Doxygen link : https://foss.coep.org.in/xv6-public/doxygen/html/index.html

Question:

True : Multiprogramming System may not be multitasking

Multitasking system must be multiprogramming

OS code only executes in instances like Hardware Interrupts, Software Interrupts, Exceptions. It doesn't run normally. Hence it is event driven.

OS is not "always" "running" "in the background"!

1. Always - CPU is not 'always' executing OS code.

3. in the background - It is a concept in the application world. In the hardware there is no notion of background

Software Interrupt instructions - Changes the mode bit from user mode to kernel mode.

scanf() runs INT instruction. This INT instruction is a system call.

scanf("%d", &i) -> The OS will first run some part of scanf() in user mode. WHen INT instruction comes, it goes to kernel mode

The OS will copy data from keyboard to the memory location of i.

CPU constantly checks the mode bit while exeecuting an instruction. It doesn't execute a privileged instruction if mode bit = user.

CPU won't allow OUT,IN instructions to be executed in user mode. instead some INT instruction needs to be executed.

```
main() {
        scanf()
}
// C library
int scanf(..) {
        some code
        int 80;
        some code;
}
// OS
control goes to OS.
```

ISR code:

Check value of some register

Do the task as per the code

IRET.

Scheduler is also an ISR

OS runs only when an event occurs.

INT instruction changes the mode bit and then executes the ISR which contains privileged instructions

2 different modes - user and kernel

But the contexts can be process(software-interrupt) and system(hardware-interrupt)

So user mode and process context - appln programs

user mode and system context - not allowed. Bcoz in user mode, the system context interrupts cannot be processed.

kernel mode and process context - means that an interrupt occured due to a process, which caused cpu to change to kernel mode.

kernel mode and system context - means that hardware interrupt changed cpu mode to kernel mode.

Command line and mounting

1. WHy GNU/Linux

a. Programmer's Paradise - most versatile, vast, all pervasive programming environment

b. Free Software - freedom, freely use, copy, modify

c. Highly Productive - do more in less time

d. Better quality, more secure, very few crashes

2. WHy command line?

a. Not for everyone! Absolutely!

b. Those who do it are way more productive than others.

c. Makes you think in programmatic fashion

d. Portable. Learn one, use everywhere on all Unixes, Linuxes.

e. ALl servers come without GUI

3. Few Key Concepts:

a. Files don't open themselves. ALways some application/program open() s a file.

b. Files don't display themselves. A file is displayed by the program which opens it.

c. Programs also don't run themselves. WHen we click on a program, we request OS to run it. OS runs your program.

d. Users(humans) request OS to run programs using Graphical/Command Interface.

4. Pathnames

a. Tree like directory structure

b. Root directory called /

c. A complete path name for a file /home/student/a.c

d. Relative path names: every running program has a current working directory

    eg getcwd() is a system call in C to print current working dir.

    . - current directoru

    .. - parent directory

5. Command

a. Name of an exeecutable file

b. ls is actually /bin/ls

c. Commands take arguments, options. Options and arguments are basically argv[] of the main() of that program

pwd gives the current working directory of the program called as 'shell'.

Every user has a home directory.

6. Shell;

Shell = Cover

CPU -> Kernel -> Shell(user interface)

Shell - interface betn kernel and user to execute programs. Shell requests OS to run programs.

GUI is also a shell!

7. File permissions on Linux:

a. ls -l filename

2 types of users: root(all-powerful) and non-root. Users can be grouped into 'groups'

Every file has 3 sets of 3 permission:

- Octal notation
- Read = 4, Write = 2, Execute = 1
- 644 means Read-Write for owner, Read for group, REad for others.

r on a file: can read the file

- open(..., O_RDONLY) works.

User can read file means, the program started by that user can read the file!

w on a file: can modify

x on a file - can ask OS to run file as executable.

r on a directory: can do ls

w on a directory: can add/remove files from that directory(even without 'r'!)

x on a directory: can cd to that directory

8. Man Pages:

They have sections

man 2 mkdir - read section 2 of mkdir

man -k mkdir - shows all sections

section1- commands

section2 - system calls

section3- lib functions

section4- device drivers

section5 - file formats

GNU/Linux filesystem structure:

/ - Root dire

/bin - contains all essential commands for system to run

/boot - Contains kernel images(actual machine code of the kernel), initrd and configuration files

/dev - files representing hardware devices. These are different types of files. They are device files. Unix and Linux - everything is accessible as a file. Keyboard, mouse are accessible by files. Hence we can play with hardware

sda - Hard disk

nvme - ssd

// C program to access hard drive:

fd = open("/dev/sda", O_RDONLY);

read(fd, arr, 128) // this reads first 128 bytes. But to be run as a sudo!

here is specific information about ongoing processes. /proc is not present on HDD. This is a kernel data structure available as /proc folder. So currently running programs are also available as files on linuxes and unixes.

/sys - system and device control

/tmp - for temporary files. Normally whatever we write in /tmp is removed by ubuntu when we reboot.

/usr - regualar user tools(not essential to the system), /usr/bin, /usr/lib

/usr/local - preferred to /opt/

/var - data used by the system or system servers

/var/log - system log, server log

Files: cat, cp, mv, rm

cp a.c /tmp/ -> means that copy a.c into the folder /tmp

cp -r -> copies folders

Useful Commands:

1. echo
2. sort
3. firefox
4. libreoffice
5. vlc
6. grep
7. less
8. alias
9. tar
10. gzip
11. touch
12. strings
13. adduser
14. su (switch user)
15. df
16. du
17. time
18. date
19. wc

Network related commands

Unix Job Control:

The shell prompt only gets back after a program returns

google-chrome & -> means process runs in the background

If we start process as: google-chrome

And then want to run it in backgrond: then press ctrl+Z(suspend/pause the process, not stop!)

And then type bg. Then it will run in background

$jobs # gives all the jobs running

$fg # brings the background running jobs in a stack wise fashion to the foreground, we can stop them once they come here

$bg # put that job in the background

Ctrl + C - terminate

kill %1 - first job

Bring into foreground fg %1

Mounting:

Windows namespace is a forest. i.e. C:, D:, F: are different trees. There is one tree per partition in windows.

Linux Namespace - single tree namespace (at the top we have a /)

Root is also / and Separator is also /

Combining of multiple Partitions(trees) into 1 is called mounting.

In Linux, after mounting, a partition comes as a single folder on the file system!. Mounting means - accessing a partition as if it was a folder in the current file system.

Use mount and umount on Ubuntu:

```
sudo mount /dev/sda1 /extra/
```

If /extra already contains files, then those are hidden, and /extra now points to partition. Once unmounted. The older data from /extra is visible. It's not overwritten. As Partition is physically different present on HDD.

Partitions have a file system of a particular type. So to mount a partition of type zfs, our ubuntu must have support of type zfs. So we need to install zfs support

```
mount -t nfs 172.16.1.200/folder /tmp/a
```

This mounts folder on another computer to our computer! This is what is called as shared folder. This is nfs sharing.

Mounting and Windows: (IMP!)

https://www.top-password.com/blog/fully-shutdown-windows-10-instead-of-hibernating-it/

When we click on Shutdown on windows, it actually hibernates(incomplete shutdown). Windows 10,8 do this so as to start faster the next time user switches on the computer. But, in this hibernation mode, windows doesn't release complete control on all hardware devices. That's why when we try to mount a HDD in ubuntu with read/write mode, we get following error:

```
$sudo mount -t ntfs /dev/sda2 /home/oem/mount -o rw -o noload


The disk contains an unclean file system (0, 0).

Metadata kept in Windows cache, refused to mount.

Failed to mount '/dev/sda2': Operation not permitted

The NTFS partition is in an unsafe state. Please resume and shutdown

Windows fully (no hibernation or fast restarting), or mount the volume

read-only with the 'ro' mount option
```

So, some HDD state was present in the windows cache. Hence HDD is in an inconsistent state. So shutting down windows COMPLETELY is essential.

Q: How to shut down windows completely?

```
> shutdown /s
```

Type above on COmmand Prompt. See various options for shutting down by typing 'shutdown /?'

passwd command modifies files like /etc/passwd and /etc/shadow

See 's' bit in permission. suid. Process executing a user id.

SYScall

System call - INT80. The value is present in EAX register. Depending on value, a specific function is invoked.

OS kernel can read CPU registers, as assembly code can be written in C lang. So we can read registers.

./a.out - Program becomes a process(code, data now exists in the ram)

fork() - duplicates a process. Creates exact duplica of the current process. These processes run independently of each other

Caller process is parent, Created process is called Child.

fork() returns 0 to the child process, and returns pid of child process to parent process. If child process isn't created then fork() returns -1 to parent and errno is set to indicate the error.

No order of execution guaranteed betn parent and child. The order of execution can get intermixed.

$/usr/bin/ls - also executes ls

execl() - takes pathname of the executable file.

after execl(path) - OS overwrites in memory image of the currently executing process by the code of process pointed to by path.

currently executing process is vanished from memory. exec loads the new process.

Therefore exec doesn't return. Becoz the program in which it should return has vanished!

exec returns if there is some error in loading the new process/ if there is invalid argument.

After execl there is only 1 process running

```
printf("hi");

execl("/usr/bin/ls", "/usr/bin/ls", NULL);

printf("bye"); // doesn't execute. This process is removed from ram.
```

In above program, pid of ls remains same as pid of caller of execl

thread vs fork() - fork() creates duplicate process., thread is a part of a process.

If a process contains 2 threads, and then how does fork() create a duplicate process? - Ans is pending!

wait(0) is a system call. waits until child terminates.

In the shell program, that sir has written, write commands like /usr/bin/ls, /bin/mkdir as the directory is not handled properly while giving to execl.

Try:

$ps

$ps -eaf

ANY PROCESS CAN BE CREATED ON UNIX/LINUX only using fork() + exec() except the init process. There is no other way of creating a process.

pid - process id

ppid - parent process id(t)

Try:

In a C program: After fork() write: sleep(20) (maybe write 2 forks, write sleep after each)

And goto another terminal and type: ps -eaf | grep fork

Where grep fork is written, as the program was started as ./fork

In the ps -eaf output, see the pid, ppid of the number of processes created. you can see who's the parent of whom.

Also try this with execl("./myprogram"). Inside myprogram write sleep, so that we can see what's the pid of myprogram and its parent program. See who's its parent program!. Try this with fork() + exec(). And see whos parent of whom

Also check that, the first ./a.out process is started by bash. Check the ppid of ./a.out. That is same as pid of bash!.

Compilation Linking loading

On ubuntu: C library present at: /usr/lib/x86_64-linux-gnu/libc-2.31.so

libc - library c

file.o contains machine code/object code/binary code of ONLY file.c Afterwards file.o is linked with C lib functions machine code.

Linking - connecting code of a function to call of a function undefined reference to function f - means that linker is not able to find the machine code of f

linker by default has the input of scanf, printf, etc functions. i.e. standard library functions. linker by default doesn't have input of math library functions, etc.

Header files only contain prototypes. Code doesn't exist in header file

What is the use of header files then?

- In the compilation stage(before linking stage), compiler needs to know the function signature, so as to write machine code for CALLING the specific function. Remember that compiler only needs to write machine code for calling a function. Hence compiler needn't know the machine code for that function. So compiler only wants the function signature which is obtained from header file. It's the linker who needs the machine code of the functions so as to combine different machine codes!.

- implicit declaration - compiler assumes some default function declaration - eg return type is int

gcc -c : only till compilation

gcc -E : preprocessor stage only

   -S : assembly code

try:

ps -eaf | less

Try the math library function as sqrt()

1. Do -c option to compile - no problems

2. Try to link. gcc file.o -o file. What we get is an error: undefined reference to sqrt! WHy? - Bcoz linker doesn't by default read the math library object code!

So do: gcc file.o -o file -lm   # l is library. m is math.

Linker: /usr/bin/ld

gcc doesn't assemble or link! It internally calls 'as' and 'ld'

Implicit declaration of f - prototype not available to compiler

Undefined reference to f - the machine code of f is not available to linker

.o - output of a single C code file

.so - (shared object) multiple C machine code files. Also contains extra information of a kernel. It is dynamically linked.

.a - statically linked library file

The gcc works for a particular processor. For eg gcc on my computer converts into machine code understandable by intel i5 processor.

Linux kernel takes around 5 hrs to compile

So if we write code in different files, compilation time is reduced, as we compile only the modified file.

#include"f.c" is allowed. But it doesn't serve purpose. It means that the file to be compiled will become much bigger! As we are copying 1 .c file into another! So more time to compile

Environment Created by OS - means fork() and exec()

Normally compilation is automated by makefile

EXE is not a format. Its a name of file. The format is actually PE.

exec() on windows checks contents of executable file and also whether filename has .exe extension also

exec() on linux doesn't check the extension. It just checks whether contents are in ELF format.

a.out is JUST A FILENAME. It's no more a formaat. It's of ELF format

```
objdump -D -x /bin/ls | less
```

-D: means disassemble

You can read the machine code of printf, scanf, getchar also using:

```
objdump -D -x /usr/lib/x86_64-linux-gnu/libc-2.31.so | less
```

We see the machine code of getchar(), etc!!!

The loader is actually a part of exec()! Bcoz exec() has to read the file on hard disk and check if the file is of the ELF type as processor has to execute it. Hence each OS has its own executable file format. As kernel has to read and understand the executable file and give instructions to the processor to execute. So format is necessary

Memory management features of processor - for eg stack pointer, IVT

memory management architecture of kernel - OS decides to occupy memory based on memory occupied by processor.

Cross-Compilation:

xv-6 doesn;t have a compiler. So we compile on Ubuntu for xv-6

gcc -m : we can compile for different processors

Try this to check format of file:

```
$file code.o
$file /bin/ls
```

```
gcc f.o g.o try.o -o executable -lm
```

After above command, gcc invokes the linker and tells it to link g.o, f.o, try.o, math lib, standard C library to form an executable.

Calling convention

pop %ebp - means store top of the stack in ebp and decrement stack pointer

Q. Who is the calling convention for ? - CPU, OS, Compiler?

Ans – Compiler

rbp,rsp are index registers.

rbp is basically an address

push rbp means copy the address in rbp into top of stack. After this rsp decrements. Is this right?

Q. Does push decrement rsp or rbp? - Ans - push decrements rsp

-8(%rbp) is which form of addressing?

WHat is the difference between movl and mov and movq?

movl - move long

rbp always points to the top of the activation record. rsp moves back and forth in the activation record.

There are functions which call main() also.

Activation record = local vars + parameters + return address.

x86 caller and callee rules - these are the rules for COMPILER.

To see how compilers behave on 32 bit machines, install a 32 bit OS on VirtualBOx.

Calling convention is specified by the processor manufacturer and convention is used by the COMPILER.

In simple3.c code, 6 parameters are saved in registers, rest are saved in stack. This is bcoz, upto 6 parameters are in registers on 64bit machines.

gcc -O : tries to Optimise the code.

Write many C programs. Try to read assembly code. gcc -S

X86

AX-DX - general purpose registers

IP == program counter

Address space - the space in the ram which can be accessed by passing address

multiply to 16 means left shift by 4

pa = va + seg*16 (va means virtual address)

Q. Suppose there is a 8086 processor, and we are running some OS on it. Whose Job is it to set value of code segment, data segment, etc. - Ans: Think. It's ok if you are not able to come up with the answer

80586 is called as Pentium by Intel

After pentium, intel started calling processors as i3, i5, etc

Intel gives backward compatibility.

While booting cpu runs in real mode - means 16 bit mode!

for 32 bit processor, addressable memory = 2^32 = 2^2 GB - 4 GB

In 80386, there are essentially 4 registers: EAX, EBX, ECX, EDX. AX, BX are parts of EAX,EBX

Segment - chunk in the RAM which can be obtained by using the value of segment register

- Segment register contains starting address of that segment

In 80386 how is CS:EIP used? Hold on. CS is not multiplied by 16 in 80386. The stuff is more complex here.

OS sets values of CS, DS, etc. But The calculation of address of CS:EIP happpens in the processor.

GNu assembler(gas) uses AT&T syntax: op src, dst (xv6)

b - 8bit - byte

w - 16bit - word

l - 32 bit - long

movl $0x123, %edx - means store the number 0x123 in edx - immediate addressing - as operand is preesnt directly

movl 0x123, %edx - means goto address 0x123, take 4 bytes(bcoz l is present) from there, store in edx. - direct addressing - as operand's address is directly given

movl (%ebx), %edx - goto address present in ebx, load operand in edx

Fetching instruction from memory - 1 cycle

Fetching operand from memory - 1 cycle

mov %eax, %ebx - assembler interprets this mov as 32 bit bcoz the registers specified are 32 bit.

mov %ax, %bx - assembler interprets this as mov as 16 bit bcoz the registers specified are 16 bit

mov $0x123, 0x123 # ambiguous. Assembler doesn't understand how many bytes should be used to store 123

Protection levels / privilege levels - User mode and kernel mode

Linux uses int 0x80 for system calls

xv-6 uses int64 for system calls

IDTR register gives address of IDT table in memory

Each entry in IDT is called Gate

IDT limit - tells how many entries are there in IDT

OS initializes IDT, IDTR

Memory Layout of a C program- This is a memory layout BY the compiler!. Compiler conceptulaizes this memory.

text - object code of the program. i.e. object code of functions

initialized data/data - global variables

When we say data, we mean global variables

uninitialized data(BSS) -

Stack and Heap are present in same section. Stack grows downward, Heap grows upward.

environmemt vars - vars provided by parent of our program. eg shell is the parent of our program. So shell can provide some vars to the child.

Sections which are precisely known by the compiler - text, data, BSS

text,data,BSS are present in ELF. When execv loads program in memory, these text,data,BSS are copied as it is into memory. THe OS creates stack and heap afterwards

Why is Data,BSS present in ELF? - bcoz they have to be present in memory before the main() starts executing.

Real mode and protected mode is not same as User mode and kernel mode.

8086 didn't have a notion of user mode and kernel mode. 2 modes are necessary to write a multitasking OS. So not possible to do multitasking in 8086.

In real mode, 16 bit mode, the calculations are done like segment*16 + offset. These are not the calculations done in 80386.

Compilers Job is to generate machine code for the program.

XV6 : single user

call graphs - has functions has vertices and function calling another function as edge

Design of OS - 2 approaches - microkernel and monolithic kernel

What is monolithic approach - kernel actually is structured in multiple layers, where each layer does its job. eg File system layer, System call layer. all layers talk with each other. Kernel part is big

microkernel - keep kernel as small as possible. Code provides only basic services. Handling files, hardware devices has to be done on top of kernel. Kernel part is small

fs.img - image of the file system

xv6.img - image of the kernel

cat.c, echo.c,....zombie.c are not a part of xv6 kernel code. They are applications to test xv6.

kernel.ld contains instructions to linker to create executable. This executable is not same as our C executable. It has to be different. As it is an executable which needs to be loaded by boot loader.

mkfs used to create formatted disk

/dev/zero - infinite length files containing all 0s

xv6.img is concatenation of bootblock and kernel. It is created using 3 dd commands

If the count is not specified, then dd copies the entire input file

```
dd if=/dev/zero of=xv6.img count=10000
```

This creates a xv6.img file completely of 0s of 10000*512bytes

```
dd if=bootblock of=xv6.img conv=notrunc
```

This means that don'e truncate the file, overwrite the starting 512 bytes of xv6img by bootblock

seek=1 means don;t overwrite the first 1 block. Copy from the second block of 512 bytes

xv6.img is a concatenation of bootblock, kernel followed by some 0s

fs.img is another disk file. it contains user programs and README

qemu is given both xv6.img and fs.img. qemu runs with 2 hard disks.

Symbol table - listing of addresses and the corresponding variables.

kernel code executes asynchronously.

See sh.c file in xv-6 to understand how the shell is written in xv-6

int $T_SYSCALL is int 64

There is no C library in xv-6. Whatever we need, we have to build ourselves there.

To compile programs for xv6, we have to compile them on Linux for xv-6. So we cannot use the Standard C library on Linux to link with the xv6 code.

We have to use ulib.o, printf.o, umalloc.o to link out code with library code.

But mkfs is not linked with utility files like ulib.c etc, but linked with Linux C library.

Start playing with xv6. Understand the compilation process in make.

Read output of make qemu command and understand each and every line.

Why can't we use the C library on Linux to link programs for xv6? - Bcoz the C library on linux has machine code for Linux OS. We want machine code for xv6!

link is a system call to rename a file.

BASH

1. Shell Wildcards

a. ?(question mark)

Any one character

a?c means a followed by any character followed by c

ls a?c

ls ??c

b. *

0 or more number of characters

echo * - * matches every filename in current directory

c. []

Matches a range

ls a[1-3].c means a1.c, a2.c, a3.c anything will match

d. {}

ls pic[1-3].{txt,jpg}

Matches pic[1-3].txt or pic[1-3].jpg

Bash shell BEFORE executing the command, checks all these wildcards and modifies the argv before doing the exec.

2. Shell variables:

j=5; echo $j

All variables in bash are strings. j=10 doesn;t mean the number 10

No space in j=5

set command shows all variables that are set right now

set | less

a=10; b=20; echo ab (concatenation of a and b is done before passing to echo)

Shell's predefined vars:

USER

- Name of current user

HOME

- Home directory of current USER

PS1

LINES

HOSTNAME

OLDPWD

PATH

$?

Redirection: Inter-process communication

1. Output redirection

   cmd > filename

   Redirects output to a filename

   eg ls > file

   The output refers to stdout. means functions like printf(), ...

2. Input redirection

   cmd < filename

   The command reads its input from file

   The input here refers to stdin.

   stdin means input from scanf(), getchar()

Reads input from file instead of keyboard.

Pipes:

last | ls

grep bash /etc/passwd | head -1

Connect output of LHS command as input of RHS Command

Output refers to stdout and Input refers to stdin

cmd1 | cmd2 | cmd3

Concept of Filters: Filters are programs which read input ONLY from stdin (keyboard, eg scanf(), getchar) and write ONLY on output,stdout (eg. printf, putchar)

Programs can be connected using pipes if they are filters!

Most unix/linux commands are filters

The test command:

test 10 -eq 10

test "10" == "10"

test 10 -eq 9

test -f "filename"

After running test, it doesn;t print anything. Check the $? value. 0 means true. 1 means false.

Shortcut notation for calling test:

[]

[10 -eq 10] - the spaces are compulsory

Write $[10 -eq 10]

Check echo $?

The expr command and backticks

expr 1 * 2   - we have to use \ with a * bcoz, * means wildcard by default.

Backticks `` capture output(stdout) of a command

v=`ls`

Bash is also capable of using if then else

We can use ; to separate shell statements.

echo $a; echo $b   - combine 2 different statements on a single line

In shell

0 - TRUE

Non zero - FALSE

Exactly opp to C!

if ls: then echo hi; else echo bye; fi

So if ls has return valiue of TRUE(0) then echo hi!

while do done

In bash: we can either separate statements based on newline or ';'

for x in ... do done (this ... is a list)

for i in {1..10} do done

for i in * do done

case ... esac

grep .... | head -3 | tail -1

Memory Management Basics:

When we say CPU, we mean Execution unit of the CPU. (fetch, decode, execute)

Memory layout - It is a view taken by the compiler while compiling a C program

$size /bin/ls - run size on elf files

there is no size calculation for stack and heap bcoz they are allocated on runtime.

One process should not be able to access data from other process. That's why we get segmentation fault when our pointer points to some other memory location.

Loader - read elf file and move it into RAM

Address binding - determining address of different components of a program. i.e. determining address of bss, data, code, , heap, etc.

Limit and relocation are 2 registers.

Relocation register - Base register

Location of Process in RAM = base(relocation) register

Total size of the process in RAM = limit register

MMU - memory management unit of processor

MMU is the part of processor.

Logical address is the address issued by execution unit of Processor.

MMU issues physical address. It actually adds the Base register.

In base + limit scheme, Compiler assumes that process starts from 0

OS remembers the values of Limit and relocation registers in its own data structures.

HW - It should be possible to have relocatable code even with "simplest case" - How?

In multiple base + limit pairs - there are different segments(different chunks of memory).

Indirection - commonly employed techniques in programming. I won't do it, you do it. Use of pointer is also indirection. Bcoz we are accessing variables using someone else that is pointer.

IA-32 : 32 Bit intel Architecture

Selector - index into the table.

When Multiple base + limit pairs are in memory, it is highly inefficient. Bcoz for conversion of an address needs memory address.

The address that we print in a C program using %p is a logical address.

Logical memory - view of compiler

Paging means not segmemtation world! Its a different world

Page table is a simple array.

Paging allowes us to load a process anywhere in the RAM. But a page is kept continuously in memory!

Page number = Byte Number / Page Size

Page table is set up by the OS in mem   ory

PTBR - Register - Page Table Base Register - points to page table in memory

p - page number - index in page table

d - offset

For paging scheme:

Once OS is done setting up PTBR, page table, essential hardware setup, schedule the Process, process runs on its own. The logical->physical address translation happens in Hardware!

Physical address = logical address given by the compiler + base

The OS sets up the hardware, and after that the hardware manages translation on its own

Segmentation and Paging are different concepts

But x86 provides both segmentation and paging.

Explanation: Multiple Base + Limit Pairs, with Furthur Indirection:

Address generated by CPU is logical address. It has an implied selecter. This selector is an index in the table.

The descriptor table is an array of Base and Limit pairs.

Selector gives which Base+Limit is to be used.

The value of that Base in The table is added to the offset.

Problem with this approach is that it is highly inefficient as memory needs to be fetched for decoding the address. Memory access is slow.

Xv6-bootloader

Qemu has its own BIOS. The BIOS of Qemu loads the boot loader present on sector 0 of the disk which we pass to Qemu.

We are going to see the Code which will load OS from HDD.

xv6.img - image file of xv6

The first sector of xv6.img is bootblock (bootloader must be present at sector0)

the next sector is the kernel.

See in makefile, xv6.img, bootblock

bootblock.S and bootasm.c are the 2 files which form the bootloader

Bios loads boot sector into 0x7c00 memory location in x86

This 0x7c00 is documented by x86 and qemu.

Read the Makefile :

xv6.img, bootblock part etc

The start -Ttext 0x7c00 means that load the variable start in memory at 0x7c00

start contains code of bootloader

So it means load the code of bootloader at 0x7c00

When bootasm.S starts, it sets cs to 0. Hence entire memory is available.

.code16- assembler directive : no machine code generated for this

start is a variable which is at 0x7c00

cli instruction - clear the Interrupt Flag. This causes processor to ignore maskable interrupts.

A not so necessary detail: Enable 21 bit address.

In original scheme of segmentation, we had 20 bit address.

In this part of code, we just enable the 21st bit.

inb - IN instruction

The set20.1 and .2 are just loops.

We can skip this part. As we don't need it.

In x86, paging has 2 modes - one-level/2-level paging.

In logical address, selector is implied if it is not explicitly mentioned.

GDTR - Global Descriptor Table Register gives the location of Descriptor Table.

OS stores the Address of GDT in GDTR.

In basic Paging Hardware: Address

Page Table needs to be continuos in memory. If physical memory is of size 4GB and Pages are of size 4KB, then Number of entries in Page table are 4GB/4KB. This is very large and probably that much continuous space isn't available. So to avoid this multilevel/hierarchical paging is done.

The address is divided into those many parts. In 2-level of paging, address is divided as p1p2d.

x86 allows us to choose betn 2 level and 1level paging. for 1level paging, the size of memory page is larger - 4MB. For 2level paging the size of memory page is smaller - 4KB.

For 2level paging, CR3 register gives the address of Page directory(1st page table) and then in the Page Directory we get address to be indexed in Page table.

For 1 level paging, 22 bit offset is used. Hence memory must be 2^22 which is 4MB.

For xv6, we are not going to use 4KB page(2-level paging). We are only going to use 1 level paging.

Page Directory and Page Table are in memory tables.

CR3 gives the base address of 1st page table(i.e. page directory)

Ignore CR4 table for now.

The address bits given to Page Directory are 10

therefore 2^10 entries are there.

Each entry is 4 Bytes

So Page Directory fits in 4KB

Each Entry in Page Directory points to a Page Table. There are multiple Page Tables!

The next 10 bits in linear address are used to index into that particular page table whose address is given by the Page Directory.

IMP: The job of the kernel is to:

Set up GDT, store appropriate value in GDTR

Set up the Page Directory and Page tables. Fit the address value in CR3 register.

After kernel loads all this, all address translations/violations are handled in hardware! Kernel doesn't need to do anything.

xv6 sets Base to 0 and Limit to 4BG in all entries of GDT!

So segmentation is off. It means all segments begin at 0 and occupy 4GB. And in 32 bit, max addressable memory is 4GB.

In GDT table, the base and limits are splitted. Some bits here and some bits there.

In protected mode, the Segment selector is interpreted in different ways. lower 3 bits are used for different things. It is actually only a 13 bits entry.

Now x86 has 2 kinds of tables - GDT and LDT. xv6 only uses GDt.

Now read line numbers 42-52 of bootasm.S gdtdesc:

    .word

    .long

This means that gdtdesc is actually address of an array containing the data of 2 bytes and 4 bytes!

start:

    mov

    add

    mov

This means start is a pointer to the object code of mov,add,mov...

So in assembly we can see that code and data are residing together

lgdt gdtdesc - load the GDT register with gdtdesc

What is gdtdesc? - Go down and search for gdtdesc:

.word - 2 bytes

.long - 4 bytes

gdtdesc is a 6 byte data. - this data is stored in GDT register

gdt: is the gdt table

SEG_ASM(STA_X|STA_R, 0X0,0XFFFFFFF) - This block of data has executable and read permission. from 0 to 4GB. // so we are disabling segmentation

gdt is actually an array of 3 entries

lgdt gdtdesc will load the address of array of 3 entries whch is the gdt into GDTR.

So now GDTR points to the GDT table.

Now with this primitive setup, kernel will start running. Kernel will have access to all the memory here;

movl cr0, %eax means copy cr0 to eax

The last bit of cr0 is effectively made 1 by using the 3 lines

ljmp a, b

does this : CS = a, IP = b

SEG_KCODE = 1

1 is left shifted by 3 in the Code Segment. Why are we doing left shift by 3?

Bcoz there are 3 bits in the segment selector which are not the actual data. So now we stored 1 in CS.

Why did we store 1 in CS?? ANs - Remember that in protected mode, The segment registers are an index into the GDT.

In the GDT, at index 1, the actual Base+Limit for Code Segment are present!!

Hence we are putting 1 in the code segment

In assembly language, code and data can be present together!

$start32 is the address of the code starting from start32.

As after start32, code is present, start32 gives address of the code.

After gdt: we had some data present, so gdt is giving address of data

so in assembly depending on what is stored after the label, it either gives the address of code/data.

SEG_KDATA means value 2. Why are we storing the value 2 in ds?? - Bcoz DS will be an index in the GDT. at index 2 in the GDT we have actual Base+Limit of DS.

We left shift SEG_KDATA by 3 bcoz the structure of segment registers is like that in Protected mode.

movl $start, %esp

means 7c00 is stored in stack pointer.

stack pointer grows downwards

Above 7c00, code of bootloader was present

So below the stack will grow!

Why do we need a stack?. Bcoz we need to call bootmain

What is bootmain? - bootmain.c -> Now we are in the world of C

The call of bootmain and the code of bootmain was already linked in the Makefile.

Now the C functions that will be called in bootmain() will work on the stack that is growing downward from 0x7c00

Now we are reading C code. So now we are in the world of calling convention. So code of bootmain.c is compiled using calling convention

bootmain.c will read the kernel code from the disk.

Different permissions are there in the GDT are there bcoz there are some areas where we don't want overriding. For eg if code is present in some segment, then we don;t want to overwrite. So hardware will now check for read write protection also.

Real mode - processor runs as 16 bit

So the address is to be generated by segment register + offset register

We are going to read the code of bootasm.S

There is another file : bootmain.c

We will learn how bootasm.S loads the kernel

# - comment

.code16 - tells assembler that machine code is to be generated for 16 bit register

.globl start   - jump to start

start must be present at 7c00

By default 21st bit is disabled in all processors

inb - IN instruction

Now protected mode - 32 bit mode starts

x86 uses both segmentation and paging

x86 address translation:

offset is given by the CPU

The selector is implied. So for sp, ss is implied.

selector - ss, cs, ds

The descriptor table is in memory. It contains Base and Limit pair

In x86 we have 1 and 2 levels of paging

CR3 register gives location of page directory

PDE - Page Directory Entry

xv6 turns every Base and Limit to 0 and 4GB respectively to turn off segmentation and effectively only using Paging

GDT - Global Descriptor table

lgdt gdtdesc

gdtdesc is a label in the same file

SEG_NULLASM is present in asm.h

These macros(SEG_ASM) are generating a GDT Entry having a format specified in our ppt

STA_X - memory segment is executable

STA_R - memory segment is readable

0xffffffff - is 4GB

lgdt gdtdesc - load into gdt register, the address given in gdt descriptor

For FS and GS - limit is 0

movl $start, %esp means move 7c00 into esp

so from 7c00 to 0 memory will be available for function calls

We are in the world of C, means we are in the world of calling convention

The only effective code in bootasm.S is from lines 42-51. Here the gdt table is set properly.

ljmp means long jump

Shifting (<<3) is done bcoz the format of the register is like that.

We can access any address in bootmain.c bcoz the gdt table has been set like base = 0 and limit = 4GB. So entire memory belongs to us

insl - copies multiple bytes one after another

What does the bootmain() c code do?

It has struct elfhdr -> elf file has a format. So we read that using elfhdr

This code does -> calls function readseg

readseg - reads from disk into RAM.

In the loop, information from kernel ELF file is read and stored at appropriate positions in the RAM

For that we need to understand the structure of ELF file. Different features of ELF file are used at different times.

entry at the end of the bootmain() is actually there in a assemblu file entry.S.

Uptil now we have not enabled paging yet:

goto bootmain() function in bootmain.c

This function loads the kernel into memory

xv6.img - concatenation of bootblock and kernel

First there is a bootblock

After a seek 1 is a kernel

While running qemu, we passed xv6.img to it. That's why the code of bootmain is also passed to qemu.

So the kernel code is only few sectors ahead of bootmain (!)

ELF can be an executable file, kernel file, etc,

Program header table is an array. It has offsets to different sections in the ELF file.

uint magic: It identifies the file as an ELF file.

uint entry - first instruction in the ELF file that should be executed

uint phoff - program header offset - tells where the program header is present in the ELF file

ushort phnum - number of program header entries.

For each section - text,data,rodata there will be one program header.

uint filesz - size of the segment in the file image. Size of text/data/rodata

uint memsz - size of the memory in which that segment is loaded. - It can be the case that .text is only of 300MB, but it is loaded into memory region of 500MB.

xv6 decided that kernel will get loaded in virtual address 0x8010000 which corresponds to a physical address of 0x0100

Our kernel code can write to any address in the memory as we have setup the MMU in that way.

readseg(pa, count, offset) - reads the count starting from offset

in the readseg() code, 1 is added to the offset. So readseg(pa,4096,0) actually reads from the starting of the first sector where the kernel elf file is present

512 is default sector size on any harddisk

So count/512 times readsect() is called by readseg. As at any time 512 bytes are read.

Data is available in the hard disk controller.

elf->phoff gives the offset of program header.

elf->phnum - number of program headers

ph->memsz > ph->filesz   - if memory size is greater than file size

entry() -> is the code from Kernel. It is present in entry.S

Now we are out of the bootloader.

Kernel begins at entry - remember that we are still running without paging.

Now we will read entry part of code in entry.S

Figure out how big the kernel code actually is?

Run objdump on the kernel elf file.

In the next few lines of entry() we are going to setup only 1 page of 4MB.

in the page directory, only 2 entries are filled - 0th and 512.

512 * 4 = 2GB

So logical 2GB address space is mapped to the first 1 MB page.

Identity mapping means that Base=0 and Limit=4GB. So entire memory is available.

We are going to be setting only 1 4MB page frame. Both 0th entry and 512th entry will point to this 4MB page.

We have already loaded the kernel in this 4MB page. Not at 0, but at 0x01000 (something like this)

What is there in entrypgdir? - It is actually declared in main.c on line 103. It is actually an array

entrypgdir is an array of 1024 elements. Each element is of 4 bytes. Out of it only 2 entries are being initialized.

First entry is 0.

Second entry is KERNBASE >> PDXSHIFT.

KERNBASE is virtual address at which kernel is loaded.

PDXSHIFT is 22 // 22 is actually the offset in the linear address given to Page Directory.

Why do we need to map [KERNBASE,KERNBASE + 4MB) to [0,4MB], bcoz we have loaded the kernel in that way from bootmain.c

While loading we have mapped the address 0x80100000 to 0x010000

So we need to map the addresses of the C code to first 0-4MB Physical address

As the offset is of 22 bits, if we shift the KERNBASE 22 bits to the right, then we will get page number corresponding to KERNBASE

This KERNBASE>>PDXSHIFT comes out to be 512

Now these 2 entries are set to point to 0th page, along with some flags.

PS=1 means Page Size is 4MB

Page Size Extension -> enabling 4MB pages

Base addr of Page Dir is stored in cr3

What is V2P_WO(entrypgdir) ->

Remember that entrypgdir is a part of the kernel code. So the address corresponding to entrypgdir in kernel ELF file is a virtual address wrt to the KERNBASE.

V2P - Virtual To Physical

V2P just subtracts KERNBASE from the argument. This gives us the actual physcial address

movl $(stack + KSTACKSIZE), %esp

KSTACKSIZE is 4096

stack is present in the same file: entry.S

.comm stack, KSTACKSIZE

This .comm directive allocates a block of size KSTACKSIZE and the name of the block is stack.

stack is actually of 4096 bytes.

$(stack + KSTACKSIZE) means add 4096 to the address where stack is stored. Why to add those many bytes? - bcoz stack starts growing downwards!

The stack pointer is being initialized once again.

When was the last time we initialized the stack pointer?

Why are we re-initializing the stack pointer again?

$main is nothing but the address of main in main.c

Now kernel will enter the world of C code.

What is *%eax?

If the code was jmp $main

main is name of function in C code. Assembler while generating object code, uses the current PC as virtual address. to avoid this we use *%eax so that it jumps to a fixed virtual address.

bootasm.S bootmain.c steps:

3. Reads kernel elf file and loads it into RAM, as per the instructions in the ELF file:

"As per the instructions in the ELF file" is important. Bcoz the instructions were present in the Program Header!

There are 3 program headers

How to interpret Program Headers

```
LOAD off 0x00001000 vaddr 0x801000 paddr 0x00100000
     filesz 0x00007aab memsz 0x00007aab
```

This tells that: load the contents of the ELF file at offset 'off' into the physical memory address 'paddr' and load 'filesz' number of contents. If memsz and filesz has some difference, then it is filled with 0s

In the objdump, what u see at first 5 lines is the elf header

later we have to program header.

Processes

Processes - Third Chapter

Kernel has to maintain data structures for maintaining processes. - Bcoz it's the kernel's job to allocate memory to a process.

Every process has its own list of opened files. That has to be maintained for each process.

"waiting" for different events - waiting for hard disk controller to read/write data or waiting for data to arrive on network.

There is one PCB(Process Control Block) per process. It is identifier of a process.

PID - unique number that identifies a number

Why to store register values in PCB? Bcoz multiple processes are time sharing the CPU. So the PCB must save current processes's Registers.

Memory limits of process - means the memory management information for the process

fd[] : Array of pointers to struct File

The first NULL pointer in the fd[] array is selected to point to struct file

So file descriptor is the index in the open files array of PCB

When we open(), normally the first fd which we get is 3. Bcoz the current process is already initialized with 0,1,2 as stdin,stdout,stderr. File numbers 0,1,2 are already open for our process.

The PCB is the part of the kernel's memory space.

In xv6 the pcb is represented as struct proc

struct file : uint off means offset. at this offset, the next read and write will occur

in xv6 the list of processes is given by:

```
struct {
        struct ..
        struct proc proc[NPROC]; // here the name of array is also proc and structure is also proc
}ptable;
```

ptable is a global variable.

In xv6 the only data strcture for processes is this ptable. That's it.

So in xv6, processes which are on hold, which needs to be scheduled are all present in the ptable struct. So scheduler needs to do more work in xv6

Go to sched.h in Linux Kernel and see the struct task_struct. It's a HUGE structure.

struct List head just has pointers of the same type.

In linux kernel, the processes are linked together through internal pointers!

In linux kernel, they link the internal members of the structure together

Time slice expired means timer interrupt occured

Moving a process to a Wait Queue means moving the PCB of that process into a Queue!!

Child termination wait queue - Means list of processes who have called a wait() system call!. The parent will wait until child terminates.

New State - Fork has happened just now. PCB is being constructed. Afterwards PCB is moved into the Ready Queue.

Ready State means - Scheduler can select the process for executing on CPU

After Scheduler picks up the process, it changes the State to Running.

Any process must exit by calling exit() system call

exit() will clean up the PCB bcoz exit is a system call, and the last line of exit() will call the scheduler to schedule !

To call a exit(), process must be running.

Remember that PCB is present in the kernel data structure. This PCB is not same as the Code,Stack,Heap available to the process!

See in the Conceptual Diagram: Process which is waiting cannot terminate.

What is not shown in this diagram is not possible!

Pressing Ctrl+C generates a signal. Signal is also handled similar to interrupts.

When Ctrl+C is pressed, That ctrl+c goes to kernel, and then kernel will notify the process that the signal has occured. And the process's own code called the signal handler will run. But if not then kernel runs its own default signal handler. Terminate the currently executing process. Note that here the kernel terminated the process, and not exit()!. Individual programs can also write signal handlers.

The state diagram which is shown in the slides, is the state diagram of a process running by its own process code. It doesn't show how process behaves on a signal/segfault

What happens to the return value of the process, when Ctrl+C is pressed?

When a process is terminated, it hasn't returned. Bcoz return value means value which is received after the function returns. So after pressing ctrl+c the return value which we get on the shell (echo $?) is 130.i.e Owner Died.

Always remember that __asm__("int 0x80") means generating an interrupt. After generation of an interrupt, The OS code will run! So when we call read(), the kernel will come to know that the current process is trying to do a read(). Hence kernel comes to know that we should move this process to a wait queue.

disk_read() will keep executing in hardware!! So the reading, writing on files keeps on happening in the hardware parallely. disk_read() is not blocking.

When interrupt occurs, the scheduler will run to schedule next process. In the triple dots shown in Context Switch Diagram, the kernel context is being executed. i.e. the kernel code is using the Registers.

So actually in the Diagram there are 2 context switches. One from the process to kernel. One from kernel to another process.

Calling Convention makes function calls work in the same process.

Code for context switch must be in assembly. This is bcoz if we write it in C, then the code will be in some function according to calling convention. Hence we don't write in C.

Now we start reading xv6 code.

Meaning of Layout of process's VA Space - THe compiler, linker will create code for the process assuming the LAyout which is shown. So compiler assumes that data will be present at address 0, then User data is there and so on.

KERNBASE = maximum possible memory available to a program. = 2GB in xv6 - 0x80000000

It is assumed that Kernel itself is present from KERNBASE onwards.

The mapping that is shown is for kernel part only.

2GB is for Processes, 2GB is for kernel

When a process is forked, then a mapping of User stack, User data, Kernel Data, Kernel text everything must be mapped to physical pages.

0x8000000 - 2GB

In xv6 every process has 2 stacks

Kernel stack - present for every process. Note that kernel stack is present in the PCB. So kernel stack is present in Kernel part of memory.

User stack is present in user process memory space.

Kernel stack is used for running the kernel code on behalf of the kernel.

Kernel and apllications have different stacks so that no clever programmer tries to get pointer to a kernel executing function.

There is a third stack also! But this stack is not a per-process stack. It's just a genetic stack.

Process has to be less than 2GB in xv6

xv6 has 2 level paging. the pde_t* pgdir gives a pointer to the page directory for a process. This pointer later goes into the CR3 register while scheduling.

context - gives address of structure where the context of the process is saved.

The pgdir has mappings from 0-4GB. The page directory and different page tables are set up when fork() is done

Trap means interrupt.

The argv are stored on the stack, when exec() is called.

All the page directory and page tables are present in the Free Memory section of the kernel. See the diagram.

When does a kernel run on behalf of a process? - When kernel is doing system call.

xv6 does 2 level of paging.

To read Kernel code, understand the data structures first.

```
struct .. {
        uint off_15_0: 16; // means that the variable is of 16 bits.
}
```

Interrupt In processes

xv6 uses segmentation + 2 levels of paging.

On a trap, we transit from user mode to kernel mode

The 3 stacks are present in xv6 code.

The global kernel stack is used when the kernel is executing on the kernel's behalf. i.e. the system context. This stack is the global stack

The per-process kernel stack is used when kernel is executing on behalf of user program;. User program means application program.

The current privilege level is stored in 2 bits of the Code Segment Register.

In x86, int 10 means "make 10th hardware interrupt". So we are generating hardware interrupt using software interrupt(int)

xv6 uses number 64 for actual system calls

Linux uses int 0x80

In protected mode, the segment registers are used as index inside the GDT.

System call interrupt means the software interrupt.

IDT is an in-memory table.

Goto xv6 code directory

```
$cscope
```

Search for gatedesc in Global definition.

see struct gatedesc in mmu.h

gatedesc is a C structure for the Gate entry in IDTR.

in gatedesc: we have cs and off

uint cs: 16 means cs is of 16 bit size.

uint dpl: 2 means dpl is of 2 bits.

Kernel has to put such gate descriptors in IDT array

Now go to tvinit function in trap.c.

This tvinit function is called from main(). main is the kernel initializer function.

256 is the size of the idt table. idt is an array of struct gatedesc.

SEG_KCODE<<3 is done so as to set CS to 1(as SEG_KCODE has value of 1). the left shift by 3 is done bcoz the CS register has lsb 3 bits occupied for some other purpose.

vectors[i] is address of function to be exectuted.

WHat is there in vectors[i].

extern uint vectors[] : extern means vectors[] is a global variable but is defined in another file.

Linking not only means linking call of a function to code of a function, but also to link a variable with its reference in another file.

vectors[] array is present in vectors.S assembly file.

T_SYSCALL is actually 64. Entry number 64 is actually being overwritten! Only the Second field is 1, and last field is DPL_USER.

2nd field in SETGATE is istrap.

last field in SETGATE is dpl

see dpl in gatedesc: dpl actually means the privilege level. It is 0 or 1

and DPL_USER is 0x3. WHich means that the 64th entry in idt has privilege level of user!

Open vectors.S file. Search for the array called vectorr on line 1280

```
vectors:
        .long vector0
        .long vector1
        .
        .
        .
```

What this means is that vectors is an array, and .long vector0 is actually the 32 bit address of vector0

vector0,1,2... are interrupt handlers

These interrupt handlers are pushing 0 first, then they are pushing their particular number. And then code jumps to alltraps.

CPL - Current Privilege Level

DPL - Privilege level in the CS register:

%cs <= DPL means it is checking if we are jumping to a lower privilege level.

Load %ss and %esp from a task segment descriptor.

We basically have to run code of the kernel on an interrupt. Hence we have to load ss and esp for the kernel. So task segment descriptor gives us these values of the kernel. During initialization of the kernel, this task segment descriptor was setup.

TS descriptor - task segment descriptor.

See on slide number 23 of xv6 processes and trap handlers:

push %ss

push %esp

push %eflags - flags register

....

This ss, esp, eflags are basically the registers belonging to the process who generated the interrupt. And these registers are being pushed on the kernel stack!.

All these tasks are done on an interrupt by the HARDWARE!

Set %cs and %eip to the values in the descriptor : Now we will start executing the actual code of vector0, vector1...

The ss,esp,eflags,cs,eip is pushed by hardware

The 0,64 are pushed by vector64

This 0 is pushed for storing error number later

System call has to return an error number

64 is pushed so that alltraps understands which trap number is being executed.

struct trapframe is in x86.h

The variables in struct trapframe are in the reverse order as the kernel trapframe!

In struct proc -> we have the pointer trapframe

The struct proc also contains kstack(kernel stack) pointer. This is the kernel stack that we switched to on the interrupt.

SEG_KDATA is 2

In the assembly code we pushed the esp first and then we called trap!

ANd in trap(struct trapframe* tf) function the argument is a pointer.

So according to calling convention, the tf now points to the esp! So now tf can access all the elements in the trapframe.

So using the assembly code, we constructed the trapframe on the kernel stack.

Now we are in the trap() function

syscall() is generic function to handle all system calls. syscall() checks the eax register

ideintr means hard disk interrupt

Who set this trapno?

trapno was pushed on the trapframe by vector

When trap() returns, we come into assembly code

popal

popl gs

..

are the reverse of the pops that were done before

int pushes 5 values. iret pops 5 values.

iret reloads the ss and esp with the old values. Now we switch back to the user stack. The CS, Ip are also reloaded. So now we are into our process.

Q. WHy did we create the entire trapframe in assembly and not in C?

- Bcoz there is no way of doing it in C.

Trapframe is essential info needed to handle a trap.

Interrupt only happens after the end of 1 single instruction. Interrupt doesn't happen during the execution of an instruction.

Race condition - multiple concurrent pieces of code trying to change a variable.

The compiler only plays with the general purpose register. Compiler doesn't touch esp, ss, etc.

The code of functions like printf() is present in file like printf.c. But where is the code of system calls like open, read, write?

Go to the file usys.S. There we can see the macro named SYSCALL. It expands the names like fork, exit into a certain piece of code. The gas(GNU assembler) also supports macros. The SYSCALL macro basically calls int 64.

WHat happens after INT 64?

movl $SYS_ ## name, %eax

SYS_ ## name is concatenation

if name is fork, then it is SYS_fork

See what is the value of SYS_fork

The argument is copied into eax

On int, we jump to vectors.S

We will goto vector64

Then we will goto alltraps

Then we call trap()

From trap(), if our call is system call(64 value) then syscall() function is called.

Process_state proc

There is a command called ps. It shows status of processes.

$ ps   -> shows only the processes associated with the terminal

ps -eaf   -> shows all running processes

pid - kernel assigned unique id to process

ppid - parent pid

All processes are created using fork+exec. So all processes have child-parent relationship

We will study /proc file system

$ mount | grep proc

It says that /proc has file system type of proc

/proc is a virtual file system created by linux kernel. It doesn't exist on hardware. It exists only in kernel data structures.

cd /proc

ls

Whatever we see in this folder, is a reflection of kernel's data structure. Each number that we can see in this folder is a pid of a process.

There is a folder for every running process pid=1 is for init process.

Go to the folder corresponding to current bash shell

man proc   - tells us about each file and folder in /proc.

cd /proc

$ cat cmdline

$ vi maps

$ cat environ

In the maps folder, we have virtual memory mapping

There is interesting system call about mmap

$ vi status   # very important file (!)

See the "State" variable in the file

$ ls -l fd

We can see files named 0,1,2,255

These are actually soft links or symbolic links

/dev/pts/2 is the console. Console is the combination of stdin, stdout, stderr.

getpid(): get its own pid

What happens when program does scanf():

read system call is made

Thus process is blocked/made to wait until user types something. The process is sent to a state called Sleeping

sleep() also makes the process wait.

Child and parent can be scheduled any how. First the parent, then the child, vice versa

Orphan is a process, whose parent has died.

Zombie - process that has exited, but its parent hasn't called wait() on it.

If a parent doesn't call wait, then its return status is saved in the kernel data structures until its parent calls wait() on it. So if a parent doesn't call wait(), then the process has died, but it exists in kernel data structures.

Orphan process - executing process, parent has died

Zombie Process - Terminated process, parent is not calling wait() on it. Parent may be alive or dead.

grep State status

ps aux

When parent of a zombie process dies/when process becomes orphan, then init is set as the process's parent. When the computer is booting, the init process does a lot of tasks of setting up GUI, etc. Afterwards the init process just has an infinite loop, and in that infinite loop, it keeps calling wait()

So when we have a zombie process, its parent is made as init, and init will call wait(). So the process will be deleted from the kernel data structures.

goto /proc/[pid]/fd

This shows us all file descriptors

do ls -l

vi fdinfo/3

after close(), kernel cleans up data structure corresponding to the file

cat cmdline

What are files 0,1,2

When program starts running, then stdin, stdout is also manipulated through files

stdin - 0

stdout - 1

stderr - 2

These files are opened by default for our process.

Call to scanf() is a call to read() with 0 as file descriptor

File descriptor number that we get is the FIRST free descriptor number available.

Schedular:

Scheduler runs on behalf of the kernel itself. Not on behalf of the user process. So to make the scheduler code run, it is run on the kstack_Scheduler. kstack_Scheduler is needed bcoz the scheduler also calls many functions.

When scheduler wants to give control to cat, then stack must be changed to kstack_Cat. WHy kstack_Cat? bcoz kstack_cat contains the context of cat!

Goto main in main.c. The bootloader code passes control to main. The function we are going to see now is mpmain(). Before mpmain(), userinit() was called which created the init process. Now we have to schedule init.

Before mpmain(), mpinit() was called, which started initializing the processors. mpmain() will end the initialization of processors and processes will start running.

Goto mpmain(). mpmain calls the scheduler. scheduler makes all the cpus run

Goto scheduler function in proc.c

code of scheduler is C code. When this code was compiled, it was compiled using calling convention.

scheduler should go through the ptable, and select a process which is ready and run it on the cpu.

sti() -> asm is gcc directive. sti() enables interrupts. So far interrupts were disabled.

Q: When did we disable interrupts?

RUNNABLE is READY process in terminology of xv6

We ignore all non runnable process.

In struct proc we had pointer called context. Goto struct context. It is just 5 registers. Q. Why is the context only these 5 registers.

See what is struct cpu

Per-cpu state means , for 1 cpu there is a state.

See struct proc* proc in cpu struct.

In cpu struct, there is an array called gdt. This is a per-cpu gdt table. The gdt table that we saw during boot process was containing only 3 entries.

The variable called scheduler is just a pointer to struct context. Where is that scheduler going to be set? Goto mycpu()

Actually in the swtch(&(c->scheduler), ...) , we are passing the address of the pointer scheduler to the swtch

Why do we pass the address of pointer -> Bcoz swtch() changes the c->scheduler pointer.

p is the process to be scheduled. p's context is already there/

Now goto swtch

swtch is in assembly!!!!!

swtch() was called using calling convention. So read the arguments appropriately

Saving the context -> move values of ACTUAL registers into stack

Loading the context -> move values from stack into ACTUAL regissters,

The called function has to save the callee saved registers? WHy bcoz the called process is about to change them

pushl %ebp

pushl %ebs

.

.

All these are pushed onto which stack?? Ans - Kernel(scheduler) stack!. WHen was this stack setup?? Ans - In entry.S. In entry.S $(stack + KSTACKSIZE) was stored in esp, after that main() was called. So the stack which we are using in all the functions of main() uptil now is the stack setup in entry.S

Always remember that, the compilers like gcc onlu change the general purpose registers. They don't change the registers like esp,ss etc. WE change the registers like esp, in assembly.

So these pushes are happening on the current stack, i.e. the stack initialized using .comm in entry.S

# Switch Stacks

movl %esp, (%eax) # this means that *old = updated old stack

c->scheduler = updated old stack

movl %edx, %esp   # esp = new stack

esp = p->context

Now esp will point to the kernel stack of process to be scheduled.

The pointer of the scheduler stack is stored in c->scheduler.

Now we are popping 4 callee-saved registers -> edi,esi,ebx,ebp. These are popped from the stack of the process now. These will go into the actual registers

Then we have a ret. WHat does ret do? -> it pops one more time from the stack. What value will be there on the staack now? - the value will be EIP register!. EIP will point to the next instruction to be executed for that process. So now ACTUAL eip will be loaded with this value. Now the process will start running.

So after this ret, we jump to the context of the process,

Why is this swtch code in assembly? - bcoz we have to play with the calling convention!

Now see the struct context again. Observe that it contains 5 elements.

edi,esi,ebx,ebp,eip

4 things are getting pushed explicitly. What about eip? - it gets pushed as a part of call

After ret, we go to the process's context. So the function called swtch() in the scheduler is not returning!!! Even if ret is called, its not returning there. Bcoz we have changed the esp!

The scheduler is called only from mpmain(). mpmain() also runs only once. So mpmain() calls scheduler. So who calls scheduler next time?

In the same file proc.c, see the function sched.

It takes arguments as swtch(&p->context, mycpu()->scheduler)

Can you see that these are the exact reverse of the arguments passed to swtch() in scheduler()?

Now we should see which functions are calling the function sched().

Use cscope to find this out:

We see that exit, yield, sleep are calling sched.

exit() calls sched bcoz after a process returns, now scheduler needs to be called to run next process.

sleep() is called when function is about to be blocked

yield() calls sched(). Who calls yield? - Ans trap()!

trap() calls yield if its a timer interrupt!

yield means to give up to someone else.

in yield() -> the state is set to RUNNABLE. RUNNABLE means READY. and then sched() is called.

What is myproc() -> gets the currently running process from struct cpu.

sched() calls swtch(&p->context, mycpu()->scheduler)

As a part of above function call, p->context points to the kernel stack of p where those 5 variables are pushed!

And the esp now points to mycpu()->scheduler. mycpu()->scheduler is the kernel stack for scheduler where the 5 pointers of scheduler context were pushed.

So we now load the context from mycpu()->scheduler. So we load context of the kernel! This is the same context was saved when swtch() was called from scheduler()

So this swtch() doesn't return in the sched() function. Then where does it return? It returns to the scheduler() function!!. Bcoz the eip of the next instruction after swtch() in scheduler() function was stored on kernel stack of scheduler! So again that loop will continue executing and scheduler will pick another process.

Now scheduler() is also going to call swtch(). Where will that swtch() return? Ans - If the process 'p' was earlier context switched out using sched(), then it returns in sched()!!! after the swtch() in the sched!

So we found one way a process can return. It will return from the sched(). It will goto yield(). And from there it will goto trap()

Who is calling sleep()? Ans - After read(), scanf(), etc

Sleep also calls sched.

When is kernel stack for process created? - WHen the process is forked.

WHy each process has its own kernel stack? - bcoz context of process cannot be stored on the application stack of the process. Bcoz application code can do malicious things with it. Kernel stack of process is not accessible to the application code, it is only accessible to the kernel code.

A series of events to better understand the swtch() calls in scheduler() and sched():

Consider a P1 process

Timer interrupt comes

User stack -> Kernel Stack change happens in hardware. See slide number 23 of 12th ppt

ss and esp of user stack stored on kernel stack

Jump to vector64

alltraps function called

trapframe built

goto trap()

goto yield() // as timer interrupt occured

goto sched()

swtch()   -> As a part of swtch() function call, the eip of next instruction after swtch() is stored on kernel stack of p1

The above swtch() returns after the swtch() of scheduler()!

we reach to statement in scheduler() after swtch():

Now scheduler() will schedule some other process. But there will come a point when scheduler() has to schedule the P1 process! AS it was marked RUNNABLE in yield(). So assume that the time has come now to schedule the P1 process.

So scheduler() will call swtch(). Now the esp will change to the kernel stack of P1. ret will pop and new value of eip will be set.

Important Question: Where does this ret take us? - It takes us to the swtch() in sched()!! Bcoz that was the last eip value pushed in the kernel stack of P1.

Now from there sched() will return

sched() -> yield() -> trap() -> alltraps.

From alltraps all the values in the stack that were built as a part of trapframe will be popped. So now esp and ss will point to the User stack of P1.

Now the code of P1 will start executing from where it began.

Mem_management

It is job of MMU(hardware) to detect memory violations.

Static Linking:

Problem with Static linking is that, each C program will have separate standard C library code as a part of it.

"stub" code - placeholder

Using dynamic linker, the executable file is no longer bigger.

PLT is a section in the ELF file.

In Dynamic Loading, only the functions which need to be executed are loaded. For eg in the beginning only main() might be loaded and later as req functions can be loaded.

The Dynamic Loader needs to understand Dynamic Linking, as the ELF file will contain Dynamic linking stubs!. So loader needs to understand dynamic linking.

Why link-loader? - Bcoz at the time of loading, loader must perform linking.

Holes in memory - free spaces in memory.

External Fragmentation problem - the total free size is greater than required

but no single hole greater than reqyured.

Internal Fragmentaiton - Space wasted

How small ? - Generally 4KB.

Paging is an extension of the Fixed Partition scheme.

Paging: Process is considered as a logically continuous chunk. However, physically its not continuos in RAM.

size of page = size of frame.

Page table is an in-memory continuous page table.

From the page table, we get a frame number.

Compiler assumes that process is stored in a continuous chunk in memory.

Compiler can even assume that process is present from 0 address.

Q. Do different processes have different page tables?

Grey coloured blocks are free. The new process is saying that I need 4 pages.

Global DS -> Free Frame List

Process specific - Page Table DS

Why 2 memory accesses? - One through the Page Table in RAM, One to the actual address.

TLB:

When address is issued, page number is searched first PARALLELY(constant search) into TLB.

For TLB hit-> 1 memory access

For TLB miss -> 2 memory accesses.

TLB can be used like Least Recently Used, Least Freq used. The TLB updation is done entirely in the HARDWARE. The hit and miss also is done in the HARDWARE.

miss ratio = 1 - hit ratio

Q. How big is the page table in size? - Ans - Depends on the Process. Number of entries in the page table = Size of process / Page size. But here page table size varies from process-process.

The valid, invalid bit can be used to detect memory violation. If address is given which refers to invalid bit, then it is a memory violation.

In the PTE: Why don't we need All 32 bits for storing the Physical Page number? - Bcoz we don't have 2^32 pages!. If page size is 4KB, = 2^12, then we only need 20 bits to point to a page number. Bcoz there will only be 2^20 pages.

Shared Pages diagram:

Instead of allocating memory for library multiple number of times, we use paging. All process1, process2, process3 want to access the C library.

The Kernel allocates only 1 instance of the library into the memory. Other processes will have the same mapping to the shared library as the process for which library was loaded.

That's why on linux system, library is named as Shared Library

Homework: goto /proc. Goto maps file for each process. Locate Virtual address for C library for each process. Deduce something from it.

Major problem is that page table needs to be continuous in memory.

One solution to the above problem is to increase the offset. Thereby we increase the page size. So the number of page table entries will be reduced. But this leads to more internal fragmentation.

p1 p2 d

p1 is index in 1st page table

p2 is index in 2nd page table. Address of 2nd page table is taken from 1st page table.

d is index in final physical memory. It's address is given by 2nd page table.

PROBLEM: Each hierarchical level in paging, leads to 1 more memory access!

Solaris contains Hashed Page Tables.

Basic Idea of hashing is to reduce search time by using a function which gives us an index.

p is used as input to the hash function.

With hashing scheme, there is a problem of collision.

Here the chaining soln is shown.

s is page number for q, r is page number for p.

Motivation for Inverted page table.

Process p1,p2,p3 all have different page tables pointing to physical RAM.

Why not have a single 'Frame' Table which maps from the Physical Memory to the Process?

- Frame - Physical Memory

- Page - Virtual Memory.

Instead of Page table, we have a frame table here.

In the frame table, if we index with a Frame number, then we get page number

This inverted page table is ONE GLOBAL table. No per-process table needed.

Problem: Here we cannot use Index in the page table. As we have Page number with us, we don't have frame number with us.

So we have to Search with a Page number and a PID!. Why PID? - Bcoz that particular page number will belong to only 1 process.

The index of this entry is the Frame number!

Oracle SPARC SOLARIS has 2 page tables. 1 for kernel and 1 for process.

span = number of continuous free frames.

TLB - present in all processors irrespective of Inverted page table or normal page table.

Oracle SPARC has 3 level of hierarchy.

If we don't have sufficient memory in RAM, we are not running all proceses at the same time! So some process can be removed and put into a Hard Disk.

Process stack and heap also has pages.

Demand Paging

Virtual memory - de-facto implementation of memory in current OS

Paging is extension of fixed size partitions.

Virtual Memory != Virtual Address

Setting up MMU is job of kernel.

Virtual address is the addresses written by the compiler.

Uptil now we assumed that code and data needs to be in memory for program to execute. But entire program is rarely used.

Using virtual memory, actual program size can be larger than the physical memory.

User logical memory - view of the compiler while generating program.

eg. for 64 bit machines, compiler can assume that the program has access to 2^64 Bytes of memory.

Less I/O needed to load or swap program - as program loaded in the memory is just a small part.

Virtual memory can be implemented via demand paging or demand segmentation.

Virtual memory is the view taken by the compiler.

In the memory map, pages are mapped to Physical memory and also to The Backing store(HDD)!

sparse address space - allocate only 1 page for stack and 1 page for heap initially. There are many holes. These holes can be used earlier to map dynamically linked libraries, etc.

Now compiler also can assume that there will be holes available.

Shared memory can also be implemented like shared library using virtual memory.

sbrk() is a system call which asks kernel to allocates a region in Virtual memory and return a pointer to it.

Demand Paging: Load page on demand

Hence page which is not needed, won't be loaded.

Page fault refers to raising a hardware trap.

Page fault is one of the most difficult codes to write.

Get empty frame - As kernel needs to load needed page here.

Reset tables - reset page table to make it as "valid"

One instruction leading to multiple page faults:

eg add addr1, addr2

So one instruction is causing 3 page faults, one for 'add' code, 2 more for accessing data.

Q. What is locality of reference?

CISC processors - can give complicated instructions like moving arrays.

movarray 0x100, 0x110, 20

Until page is being loaded, the current process can be sent to the waiting queue.

Optionally there can be 1 I/O or 2 I/Os for loading a page into memory.

1 I/O for reading a page is compulsory.

If 1 access out of 1000 causes a page fault, then p = 1/1000

Real challenge of demand paging system -> reduce page faults.

Pages of parent are duplicated - means page frames are duplicated.

vfork() shall only be used if child is immediately going to call the exec.

Write a shell using vfork()

In vfork(), kernel just copies the page table. This is actually not good, bcoz both processes need to have a different stack, heap, etc.

Copy on Write - Only copy if being modified.

Page table is an array/pointer in the PCB

If we map code and shared lib on a swap partition, then we will have a duplication! As those already exist on the disk

Pagable means - that is the table in memory all the time?, or we can optionally load as needed.

Page replacement

We are going to see how to find free frames in this lecture.

Select a page for replacement in such a way that in future minimum number of page faults related to that page occur.

Dirty bit is set by the hardware automatically into the PTE.

from struct proc, there is a pointer to page directory, from page directory, pointers to different page tablese are there.

2 page transfers for one page fault - maybe one page transfer for transferring the dirty page, and another page transfer for taking in new frame.

String is a sequence of page numbers

In the FIFO algorithm example,

There were page faults for 7,0,1

For page number 2 there was a page fault and also a need to do replacement.

Belady's Anomaly - In the middle, the number of page faults is increasing from 3->4.

In optimal page replacement algorithm, we look into the future to see which page is accessed after longest time. But how do we look into future?

LRU - looks into the past.

Disadvantage with LRU -> A time has to be stored with each page. And kernel needs to search the entire page table to find a least recently used page.

LRU stack implementation: On the top: most recently used

On the bottom: least recently used.

Stack Algorithms - It is a category of algorithms. These are not related to LRU stack implementation.

Hardware sets the bit to 1 when page is referenced.

Why the name clock algorithm? - bcoz clock runs in a sequential manner.

Why the name second-chance ? - bcoz hardware marks 1->0 saying that if next time you don't become 1, I will select you!

LFU is problematic bcoz assume that a page has just been used recently and its count is 1. Assume that 1 is the minimum count in the cache. So, LFU is going to replace this page. But this page might be accessed a lot many times in the future.

Pool of free frames - Whenever there is a page fault, A frame is availabele..

Page buffering algorithms - attempt is to spread the writes. Writes are not done immediately on page faults.

Double buffering - eg. we open a file, read data in an array. The file data also present in buffer cache and also in the application data.

OS can give special system calls to applications like Databases. Using this, Kernel doesn't to buffering.

Fixed allocation - fix formula for allocation.

priority allocation - steal a frame from a lower priority process.

Mmap

mmap() is another interface to access files:

It creates mapping betn region of a file and region in the logical layout of the process.

mmap() returns a pointer to the memory region where kernel will load the file. So no read() and write() necessary.

Use the array directly to read or edit the file.

MAP_SHARED - means other processes asking for same data from the file will share the memory. However this shared memory is

representative of a file.

The shared memory here however is a representative of the file.

In /proc: the file 'maps' is the actual memory layout of the process.

This file is a table

We can see where the heap, libc.so mapped.

The code of the dynamic loader also need to be mapped.

When vi also opens the file, we can see the modified changes vi is just another process trying to access the same memoryregion.

munmap() -> removes the mapping.

When a process is loaded, kernel maps code, data, stacjk into the page table.

map() happpens according to a page.

The free memory available in a process, is used by kernel and map it into the page table.

Kernel does lazy writing. I

After read() file position changes.

If we access same file using both mmap() and read(), write(), then the file will exist in 2 places in the kernel data structures.

One in the Buffer cache and one in the memory mapped region.

Data structure of mmap is a page table.

Data structure of read(), write() is a buffer cache.

So Now how do we guarantee Unix Semantics??

How does Linux solve the problem -> The read() and write() now don't go through the buffer cache. They go through the mmap.

The mapping is done in the kernel data structures.

So, the buffer cache is no longer present in linux for read() and write().

LAb task: Write programs mixing mmap() and read() and write() and convince urself that Unix semantics are followed.

File-system:

Partitions also known as minidisks, slices

Free space management : bit vector

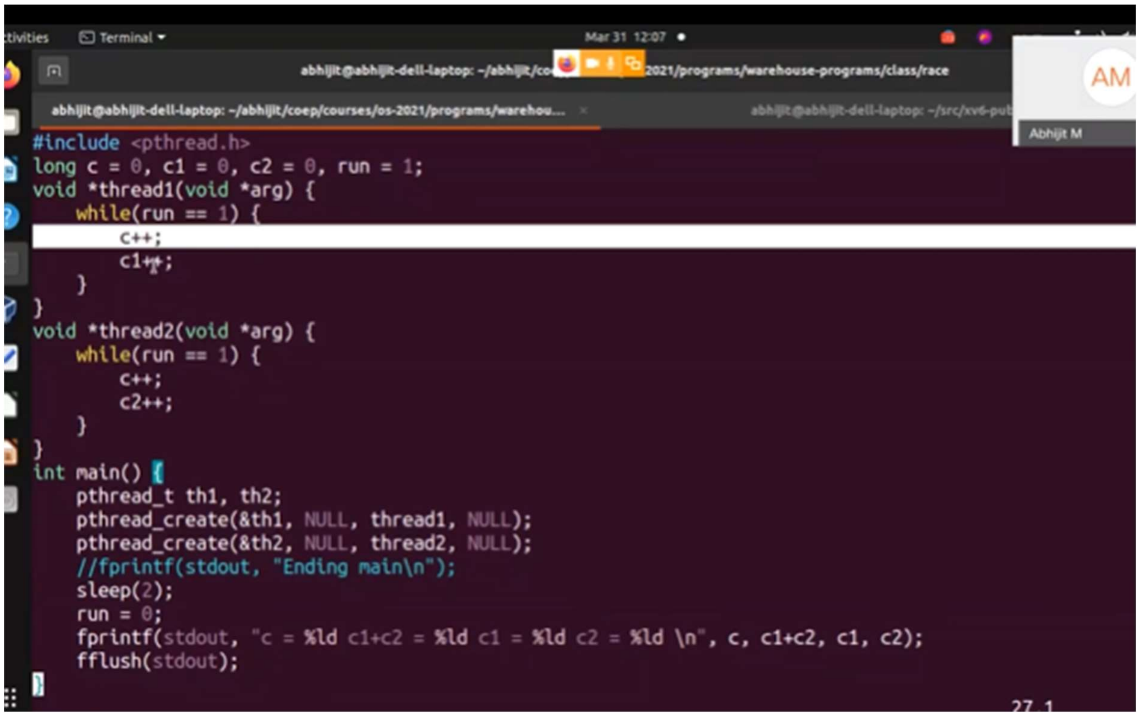
If the block is free, the bit is 1; if the block is allocated, the bit is 0.

Synchronization

OS = ds + synchronization

Demand exception coding skills

Synchronization problem make writing OS code challenging


Terminal
Mar 31 12:07
abhijit@abhijit-dell-laptop: ~/abhijit/coep/courses/os-2021/programs/warehou...
abhijit@abhijit-dell-laptop: ~/src/xv6-pub
AM
Abhijit M

```
#include <pthread.h>
long c = 0, c1 = 0, c2 = 0, run = 1;
void *thread1(void *arg) {
    while(run == 1) {
        c++;
        c1++;
    }
}
void *thread2(void *arg) {
    while(run == 1) {
        c++;
        c2++;
    }
}
int main() {
    pthread_t th1, th2;
    pthread_create(&th1, NULL, thread1, NULL);
    pthread_create(&th2, NULL, thread2, NULL);
    //fprintf(stdout, "Ending main\n");
    sleep(2);
    run = 0;
    fprintf(stdout, "c = %ld c1+c2 = %ld c1 = %ld c2 = %ld \n", c, c1+c2, c1, c2);
    fflush(stdout);
}
                                                                    27,1
```

Compile above program with lpthread

2 threads are created and sleep of 2 second is called in 2 second thread1 will increatment c1 and c while thread2 will increament c and c2 so c should be equal to c1+c2

But, C is not equal to c1 + c2 c is much much less than c1+c2

THIS is becoz race condition between thread 1 and 2 to update c

Reason below

Thread1: mov c,r1

Thread 2: mov c,r1

Thread1:add r1,1

Thread1: mov r1,c

Thread2:add r1,1

Thread2: mov r1,c

Remember registers are different

If you want performance you can not have non-interruptible kernel but can have interruptible but once you make it interupttible race problem starts

2 solution failed then Peterson solution which solves problem but don't have practical approach

Hardware solution

Atomic = non-interupptable

Spinlock does simply test-and-set on lock and waits

Spinunlock set llock to false

Bounded wait M.E. with T&S

Thumb rules of spinlock:

Never block a process hoiding a spinlock

Hold A spinlock for only short duration

Preferrable on multiprocessor

Short < = context switches

Initially spinlock is set to false i.e. no one is acquiring the process

Then we need to acquire the lock

In acquire lock there is pushcli(disable interrupt to avoid deadlock) what does it do push the flags and give you the flags then calls cli cli disable the interupts

It is ebp pointer that points to the base of stack frame

In acquire sleep you are holding spinlock and you are acquiring sleep acquiring process table lock and releasing already held lock then youre calling sched In sched your are getting current process it is actually checking if you are holding ptable lock it is expected you are holding ptable lock checking if ncli is not equal 1 you should hold only one lock switch to schedular context then switchuvm

Sleeplock is used only in two places struct buf and struct inode

Scheduling under 1 and 4 is preemptive

Throughput = total no processes completed/total time

First come first serve algo is non preemptive that means there should be no interrupt

FCFS suffers from convoy effect

Shortest job first scheduling

Each of verticsl line in gantt chart does 2 context switches

In Non-uniform Memory Architecture NUMA every processor has its own memory asymmetric access to memory it will taje long time

Entry.S basically combones 2 things loads the gdt table Turn oon paging

Spinlock is very crticak need of synchronizaition

Cycle is necessary condition but not sufficient

Log_lock will be before the ide_lock

Classical synchronization problem

Bounded buffer problem solved using semaphore

Wait(empty) //enter all N producer process allowed parallely

Wait(mutex) //only one producer can active in queue

Initial value of full is 0