190031187 OSD Assignment - CO-2
Radhakrishna

1A) The boot program is responsible for loading the unix kernel into Memory & passing control of system to it. Some systems have two or more levels of intermediate boot programs between the firware instructions & independently - executing - executing unix kernel. other systems use different boot programs depending on type of boot. Normal unix boot process has these main phases:

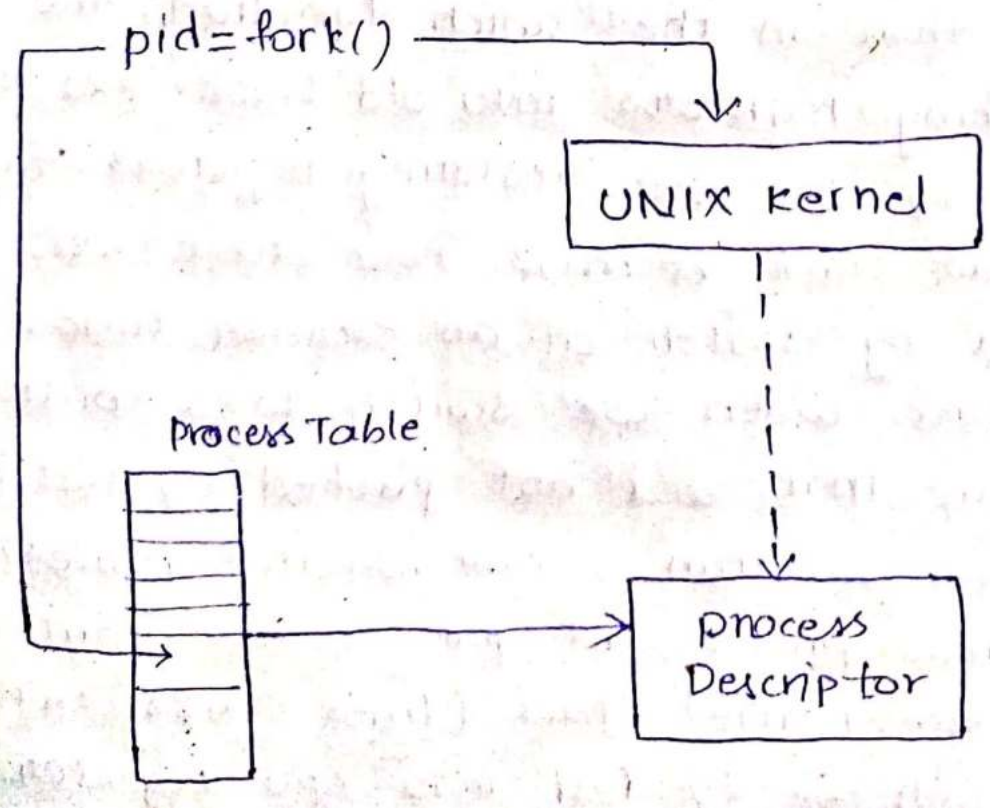
→ Basic hardware detection (Memory, disk, mouse) &
→ Executing the firmware system initializing program (happens) automatically.
→ Locating and running the initial boot program (by firmware boot program)
→ Locating & starting unix kernel,
→ Kernel starts the init process, which in turn starts system process & initializes all active subsystems.

process-creation: At system boot time, one user level process is created. In unix, this process is called init. In windows, it is the system Idle process. This process is parent grand parent of all other process. New

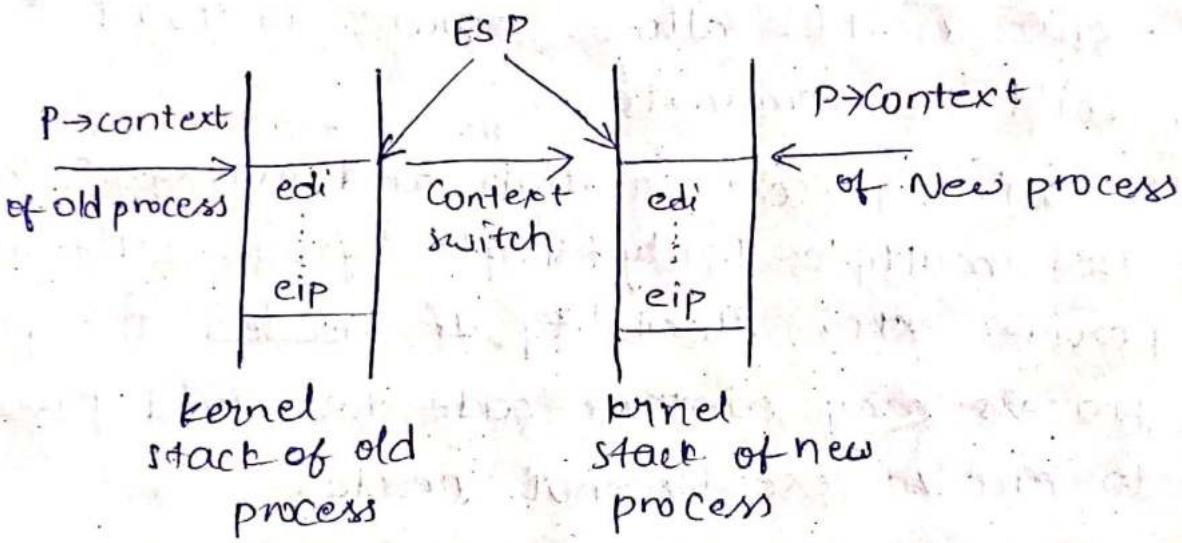
processes are created by another process (parent process)

- unix fork() creates process
  creates a New address spaces
  copy text, data & stack into New address space provides child with access to open files.
- unix exec() allows child to run a new process
- unix wait() allows parent to wait for child to terminate

In unix, process creation and managment uses multiple fairly simple system calls. This provides extra flexibility. If needed the parent process may contain code for child process to run so exec() not needed



2A) The swtch function (lines 2950) does the job of switching blw two contexts & old one & a new one. The step of pushing registers into old stack is exactly similar to step of restoring registers from new stack, because the new stack was also created by swtch at an earlier time. The process of swtch is illustrated below:



→ Note that in the swtch function, the step of pushing registers into old stack is exactly similar to the step restoring registers from the new stack because new stack was also created by swtch at an earlier time, This only time when we switch to a context structure that was not pushed by swtch is when we run a new created process. For a new process, allocproc writes context onto the kernel stack (lines 2488-2491) which will be loaded into cpu registers

by swtch when the process executes for first time. swtch ~~swtch th~~ does not explicitly store the eip to point of the address of swtch statement.

In XV6, scheduler runs a seperate thread with its own stack, Therefore, context switches happens from the kernel mode of a running process to scheduler thread, & from the scheduler thread to the new process it has identified.

As a result, swtch called at two places: by the scheduler (line 2728) to switch from scheduler thread to a new process, or by a running process that wishes to give up the cpu in function shed (line 2766) & switch to scheduler thread. The only exception is a process running for the first time, that never would have called swtch at line 2766. & hence never resumes from these.

3A) Using wait() system call : when parent process calls wait(), after creation of a child, it indicates that, it will wait for child to comple-te at it will reap the exit status of child The parent process is suspended (waits in a waiting queue) untill the child is termin-ated. zombie using fork() will, the address

space of parent process is replicated. If parent calls wait(), the execution of parent is suspended until child is terminated.

algorithm wait

input : address of variable to store the status of existing process.

output : child ID, child exit code.

```
{
    if ( waiting process has no child processes)
        return (error);
    for ( ; ; )
    {
        if ( waiting Process has zombie child )
        {
            pick arbitrary zombie child;
            add child cpu usage to parent;
            free child process table entry;
            return (childID, child exit code);
        }
        if ( process has no child )
            return error;
        sleep at interruptible priority ( event
                child processes exits )
    }
}
```

4A)
```
#include <stdio.h>
int main()
{
  for (int i=0; i<2; i++)
  {
    if (fork() == 0)
    {
      printf("[son] pid %d from [parent]
              pid %d \n", getpid(), getppid());
      exit(0);
    }
  }
  for (int i=0; i<2; i++)
    wait(NULL);
}
```

5A) A system call is programmic way in which a computer program requests a service from the kernel of OS it is executed on. It is a way of programs to interact with OS. Computer programs makes a systemcall when it makes a request to OS kernel. system call provides services of OS to user programs via application program interface (API). It provides an interface blw a process & OS to allow user-level processes to request services of OS. system calls are the only entry points into the kernel system. All programs needing resources must use system calls.

Services provided by system calls: process creation & Management, main memory management, file access, directory, data handling (I/O), protection, Networking. etc....

Types of system calls: There are 5 types

(1) process control (end, abort, create, allocate)

(2) File Management (create, open, close, delete)

(3) device Management

(4) Information maintainance

(5) communication.

FILE :- O_RDONLY, O_WRONLY, O_RDWR, O_APPEND, O_CREAT, O_EXCL, O_NONBLOCK, O_TRUNC, O_SYNC

The rest of descriptors are used by processors when opening an ordinary pipe or special file or directories.

There are 5 system calls that generate file descriptors create, open, fcntl, dup & pipe.

A computer program makes system call when it makes request to OS kernel.

system calls used for hardware services

to create or execute process and for communication with kernel.

Nameing functions :- namei [4354] doesn't make to inode, skepelem [4314] passes path func, dirlookup [4212] does name-to-inode in single dir, dirlink [4252] adds name to a directory.

6A) process termination is a technique in which a process is terminated and realse cpu after completing execution. Most of os use exit() to terminate process.

The process completes all tasks and release cpu.

→ If p1 terminated, now its time when p1 will move to zombie. p1 remains in zombie state untill parent invokes a system call to wait() when this happened process id of P1 as well as p1 entry in process table will released

algorithm zombie :

Input : process to read child exit status

output :- child finishes executing exit() while parent sleeps for 50 sec

```
{
    //fork returns processid in parent process
    if (child processId greater than 0)
        sleep for 50 sec (during process termin
                                -ation);
    //child process
    else
        exit (0);
    return 0;
}
```

A zombie process is a process that has completed its execution but still, its entry remains in process table. zombie Man undead person.

algorithm orphan:

input:- process to finish parent process without waiting for child

output:- parentID, parent exit code.

```
{   //create child process
    int pid = fork ()
    if (pid >0)
        printf ("In parent proces")
    else if (pid ==0)
    {   sleep(30 seconds during pid);
```

```
    printf ("In child process");
  }
  return 0;
}
```

An orphan is a process whose parent process has terminated or finished, but it remains running itself.

7A) Each process has kernel stack (or more generally, each thread has its own stack) just like there has to be a seperate place for each process to hold its set of saved registers, each process also needs its own kernel stack to work as its execution stack when it is executing kernel.

The xv6 scheduler implements a simple scheduling policy, which runs each process in turn. The policy is called round robin... priority inversion can happen when a low priority & high priority process shares a lock, when acquired by low-priority process can cause high-priority process to not run.

Entries in PCB

- pid - process Identifier
  Number incremented sequentially

8A) Round Robin (RR) scheduling Algorithm is particularly designed for time sharing systems: The process are put into ready queue which is a circular queue in this case. In this case a small unit is known as quantum is defined. The algori-thms selects the first process from queue & execute it for the time defined by quantum time. If a process has burst time less than time quantum then the CPU executes next process but if it has burst time higher than the quantum then the process is interrupted or next process is executed for same time quantum.
If a process is interupted then a context switch happens & process is put back at tail of the queue. It is premptive in nature. This algorithm mainly depends on time quantum. very large time quantum makes RR same as FCFS while a very small time quantum will lead to overhead as context switch will happen again & again after very small intervals.

priority scheduling:-

It executes process depending upon their priority each process is allocated a priority or process with highest priority is executed

first. priorities can be defined internally as well as externally. Internal depends on no. of resources, time required. whereas ~~being~~ external depends on time in which work is needed or amount being paid for work done on imp process

It can be preemptive or non-preemptive

Note:- → If two process have the same priority then tie is broken using FCFS.

→ The waiting time for highest priority process is always zero in preemptive mode while it may not be zero in case of non-preemptive.

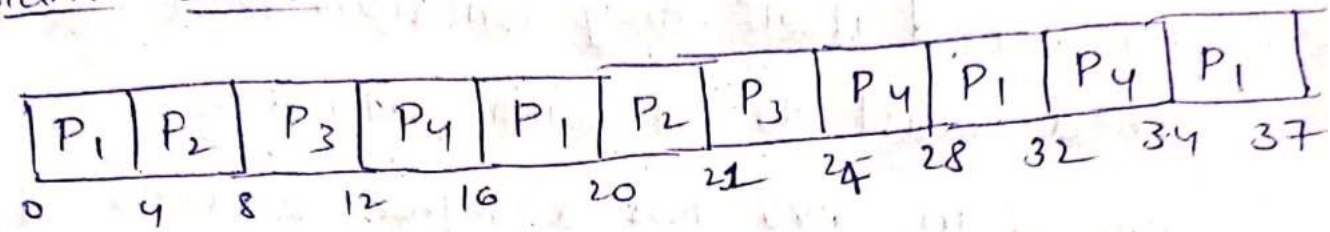
→ cpu scheduling deals with problem of deciding which of the process in ready queue is to be allocated in cpu.

→ In FCFS process requests cpu first is allocated cpu first.

→ on negative side, the avg wait time under FCFS policy is often quite long

P_1	15 4 3	
✓ P_2	5 X	Time Quantum = 4
✓ P_3	7 3	
✓ P_4	10 6 X	

Gantt chart



process Id	Burst Time	Completion time	TurnAround Time	waiting Time
P_1	15	37	37	22
P_2	5	21	21	16
P_3	7	24	24	17
P_4	10	34	34	24

Avg Turn around time = $\frac{37+21+24+34}{4}$

$= \frac{116}{4} = 29$

Avg. waiting time = $\frac{22+16+17+24}{4}$

$= \frac{79}{4} = 19.75$

9A) The priorities assigned to processes are 80, 69, • and 65 respectively. The scheduler lowers the relative priority of cpu. bound processes.

10A) A process that wishes to relinquish the cpu calls the function, sched. This function triggers a context switch, & when the process is switched back in at a later time resumes execution again in sched itself. Any function that calls Sched must do so with pTable.

process in xv6 has a seperate area of memory called kernel stack that is allocated to it. To be used instead of user space stack when it running in kernel model.

The process table is protected by lock. any function that accesses or modifies this process table must be hold this lock while doing do. computers keep of time they same way you do. The real time clock runs even when CPU is powered off its completely seperate from clock cycles of CPU. Time is maintained by chip called RTC (Real Time clock). At time of booting the current time is read from the chip.