `wait()` function from `proc.c`. The code is as follows:

```
int
wait(void)
{
    struct proc *p;
    int havekids, pid;

    acquire(&table.lock);
    for(;;){
        // Scan through table looking for zombie children.
        havekids = 0;
        for(p = table.proc; p < &table.proc[NPROC]; p++){
            if(p->parent != proc)
                continue;
            havekids = 1;
            if(p->state == ZOMBIE){
                // Found one.
                pid = p->pid;
                kfree(p->kstack);
                p->kstack = 0;
                freevm(p->pgdir);
                p->state = UNUSED;
                p->pid = 0;
                p->parent = 0;
                p->name[0] = 0;
                p->killed = 0;
                release(&table.lock);
                return pid;
            }
        }

        // No point waiting if we don't have any children.
        if(!havekids || proc->killed){
            release(&table.lock);
            return -1;
        }

        // Wait for children to exit. (See wakeup call in proc_exit.)
        sleep(proc, &table.lock); //DOC: wait-sleep
    }
}
```

Annotations on the slide include:

- A blue word "wait" in the top right corner.
- A light blue callout box with a border pointing to the `kfree(p->kstack);` and `freevm(p->pgdir);` lines, containing the text: "note : page directory, kernel stack, deallocated here ... allows parent to peek into exited child's process".
- A small grey box at the bottom right containing the text "ref : proc.c".
- The number "60" in the bottom right corner.

So, you may have noticed two things; first the kernel stack of the exiting child is cleared or rather is freed at this particular; second the page directory corresponding to the exiting child is freed at this particular statement. So, freeing these 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)

```
int
wait(void)
{
    struct proc *p;
    int havekids, pid;

    acquire(&table.lock);
    for(;;){
        // Scan through table looking for zombie children.
        havekids = 0;
        for(p = table.proc; p < &table.proc[NPROC]; p++){
            if(p->parent != proc)
                continue;
            havekids = 1;
            if(p->state == ZOMBIE){
                // Found one.
                pid = p->pid;
                kfree(p->kstack);
                p->kstack = 0;
                freevm(p->pgdir);
                p->state = UNUSED;
                p->pid = 0;
                p->parent = 0;
                p->name[0] = 0;
                p->killed = 0;
                release(&table.lock);
                return pid;
            }
        }
        // No point waiting if we don't have any children.
        if(!havekids || proc->killed){
            release(&table.lock);
            return -1;
        }
        // Wait for children to exit. (See wakeup call in proc_exit.)
        sleep(proc, &table.lock); //DOC: wait-sleep
    }
}
```

wait

ref: proc.c 61

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

(Refer Slide Time: 15:10)

```
int
wait(void)
{
    struct proc *p;
    int havekids, pid;

    acquire(&table.lock);
    for(;;){
        // Scan through table looking for zombie children.
        havekids = 0;
        for(p = table.proc; p < &table.proc[NPROC]; p++){
            if(p->parent != proc)
                continue;
            havekids = 1;
            if(p->state == ZOMBIE){
                // Found one.
                pid = p->pid;
                kfree(p->kstack);
                p->kstack = 0;
                freevm(p->pgdir);
                p->state = UNUSED;
                p->pid = 0;
                p->parent = 0;
                p->name[0] = 0;
                p->killed = 0;
                release(&table.lock);
                return pid;
            }
        }
        // No point waiting if we don't have any children.
        if(!havekids || proc->killed){
            release(&table.lock);
            return -1;
        }
        // Wait for children to exit. (See wakeup call in proc_exit.)
        sleep(proc, &table.lock); //DOC: wait-sleep
    }
}
```

wait

ref: proc.c 62

If 'p' is in fact a child of proc and is not a ZOMBIE then block the current process

So, execution comes to this particular line that is the sleep 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 exiting child.

(Refer Slide Time: 15:28)

Executing a Program (exec system call)

- exec system call
 - Load a program into memory and then execute it
 - Here 'ls' executed.

```
int pid;

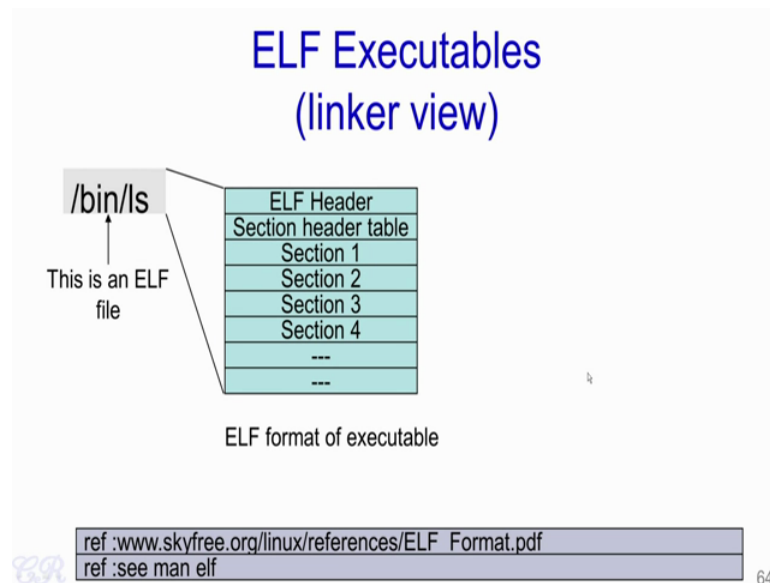
pid = fork();
if (pid > 0){
    pid = wait();
} else{
    execip("ls", "", NULL);
    exit(0);
}
```

63

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, 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.

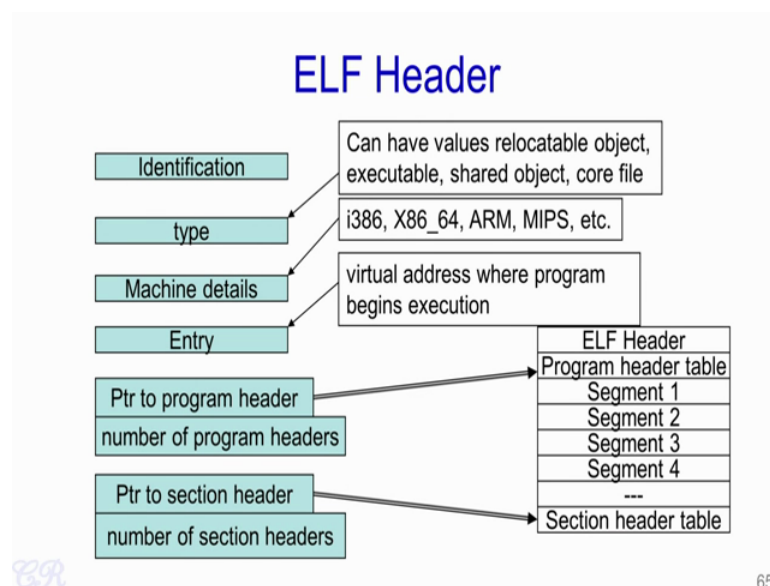
Now, 'ls' is an executable. It has a current particular format known as the ELF format or executable linker format. So, this particular format is what is interpreted internally by the exec system call within the operating system and this format is essentially understood and used to load ls from the hard 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 - slash bin slash ls, this is an ELF executable. So, it has a format has shown over here, so at least this is part of the format, and more details about the format can be obtained from these particular references. 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 are 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. 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. Second, there is the type so a type would tell that the type of this ELF file, so if 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, besides 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)

Hello World's ELF Header

```
#include <stdio.h>

int main(){
    char str[] = "Hello World\n";
    printf("%s", str);
}
```

```
ptitplex:~/tmp$ readelf -h hello.o
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                                ELF64
  Data:                                      2's complement, little endian
  Version:                               1 (current)
  OS/ABI:                                 UNIX - System V
  ABI Version:                           0
  Type:                                    REL (Relocatable file)
  Machine:                                Advanced Micro Devices X86-64
  Version:                                0x1
  Entry point address:                    0x0
  Start of program headers:               0 (bytes into file)
  Start of section headers:               368 (bytes into file)
  Flags:                                   0x0
  Size of this header:                    64 (bytes)
  Size of program headers:                0 (bytes)
  Number of program headers:              0
  Size of section headers:                64 (bytes)
  Number of section headers:              13
  Section header string table index:      10
```

```
$ gcc hello.c -c
$ readelf -h hello.o
```

66

Let us first start with an example. Let us take this particular example of our hello world program written in C, and we compile it with the minus C option. So, when you say g c c hello dot c minus c, it creates the object file hello dot o. So, this is an ELF object with then uses a utility readelf with an option minus h with the object file hello dot o. The minus h option would print the ELF header. So, this is the ELF header, 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, this in this particular example, since we used a dot using a hello dot o that is 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 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 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. The pointer to section headers link 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)

Section Headers

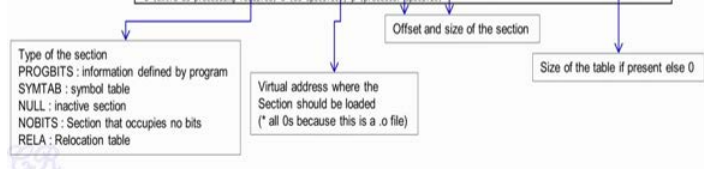
- Contains information about the various sections

\$ readelf -S hello.o

```

There are 13 section headers, starting at offset 0x178:
Section Headers:
 [Nr] Name      Type            Address           Offset           Size              Entsize          Flags    Link    Info    Align
  [ 1] .text     PROGBITS        0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000 AX      0      0      0
  [ 2] .rela.text RELA             0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000 11      0      1
  [ 3] .data     PROGBITS        0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000 WA      0      0      1
  [ 4] .bss      NOBITS          0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000 WA      0      0      1
  [ 5] .rodata   NOBITS          0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000 A       0      0      1
  [ 6] .comment  PROGBITS        0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000 MD      0      0      1
  [ 7] .note.GNU-stack NOBITS         0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000 0      0      0
  [ 8] .eh_frame PROGBITS        0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000 A       0      0      1
  [ 9] .rela.eh_frame RELA             0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000 A      11      0      8
 [10] .shstrtab STRTAB          0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000 0      0      1
 [11] .symtab   SYMTAB         0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000 12      0      8
 [12] .strtab   STRTAB          0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000 0      0      1

```

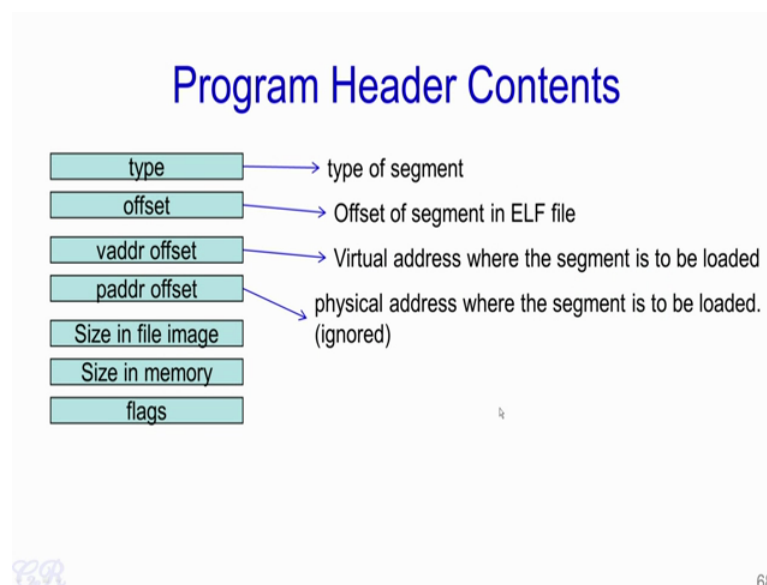


Now, we will look more into detail about the section headers, so in order to get a listing of the section headers you could use read ELF minus s hello dot o, which will print all the section headers present in the relocatable object hello dot o. So, this is actually shown

over here. 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 second column over here is the name of the sections, while the third column gives you the type of the sections.

The type could be one of these, these are the progbits which is information defined by the program, symtab which is a symbol table, null or nobits essentially is the section which occupies no bits, and rel a which is a relocatable table. Then we have here the address, which is a virtual address for where the section should be loaded; in this particular case, it is all 0s because it is a dot o file it is an object file and it could be relocated. Then here you have the offset and the size of the sections, while here 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.

see how exec system call is implemented in the xv6 OS. So, this is actually listed over here. And you could also look it up in the xv6 source code in this particular file that exec dot 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.

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 that 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.

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. So, in this case, there are two 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 slash bin slash l s So, this would be specified in the path while argv is the parameter which you are passing to the particular program. So, for instance l s if you are taking could have arguments such as l s minus l or l s minus t and so on and so forth. The minus l and minus t are arguments which are passed over here.

(Refer Slide Time: 25:16)

exec

```
int
exec(char *path, char **argv)
{
    char *s, *last;
    int i, off;
    uint argc, sz, sp, ustack[3+MAXARG+1];
    struct elfhdr elf;
    struct inode *ip;
    struct proghdr ph;
    pde_t *pgdir, *oldpgdir;

    begin_op();
    if((ip = namei(path)) == 0){
        end_op();
        return -1;
    }
    ilock(ip);
    pgdir = 0;

    // Check ELF header
    if(readi(ip, (char*)&elf, 0, sizeof(elf)) < sizeof(elf))
        goto bad;
    if(elf.magic != ELF_MAGIC)
        goto bad;

    if((pgdir = setupkvm()) == 0)
        goto bad;
    *
    *
    *

```

Virtual Memory Map

The diagram shows a vertical rectangle representing the virtual memory map. It is divided into two main sections. The top section is a pink rectangle labeled 'stack'. Below it is a white rectangle labeled 'code'. A dashed line separates the two sections. An arrow points from the code section to the text 'Get pointer to the inode for the executable'.

ref : exec.c

71

The first thing that is done is that we get a pointer to the inode of the executable. 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 slash bin slash l s, so this particular function name I would return the inode for the l s executable.

Next, what we are doing is we are using this constant called read i, 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. So, ELF header or just mention as ELF over here is defined over here as ELF header. 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 Slide Time: 26:23)

exec

```
int
exec(char *path, char **argv)
{
    char *s, *last;
    int i, off;
    uint argc, sz, sp, ustack[3+MAXARG+1];
    struct elfhdr elf;
    struct inode *ip;
    struct proghdr ph;
    pde_t *pgdir, *oldpgdir;

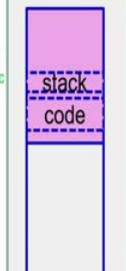
    begin_op();
    if(!ip = namei(path)) == 0){
        end_op();
        return -1;
    }
    ilock(ip);
    pgdir = 0;

    // Check ELF header
    if(readi(ip, (char*)&elf, 0, sizeof(elf)) < sizeof(elf))
        goto bad;
    if(elf.magic != ELF_MAGIC)
        goto bad;

    if((pgdir = setupkvm()) == 0)
        goto bad;
    .
    .
    .
}

ref : exec.c
```

Virtual Memory Map



Executable files begin with a signature.
Sanity check for magic number. All executables begin with a ELF Magic number string : "\x7fELF"

72

So, this essentially is a sanity check and this magic number for ELF should have this particular value that is it should have 7 f ELF. So, this is this is the 7 f is the hexadecimal value, while ELF is the alphabets.

(Refer Slide Time: 26:46)

exec

```
int
exec(char *path, char **argv)
{
    char *s, *last;
    int i, off;
    uint argc, sz, sp, ustack[3+MAXARG+1];
    struct elfhdr elf;
    struct inode *ip;
    struct proghdr ph;
    pde_t *pgdir, *oldpgdir;

    begin_op();
    if((ip = namei(path)) == 0){
        end_op();
        return -1;
    }
    ilock(ip);
    pgdir = 0;

    // Check ELF header
    if(readi(ip, (char*)&elf, 0, sizeof(elf)) < sizeof(elf))
        goto bad;
    if(elf.magic != ELF_MAGIC)
        goto bad;

    if((pgdir = setupkvm()) == 0)
        goto bad;
    .
    .
    .
}
ref: exec.c
```

Virtual Memory Map

73

The next step is a call to the `setupkvm`, 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)

exec contd. (load segments into memory)

```
    .
    .
    .
// Load program into memory.
sz = 0;
for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
    if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
        goto bad;
    if(ph.type != ELF_PROG_LOAD)
        continue;
    if(ph.memsz < ph.filesz)
        goto bad;
    if((sz = allocuvn(pgdir, sz, ph.vaddr + ph.memsz)) == 0)
        goto bad;
    if(loaduvn(pgdir, (char*)ph.vaddr, ip, ph.off, ph.filesz) < 0)
        goto bad;
}
iunlockput(ip);
end_op();
ip = 0;
    .
    .
    .
```

Virtual Memory Map

74

This particular slide shows a continuation of the `exec` implementation. 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.

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 code, so therefore, you get this mapping present over here.

(Refer Slide Time: 28:12)

**exec contd.
(user stacks)**

```
// Allocate two pages at the next page boundary.  
// Make the first inaccessible. Use the second as the user stack.  
sz = PGROUNDUP(sz);  
if((sz = allocuvm(pgdir, sz, sz + 2*PGSIZE)) == 0)  
    goto bad;  
clearpteu(pgdir, (char*)(sz - 2*PGSIZE));  
sp = sz;
```

The first acts as a guard page protecting stack overflows

Virtual Memory Map

The diagram shows a vertical stack of memory regions. From top to bottom: a large purple box labeled 'stack' with a dashed line indicating its boundary; a smaller purple box labeled 'stack guard'; a light blue box labeled 'data'; and a light blue box labeled 'code' at the bottom.

75

The next step in the exec implementation is to create the stack for the user process. So, this stack 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 two contiguous pages So, it one is used for the stack while the other one is used as a guard page 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)

exec contd. (fill user stack)

```

// Push argument strings, prepare rest of stack in ustack.
for(argc = 0; argv[argc]; argc++) {
    if(argc >= MAXARG)
        goto bad;
    sp = (sp - (strlen(argv[argc]) + 1)) & -3;
    if(copyout(pgdir, sp, argv[argc], strlen(argv[argc]) + 1) < 0)
        goto bad;
    ustack[3+argc] = sp;
}
ustack[3+argc] = 0;

ustack[0] = 0xffffffff; // fake return PC
ustack[1] = argc;
ustack[2] = sp - (argc+1)*4; // argv pointer

sp -= (3+argc+1) * 4;
if(copyout(pgdir, sp, ustack, (3+argc+1)*4) < 0)
    goto bad;

```

76

The next step in the exec implementation is to fill the user stack. So, essentially we have created the stack over here. And now we are actually filling the stack. So, we fill the stack with command line arguments. So, we know that every 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 pointed to argument 0 pointed to argument 1 and so on, so these pointers 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 this 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 right creating the argc and argv. The only thing next to do is to actually start executing the process.

(Refer Slide Time: 30:42)

exec contd. (proc, trapframe, etc.)

```

...
// Save program name for debugging.
for(last=path; *s; s++)
    if(*s == '/')
        last = s+1;
safestrncpy(proc->name, last, sizeof(proc->name));

// Commit to the user image.
oldpgdir = proc->pgdir;
proc->pgdir = pgdir;
proc->sz = sz;
proc->tf->eip = elf.entry; // main
proc->tf->esp = sp;
switchvm(proc);
freevm(oldpgdir);
return 0;

```

Set the executable file name in proc

these specify where execution should start for the new executable. Also specifies the stack pointer

Alter TSS segment's sp and esp. Switch cr3 to the new page tables.

BR

77

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 dot 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`. 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.

With this, we come to the end of this video lecture. We had seen a 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.