

slides2

bootloader is located in bootmain.c , bootasm.S ⇒ bootblock

loaded in RAM at 0x7c00

start is location0x7c00

cli → disable interrupts

in the original scheme, we had 16 bit segments, and got 20 bit address, with 80286 processor, you will have 2mb i.e 21st bit for address.

they used an existing pin for address, which will be enabled/disabled . by default disabled.

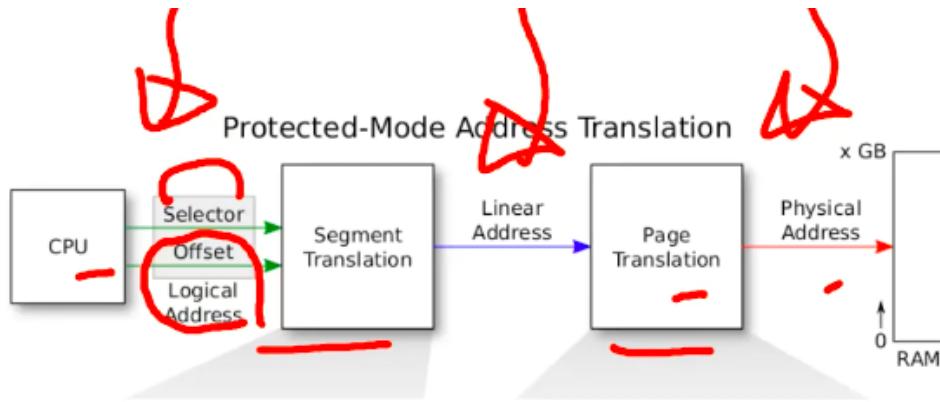
```
abhi@abhi-laptop: ~/src/xv6-public          abhi@abhi-laptop: ~
38
39 # Switch from real to protected mode. Use a bootstrap GDT that make
40 # virtual addresses map directly to physical addresses so that the
41 # effective memory map doesn't change during the transition.
42 lgdt gdtdesc
43 movl %cr0, %eax
44 orl $CR0_PE, %eax
45 movl %eax, %cr0
46
47 //PAGEBREAK!
48 # Complete the transition to 32-bit protected mode by using a long j
49 # to reload %cs and %eip. The segment descriptors are set up with no
50 # translation, so that the mapping is still the identity mapping.
51 ljmp $(SEG_KCODE<<3), $start32
52
53 .code32 # Tell assembler to generate 32-bit code now.
54 start32.
```

in protected mode, segmentation and paging both will be enabled.

segment registers are now used as index in segment descriptor table.

descriptor table consists of base limit pair.

the table is located in memory, a special register gdtr will give the address of segment descriptor table, who will store it in gdtr? OS will



Both Segmentation and Paging are used in x86
 X86 allows optionally one-level or two-level paging

Segmentation is a must to setup, paging is optional (needs to be enabled)
 Hence different OS can use segmentation+paging in different ways

Heirarchical Paging

Paging concept, hierarchical paging

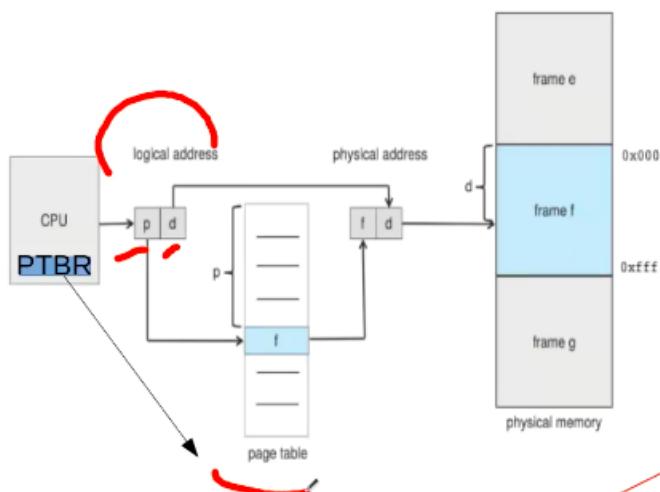


Figure 9.8 Paging hardware.

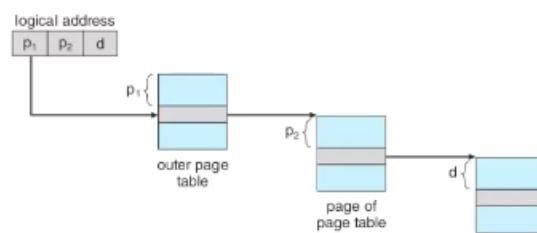
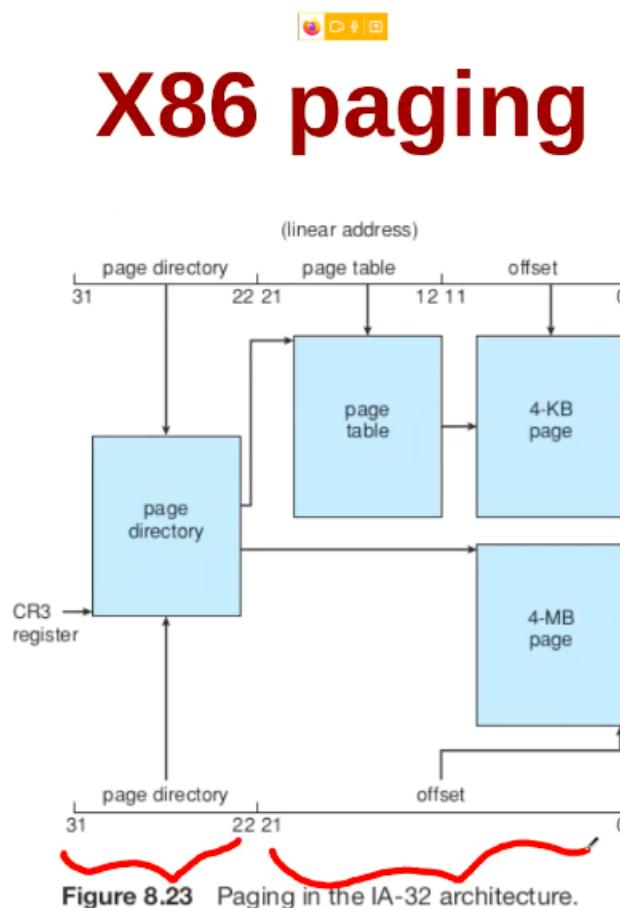


Figure 9.16 Address translation for a two-level 32-bit paging architecture.

suppose we have 4gb of memory and 4kb page size then we have $2^{32}/2^{12} \Rightarrow 2^{20}$ entries in page table, which is way too much and hence we have hierarchical page table



pagesize= 2^{offset} bytes

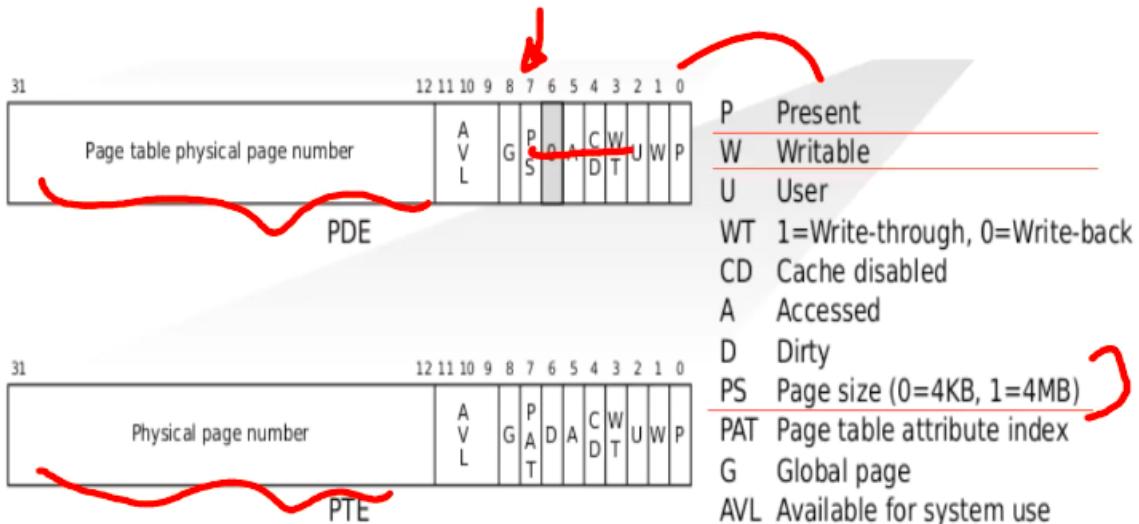
so for one level paging , page size is 2^{22} is 4mb

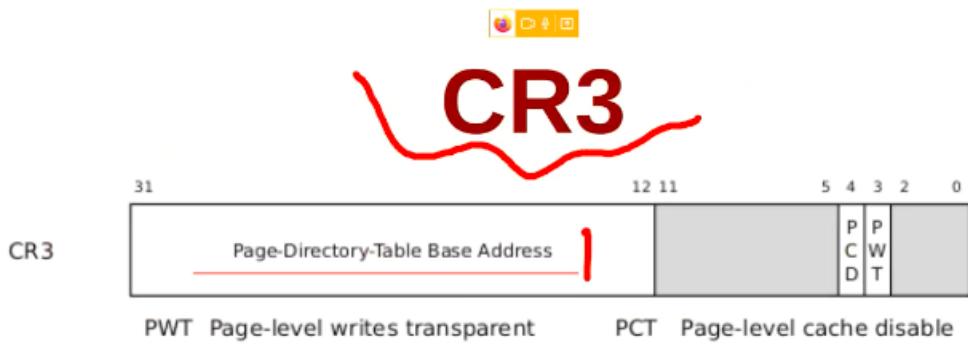
for 2 level paging, page size is 2^{12} i.e 4kb size

CR3 gives the base of page directory table

Page Directory Entry (PDE) Page Table Entry (PTE)

Abhijit

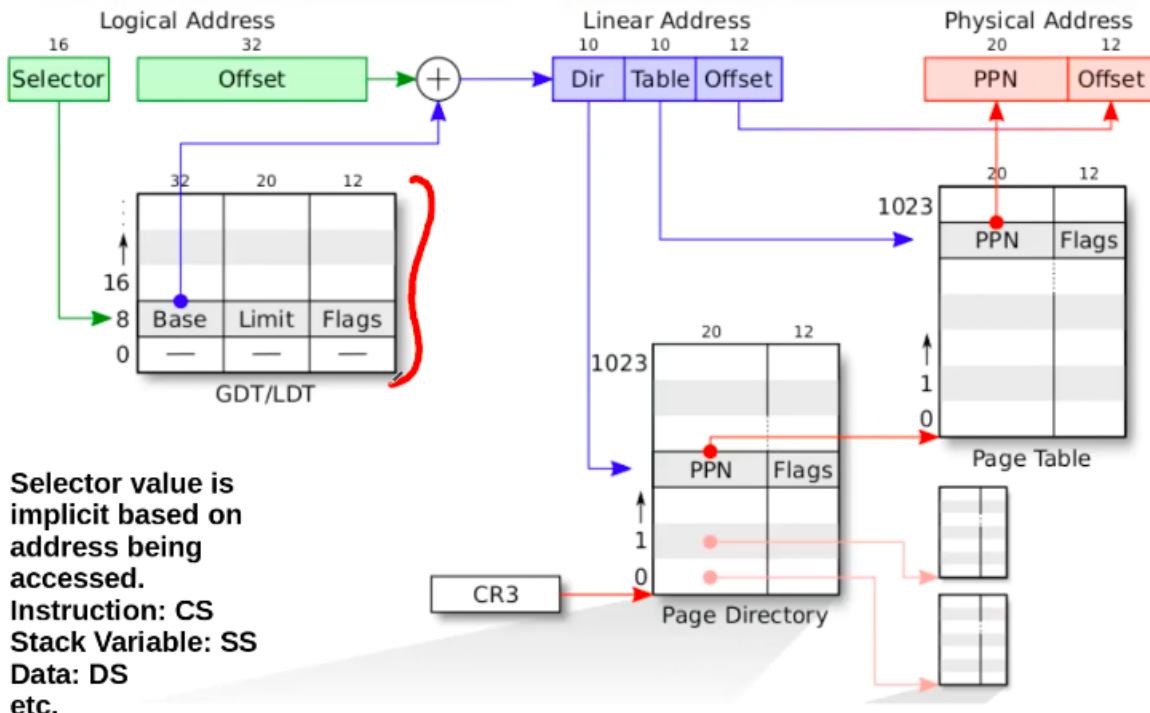




CR4

CR4	31	11 10 9 8 7 6 5 4 3 2 1 0
		O O P P M P P D T P V
		S S C G C A S E D S V M
	X F E E E E E D I E	
	M X	
VME	Virtual-8086 mode extensions	MCE Machine check enable
PVI	Protected-mode virtual interrupts	PGE Page-global enable
TSD	Time stamp disable	PCE Performance counter enable
DE	Debugging extensions	OSFXSR OS FXSAVE/FXRSTOR support
<u>PSE</u>	<u>Page size extensions</u>	OSXMM- OS unmasked exception support
PAE	Physical-address extension	EXCPT

Segmentation + Paging



Role of kernel lies in creating gdt(segmentation) table, making gdtr point to gdt, setting up page directory, page table and making cr3 point to base of page directory, once kernel does this memory management is done.

xv6

xv6 configures the hardware by setting base=0, limit =4gb. so linear address is same as logical address, this is called identity mapping, hence segmentation is practically off.

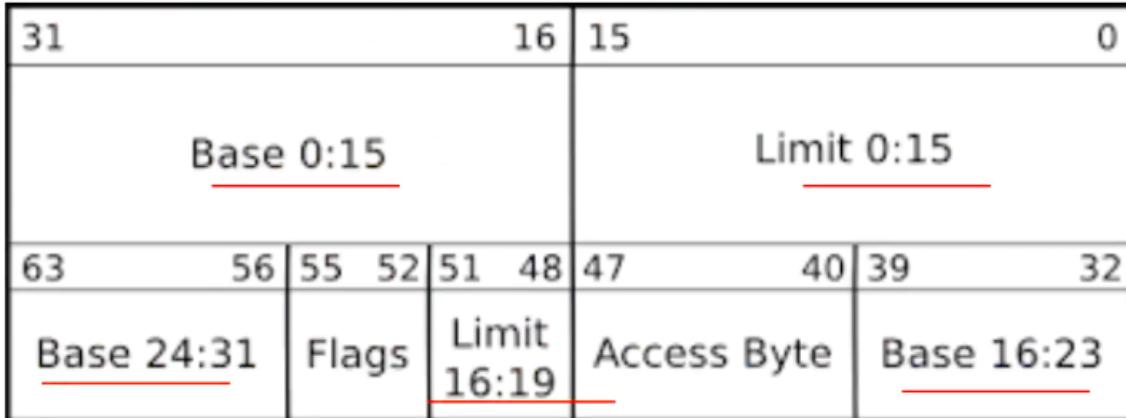
and limit is 4gb, which is maximum addressable space as $4\text{gb} = 2^{32}$, and address is 32 bit, so no address is out of bound.



Segmentation + Paging setup of xv6

- **xv6 configures the segmentation hardware by setting Base = 0, Limit = 4 GB**
 - translate logical to linear addresses without change, so that they are always equal.
 - **Segmentation is practically off**
- **Once paging is enabled, the only interesting address mapping in the system will be linear to physical.**
 - In xv6 paging is NOT enabled while loading kernel
 - After kernel is loaded 4 MB pages are used for a while
 - Later the kernel switches to 4 kB pages!

GDT Entry

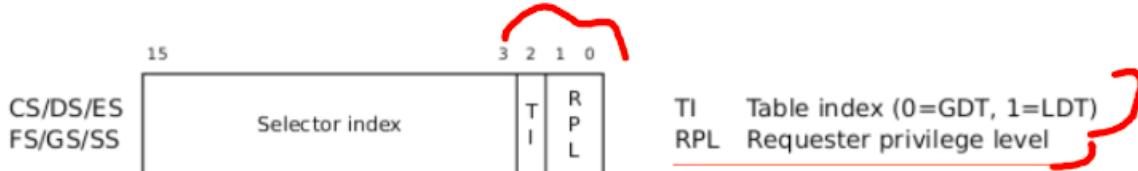

asm.h

```
#define SEG_ASM(type,base,lim)
    .word (((lim) >> 12) & 0xffff), ((base) & 0xffff);      \
    .byte (((base) >> 16) & 0xff), (0x90 | (type)),          \
        (0xC0 | (((lim) >> 28) & 0xf)), (((base) >> 24) & 0xff)
```

each entry is a 64bit entry. to make a entry in gdt table, for particular value of base limit pair, we have to do many bitwise operations for which a macro is defined in the asm.h file., that is SEG_ASM.

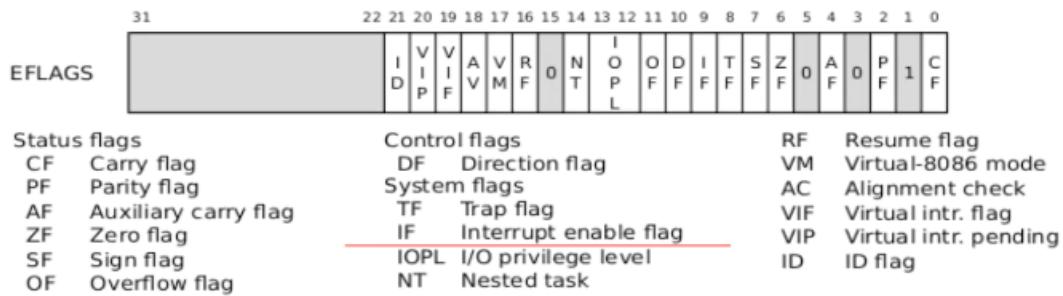


Segment selector



Note in 16 bit mode, segment selector is 16 bit, here it's 13 bit + 3 bits

EFLAGS register



segment selector(CS,SS,DS,ES,FS,GS) now have a different meaning as shown in above image.

GDT stands for global descriptor table, LDT is local descriptor table.

3rd bit of segment selector 0 → GDT, 1 → LDT

2 bits for RPL, which is used to transition from user mode to kernal mode and vice versa.

`in xv6 we only use GDT, HENCE 3RD BIT is always 0`

the entry actually is now going to be 13 bits in Segment Selector.

Lets move to code

`line 42: lgdt gdtdesc`

```

# Bootstrap GDT
.p2align 2                                # force 4 byte alignment
gdt:
    SEG_NULLASM                         # null seg
    SEG_ASM(STA_X|STA_R, 0x0, 0xffffffff) # code seg
    SEG_ASM(STA_W, 0x0, 0xffffffff)       # data seg

gdtdesc:
    .word  (gdtdesc - gdt - 1)           # sizeof(gdt) - 1
    .long   gdt                          # address gdt

```

fig 1: for code

gdt table is hardcoded and has 3 entries here

2nd entry: STA_X → executable permission, STA_R → read permission, for base 0 ,limit 4gb

3rd entry: STA_W → write permission for base 0 ,limit 4gb

gdtdesc is 6 byte data (word → 2 byte, 4byte → long) 2 bytes for storing size of gdt, 4 bytes for storing 32 bit address of gdt table.

```
lgdt gdtdesc
```

```
...
```

```
# Bootstrap GDT
```

```
.p2align 2 # force 4 byte  
alignment
```

```
gdt:
```

```
SEG_NULLASM # null seg
```

```
SEG_ASM(STA_X|STA_R, 0x0,  
0xffffffff) # code seg
```

```
SEG_ASM(STA_W, 0x0, 0xffffffff)
```

```
# data seg
```

```
gdtdesc:
```

```
.word (gdtdesc - gdt - 1)
```

```
# sizeof(gdt) - 1
```

```
.long gdt
```



Igat



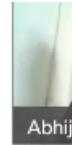
- load the processor's (GDT) register with value **gdtdesc** which points to the table **gdt**
- **table gdt** : The table has a null entry, one entry for executable code, and one entry to data.
- all segments have a base address of zero and the maximum possible limit
- The code segment descriptor has a flag set that indicates that the code should run in 32-bit mode
- With this setup, when the boot loader enters protected mode, logical addresses map one-to-one to physical addresses.
- At **gdtdesc** we have this **data**
 - 2, <4 byte addr of gdt>
 - Total 6 bytes
- GDTR is : <address 4 byte>, <table limit 2 byte>
- So Igdt gdtdesc loads these two values in GDTR

table limit 2 byte is the size of gdt table

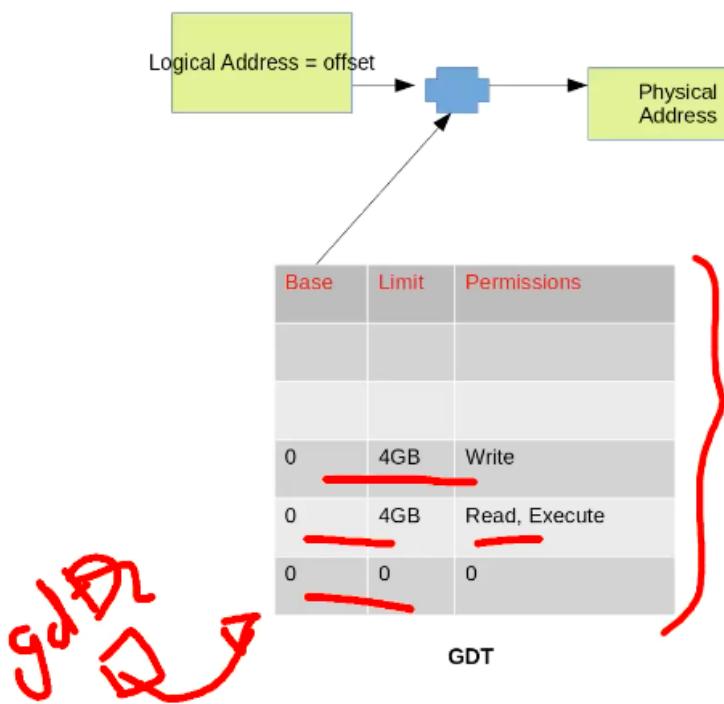
bootasm.S after "lgdt gdtdesc"
till jump to "entry"



Still
Logical Address =
Physical address!



But with GDT in picture
and
Protected Mode
operation



During this time,
Loading kernel from
ELF into physical
memory

Addresses in "kernel"
file translate to same
physical address!

gdtr register points to base of this gdt table. Note protected mode is still not enabled.

CR0

CR0	31 30 29 28	19 18 17 16 15	6 5 4 3 2 1 0
	P C N G D W	A M W P	N E T S E M P E
PE	Protection enabled	ET	Extension type
MP	Monitor coprocessor	NE	Numeric error
EM	Emulation	WP	Write protect
TS	Task switched	AM	Alignment mask
			NW Not write-through
			CD Cache disable
			PG Paging

PG: Paging enabled or not WP: Write protection on/off
 PE: Protection Enabled --> protected mode.

last bit of CR0 register is protected mode enable.

In fig 1: for code we can see 3 lines of code enabling that and then a ljmp instruction.

SEG_KCODE=1, which is left shifted by 3 and put into code segment, and jumps to start32.

why left shift by 3? because last 3 bits of segment selector which are not part of value, they are 2 bit of RPL and 1 bit for table index(TI)

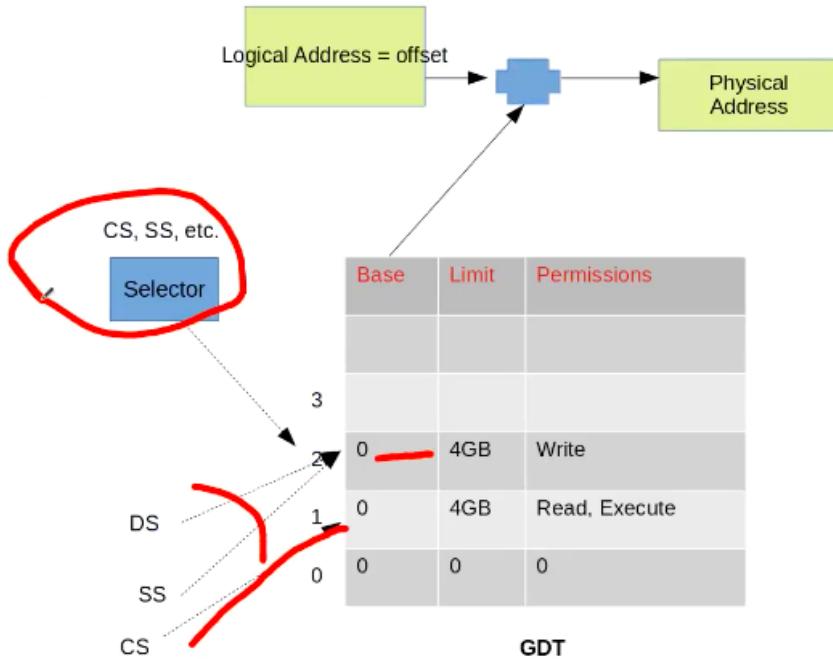
```
abhijit@abhijit-laptop: ~/src/xv6-public
```

```
44 orl    $CR0_PE, %eax
45 movl    %eax, %cr0
46
47 //PAGEBREAK!
48 # Complete the transition to 32-bit protected mode by using a long jmp
49 # to reload %cs and %eip. The segment descriptors are set up with no
50 # translation, so that the mapping is still the identity mapping.
51 ljmp   $(SEG_KCODE<<3), $start32
52
53 .code32 # Tell assembler to generate 32-bit code now.
54 start32:
55 # Set up the protected-mode data segment registers
56 movw   $(SEG_KDATA<<3), %ax      # Our data segment selector
57 movw   %ax, %ds                  # -> DS: Data Segment
58 movw   %ax, %es                  # -> ES: Extra Segment
59 movw   %ax, %ss                  # -> SS: Stack Segment
60 movw   $0, %ax                  # Zero segments not ready for use
61 movw   %ax, %fs                  # -> FS
62 movw   %ax, %gs                  # -> GS
63
64 # Set up the stack pointer and call into C.
65 movl   $start, %esp
66 call   bootmain
67
68 # If bootmain returns (it shouldn't), trigger a Bochs
"bochstrm S" 001 2005C
```

now at line 56, SEG_KDATA is 2 which is left shifted by 3 and put into ax

so DS,ES,SS gets loaded by 2, whereas FS AND GS is loaded with 0. and we are in protected mode.

Setup now



at line 65 we copy \$start i.e. 0x7c00 into stack pointer, \$esp=0x7c00, now stack pointer will grow from 0x7c00 upto 0. boot loader code was loaded at 0x7c00 to upwards, and below that is stack.

then we call bootmain, since it is call instr, it will push the returnvalue into stack at 0x7c00 and now jumps to c code of bootmain. the setup of stack was necessary to execute C program, else we cannot execute C code.

which will load the kernel from disk into RAM and then do paging setup and RUN the Kernel. As of now MMu setup of x86 is done, protected mode, gdt table setup.

why are there different permissions?

we don't want the code segment to be read and execute, we dont want code to be written over.

job of bootmain is to load kernel from disk to memory

bootmain(): already in memory, as part of ‘bootblock’

Abhijit

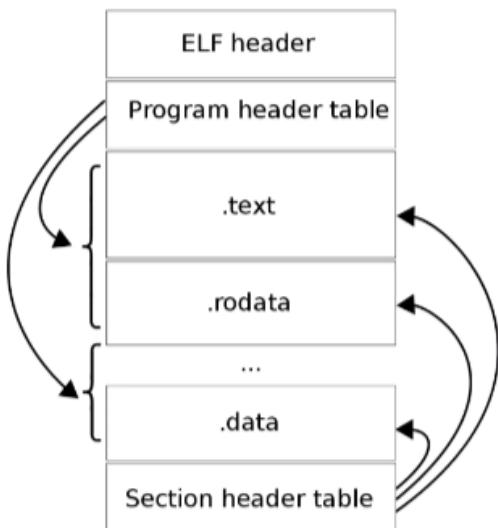
- bootmain.c , expects to find a copy of the kernel executable on the disk starting at the second sector (sector = 1).
 - Why? ↗
- The kernel is an ELF format binary
- Bootmain loads the first 4096 ↗ bytes of the ELF binary. It places the in-memory copy at address 0x10000
- readseg() is a function that runs OUT instructions in particular IO ports, to issue commands to read from Disk

```
void
bootmain(void)
{
    struct elfhdr *elf;
    struct proghdr *ph, *eph;
    void (*entry)(void);
    uchar* pa;

    elf = (struct elfhdr*)0x10000; // scratch
space

    // Read 1st page off disk
    readseg((uchar*)elf, 4096, 0);
```

ELF



```
struct elfhdr {  
    uint magic; // must equal  
    ELF_MAGIC  
    uchar elf[12];  
    ushort type;  
    ushort machine;  
    uint version;  
    uint entry;  
    uint phoff; // where is program  
    header table  
    uint shoff;  
    uint flags;  
    ushort ehsiz;  
    ushort phentsize;  
    ushort phnum; // no. Of program  
    header entries  
    ushort shentsize;  
    ushort shnum;  
    ushort shstrndx;  
};
```

ELF magic is number that identifies that this is a elf file,it is 0x464C457FU #defined in elf.h

size of elf header is 52 bytes

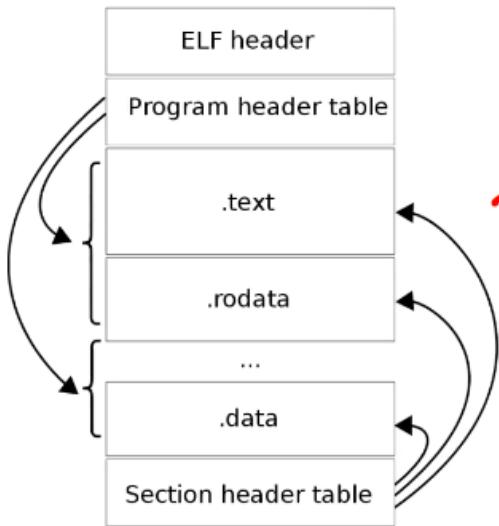
<https://stackoverflow.com/questions/64795450/xv6-bootmain-loading-kernel-elf-header>

entry is the index to first instruction in elf file that should be executed first

phoff, where in the file is program header located

the first readseg() effectively only reads the header + program headers in scratch space at 0x10000(4 zeroes)

ELF



Program header

```
struct proghdr {  
    uint type; // Loadable segment ,  
    Dynamic linking information ,  
    Interpreter information , Thread-  
    Local Storage template , etc.  
    uint off; //Offset of the segment  
    in the file image.  
    uint vaddr; //Virtual address of  
    the segment in memory.  
    uint paddr; // physical address to  
    load this program, if PA is relevant  
    uint filesz; //Size in bytes of the  
    segment in the file image.  
    uint memsz; //Size in bytes of the  
    segment in memory. May be 0.  
    uint flags;  
    uint align;  
};
```



0x80100000 → virtual addr of KERNEL, physical address is 0x00100000

there are 3 program headers of elf file of kernel

Run 'objdump -x -a kernel | head -15' & see this

```
kernel: file format elf32-i386  
kernel  
architecture: i386, flags 0x00000112:  
EXEC_P, HAS_SYMS, D_PAGED  
start address 0x0010000c
```

Program Header:

```
LOAD off 0x00001000 vaddr 0x80100000 paddr 0x00100000 align 2**12  
    filesz 0x0000a516 memsz 0x000154a8 flags rwx  
STACK off 0x00000000 vaddr 0x00000000 paddr 0x00000000 align 2**4  
    filesz 0x00000000 memsz 0x00000000 flags rwx
```

Stack :
everything
zeroes

Code to be
loaded at
KERNBASE +
KERNLINK

Diff
between
memsz &
filesz, will
be filled
with zeroes
in memory

```

abhijit@abhijit-laptop: ~/ebhijit/coep/courses/os-2022/lectures
abhijit@abhijit-laptop: ~/src/xv6-
12
13 #define SECTSIZE 512
14
15 void readseg(uchar*, uint, uint);
16
17 void
18 bootmain(void)
19 {
20     struct elfhdr *elf;
21     struct proghdr *ph, *eph;
22     void (*entry)(void);
23     uchar* pa;
24
25     elf = (struct elfhdr*)0x10000; // scratch space
26     I
27     // Read 1st page off disk
28     readseg((uchar*)elf, 4096, 0);
29
30     // Is this an ELF executable?
31     if(elf->magic != ELF_MAGIC)
32         return; // let bootasm.S handle error
33
34     // Load each program segment (ignores ph flags).
35     ph = (struct proghdr*)((uchar*)elf + elf->phoff);
36     eph = ph + elf->phnum;
37     for(; ph < eph; ph++){
38         pa = (uchar*)ph->paddr;
39         readseg(pa, ph->filesz, ph->off);
40         if(ph->memsz > ph->filesz)
41             stosb(pa + ph->filesz, 0, ph->memsz - ph->filesz);

```

readseg reads the segment according to sector size(512 bytes) and calls readsect, which runs some instr to read content from disk.

the for loop is responsible for loading kernel into memory, pa is the address in the physical memory where kernel is to be loaded and readseg loads that segment into that address.

stosb will fill the gap of memsz-filesz with zeroes.

Jump to Entry

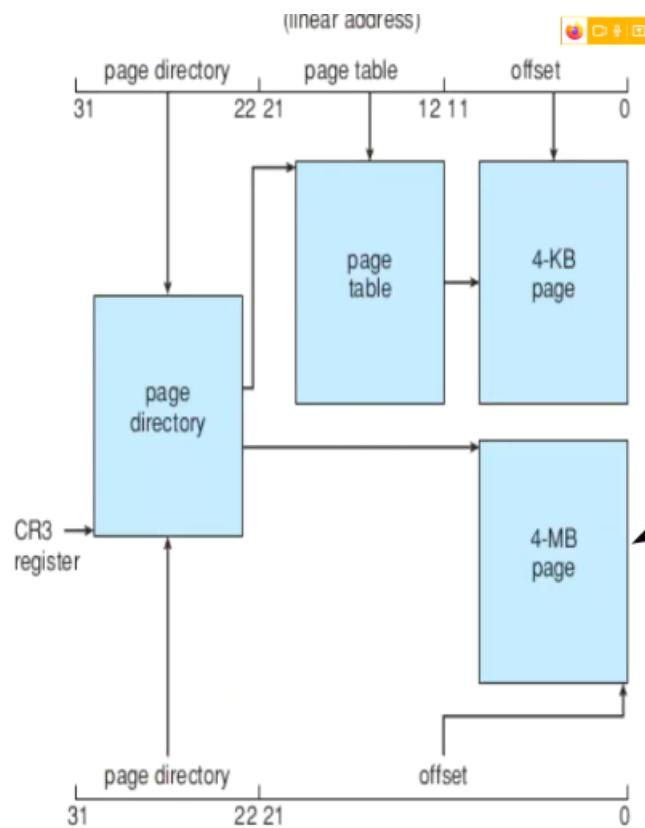
```
// Call the entry point from the  
// ELF header.  
  
// Does not return!  
  
/* Abhijit:  
 * elf->entry was set by Linker  
using kernel.ld  
 * This is address 0x80100000  
specified in kernel.ld  
 * See kernel.asm for kernel  
assembly code).  
 */
```

```
.globl multiboot_header  
multiboot_header:  
#define magic 0x1badb002  
#define flags 0  
.long magic  
.long flags  
.long (-magic-flags)  
  
# By convention, the _start symbol specifies the ELF entry point.  
# Since we haven't set up virtual memory yet, our entry point is  
# the physical address of 'entry'.  
.globl _start  
_start = V2P_W0(entry)  
  
# Entering xv6 on boot processor, with paging off.  
.globl entry 1  
entry:  

```



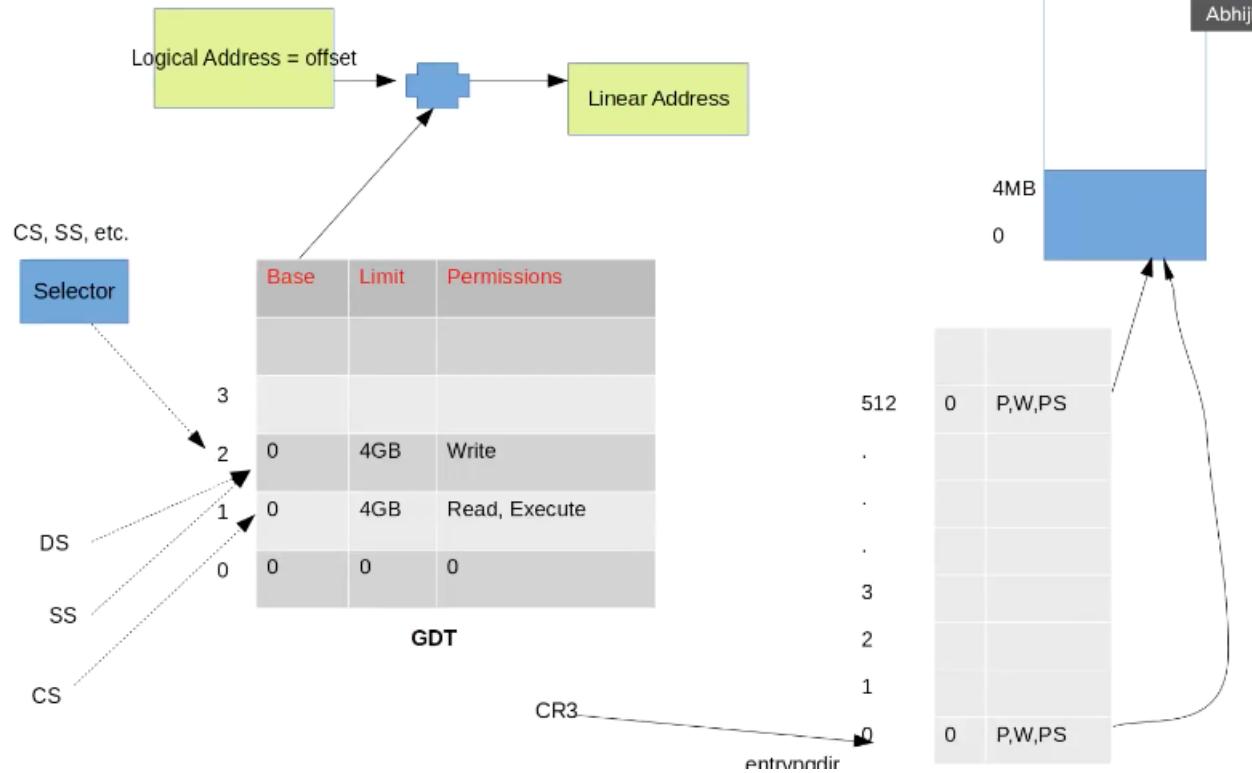
this is the code of entry called in bootmain.c



To understand further code

Remember: 4 MB pages are possible

From entry:
Till: inside main(), before kvmalloc()



how big the kernel code actually is?

we are going to set up 4mb pages, both 0 and 512 entry in the page directory points to lowest value in the physical memory, and CR3 register points to page directory base table as discussed before. remember we have loaded kernel into 0x100000 address which is the part of this lowest 4mb page that 0 and 512 entry points to.

```
abhijit@abhijit-laptop: ~/abhijit/coep/courses/os-2022/lectures          abhijit@abhijit-laptop: ~/src/xv6-pu
9 # the physical address of 'entry'.
9 .globl _start
1 _start = V2P_W0(entry)
2
3 # Entering xv6 on boot processor, with paging off.
4 .globl entry
5 entry:
6   # Turn on page size extension for 4Mbyte pages
7   movl    %cr4, %eax
8   orl    $(CR4_PSE), %eax
9   movl    %eax, %cr4
10  # Set page directory
11  movl    $(V2P_W0(entrypgdir)), %eax
12  movl    %eax, %cr3
13  # Turn on paging.
14  movl    %cr0, %eax
15  orl    $(CRO_PG|CRO_WP), %eax
16  movl    %eax, %cr0
17
18  # Set up the stack pointer.
19  movl $(stack + KSTACKSIZE), %esp
20
21  # Jump to main(), and switch to executing at
22  # high addresses. The indirect call is needed because
23  # the assembler produces a PC-relative instruction
24  # for a direct jump.
25  mov $main, %eax
26  jmp *%eax
27
28 .comm stack, KSTACKSIZE
```

```

abhijit@abhijit-laptop: ~/abhijit/coep/courses/os-2022/lectures          abhijit@abhijit-laptop: ~/src/xv6-pu
7 *(int**)(code-12) = (void *) V2P(entrypgdir);
8
9 lapicstartap(c->apicid, V2P(code));
10
11 // wait for cpu to finish mpmain()
12 while(c->started == 0)
13 ;
14 }
15 }
16
17 // The boot page table used in entry.S and entryother.S.
18 // Page directories (and page tables) must start on page boundaries,
19 // hence the __aligned__ attribute.
20 // PTE_PS in a page directory entry enables 4Mbyte pages.
21
22 __attribute__((__aligned__(PGSIZE)))
23 pde_t entrypgdir[NPDENTRIES] = {
24     // Map VA's [0, 4MB) to PA's [0, 4MB)
25     [0] = (0) | PTE_P | PTE_W | PTE_PS,
26     // Map VA's [KERNBASE, KERNBASE+4MB) to PA's [0, 4MB)
27     [KERNBASE>>PDXSHIFT] = (0) | PTE_P | PTE_W | PTE_PS,
28 };
29
30 //PAGEBREAK!
31 // Blank page.
32 //PAGEBREAK!
33 // Blank page.
34 //PAGEBREAK!
35 // Blank page.
36
37 main.c" 116L, 3264C

```

entrypgdir is an array of pde_t (which is uint → 4 bytes) which has NPDENTRIES(#defined 1024)

$1024 \times 4 = 4096B \rightarrow 4mb$, out of which 2 entries are initialized,

[0] is the 0the entry of the page table

[1] is KERNBASE(virtual address of kernel) right shifted by PDXSHIFT i.e (#defined 22), why? because for 4mb pages 22 bit is the offset and to get the page directory number we right shift by 22 (which results in 512) hence making the 512th entry in the page table also point to the lowest 4mb page in memory

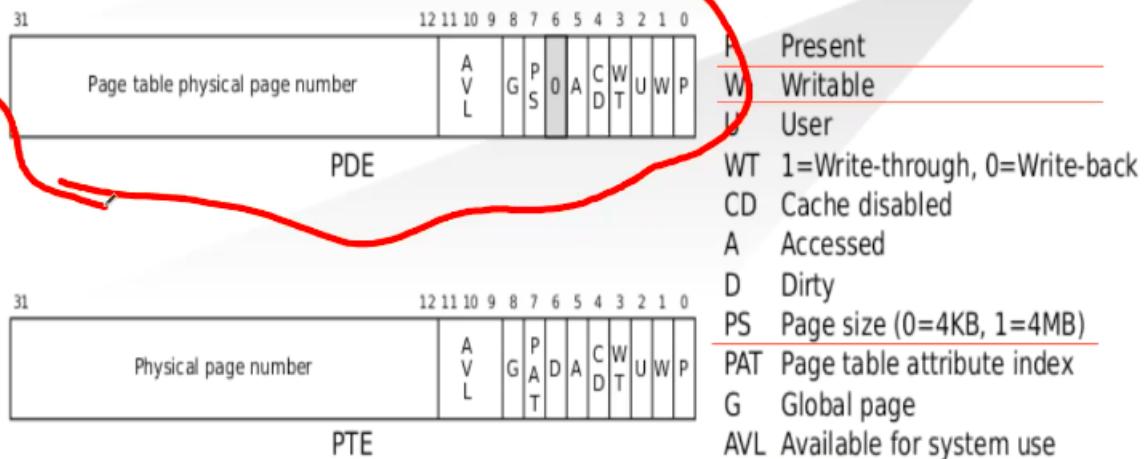
This is entry page directory during entry(), beginning of kernel

Mapping 0:0x400000 (i.e. 0: 4MB) to physical addresses 0:0x400000. is required as long as entry is executing at low addresses, but will eventually be removed.

This mapping restricts the kernel instructions and data to 4 Mbytes.

the value that we are setting is 0 | PTE_P | PTE_W | PTE_PS

Page Directory Entry (PDE) Page Table Entry (PTE)



so we are setting Present(entry valid or invalid), Writable and the PS(page size) bit to 1 i.e making page size to 4mb

NOTE:the stack setup at 0x7c00 is not available, hence we allocate a new stack of 4kb

entry() in entry.S

entry:

```
movl %cr4, %eax
orl $(CR4_PSE), %eax
    • # Turn on page size extension
      for 4Mbyte pages

movl %eax, %cr4
    • # Set page directory. 4 MB
      pages (temporarily only. More
      later)

movl $(V2P_WO(entrypgdir)),
%eax
    • # Turn on paging.

movl %eax, %cr3
    • # Set up the stack pointer.

movl %cr0, %eax
orl $(CR0_PG|CR0_WP), %eax
    • # Jump to main(), and switch
      to executing at high addresses.
      The indirect call is needed
      because the assembler
      produces a PC-relative
      instruction for a direct jump.

movl %eax, %cr0
    •

movl $(stack + KSTACKSIZE),
%esp
    •

mov $main, %eax
    •

jmp *%eax
    •
```

CR4_PSE

V2P_WO converts virtual address of kernel to physical address of entrypgdir to eax which is moved to cr3(substract KERNBASE)

Here we use physical address using V2P_WO because paging is not turned on yet

```

#define EXTMEM 0x1000000          // Start of extended memory
#define PHYSTOP 0xE000000         // Top physical memory
#define DEVSPACE 0xFE000000        // Other devices are at high addresses

// Key addresses for address space layout (see kmap in vm.c for layout)
#define KERNBASE 0x80000000       // First kernel virtual address
#define KERNLINK (KERNBASE+EXTMEM) // Address where kernel is linked

#define V2P(a) (((uint) (a)) - KERNBASE)
#define P2V(a) ((void *)(((char *) (a)) + KERNBASE))

#define V2P_W0(x) ((x) - KERNBASE)    // same as V2P, but without casts
#define P2V_W0(x) ((x) + KERNBASE)    // same as P2V, but without casts

```

CR0_PG means paging(32 bit) to be enabled, CR0_WP → write protect enabled

line 59 moves some value to esp

KSTACKSIZE is 4096 #defined

stack variable is at line 68, .comm stack,KSTACKSIZE which means allocate a data of size KSTACKSIZE which is 4kb

so stack pointer is initialized here once again(of size=4kb), to value stack + KSTACKSIZE

jmp \$main is not done directly, code generated by assembler will be according to particular value of program counter.the indirect call is needed because assembler produces a PC-relative instruction for a direct jump.

Code from bootasm.S bootmain.c is over!

Kernel is loaded.

Now kernel is going to prepare itself

\$main function is in main.c

```
int
main(void)
{
    kinit(end, P2V(4*1024*1024)); // phys page allocator
    kvmalloc(); // kernel page table
    mpinit(); // detect other processors
    lapicinit(); // interrupt controller
    seginit(); // segment descriptors
    picinit(); // disable pic
    ioapicinit(); // another interrupt controller
    consoleinit(); // console hardware
    uartinit(); // serial port
    pinit(); // process table
    tvinit(); // trap vectors
    binit(); // buffer cache
    fileinit(); // file table
    ideinit(); // disk
    startothers(); // start other processors
    kinit2(P2V(4*1024*1024), P2V(PHYSSTOP)); // must come after startothers()
    userinit(); // first user process
    mpmain(); // finish this processor's setup
}

// Other CPUs jump here from entryother.s.
static void
mpenter(void)
{
    switchkvm();
    seginit();
    lapicinit();
```

kernel reinitializes in seginit(),and other init functions

7c00-7dff(512bytes)bootsector ends,+480kb(empty) till 1mb, usme 10000 pe elf header

processes



Process related data structures in kernel code

- Kernel needs to maintain following types of data structures for managing processes
 - List of all processes
 - Memory management details for each, files opened by each etc.
 - Scheduling information about the process
 - Status of the process
 - List of processes “waiting” for different events to occur,
 - Etc.

Process control block is most impo DS for the process

process state
process number
program counter
registers
memory limits
list of open files
...

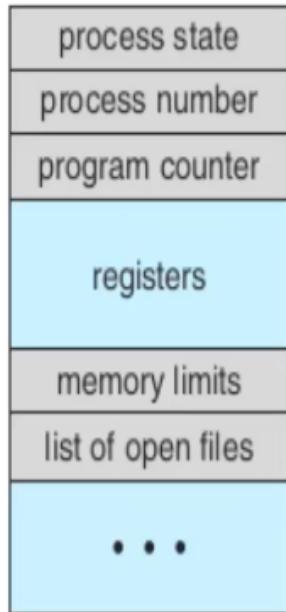
Figure 3.3 Process control block (PCB).

Process Control Block

- A record representing a process in operating system's data structures
- OS maintains a “list” of PCBs, one for each process
- Called “`struct task_struct`” in Linux kernel code and “`struct proc`” in xv6 code



Fields in PCB



- Process ID (PID)
- Process State
- Program counter
- Registers
- Memory limits of the process
- Accounting information
- I/O status
- Scheduling information
- array of file descriptors (list of open files)
- ...etc

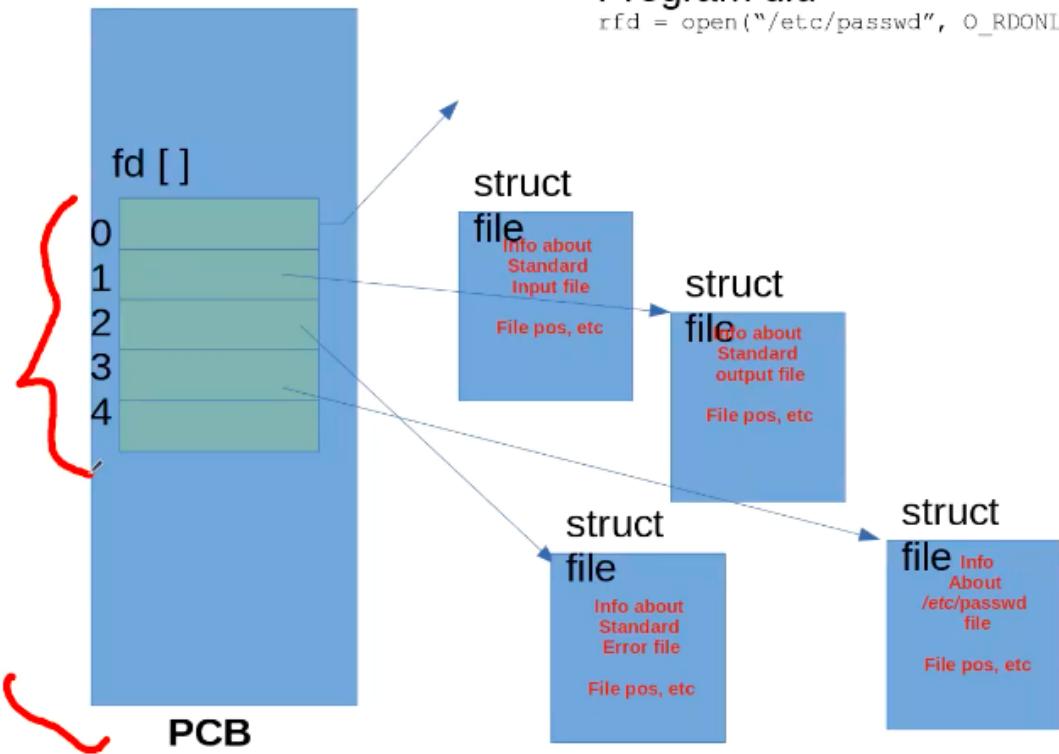
Figure 3.3 Process control block (PCB).

In a multitasking system, process gets switched, hence we need to store registers related to process in PCB.



List of open files

Program did
`rfd = open("/etc/passwd", O_RDONLY)`



file descriptor is just the index of array of pointers in the PCB.

first 3 inputs are stdin,stdout,stderr.

PCB is stored in the kernel memory i.e /proc

PCB in xv6

```

// XV6 Code : Per-process state
enum procstate { UNUSED, EMBRYO, SLEEPING,
RUNNABLE, RUNNING, ZOMBIE };

struct proc {
    uint sz;           // Size of process memory (bytes)
    pde_t* pgdir;     // Page table
    char *kstack;     // Bottom of kernel stack for this
process
    enum procstate state; // Process state
    int pid;          // Process ID
    struct proc *parent; // Parent process
    struct trapframe *tf; // Trap frame for current syscall
    struct context *context; // swtch() here to run process
    void *chan;        // If non-zero, sleeping on chan
    int killed;        // If non-zero, have been killed
    struct file *ofile[NFILE]; // Open files
    struct inode *cwd; // Current directory
    char name[16];    // Process name (debugging)
};

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

```



```

struct file {
    enum { FD_NONE,
FD_PIPE, FD_INODE } type;
    int ref; // reference count
    char readable;
    char writable;
    struct pipe *pipe;
    struct inode *ip;
    uint off;
};

```

ofile is the list of open files of struct file. procstate is zombie/orphan/running/sleeping
in struct file, off is the offset of file

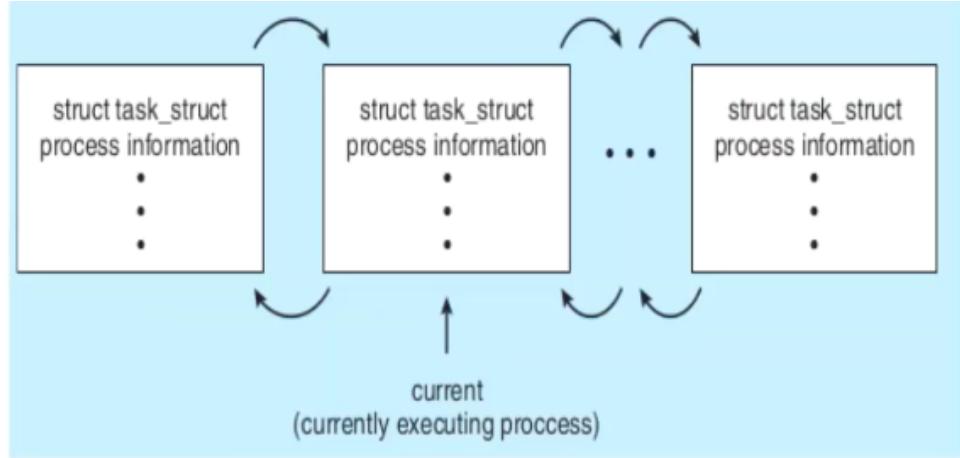
ptable is a global variable which is the main DS for process, ptable.proc[i] will give PCB
for any process. this is only data structure for process in xv6.



Process Queues/Lists inside OS

- Different types of queues/lists can be maintained by OS for the processes
 - A queue of processes which need to be scheduled
 - A queue of processes which have requested input/output to a device and hence need to be put on hold/wait
 - List of processes currently running on multiple CPUs
 - Etc.

in xv6, since there is only 1 array of ptable, then it contains all types of process, whereas some OS might have different queues as described in above image.



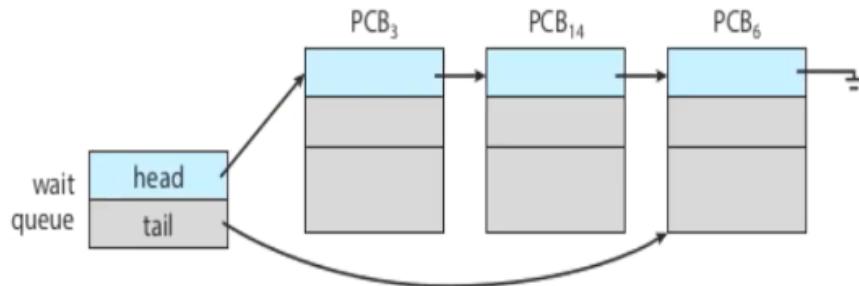
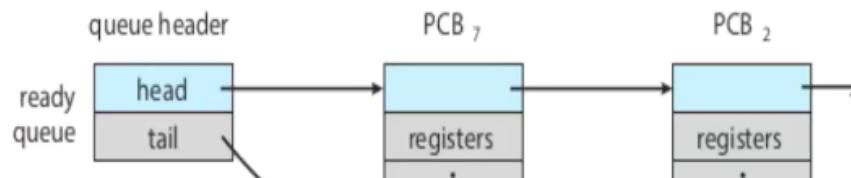
// Linux data structure

```
struct task_struct {
    long state; /*state of the process */
    struct sched_entity se; /* scheduling information */
    struct task_struct *parent; /*this process's parent */
    struct list_head children; /*this process's children */
    struct files_struct *files; /* list of open files */
    struct mm_struct *mm; /*address space */
```

```
struct list_head {
    struct list_head
    *next, *prev;
};
```

struct list_head has next and prev pointers.

suppose many pcbs having list_head children, which are linked, i.e. linking of internal members of pcb



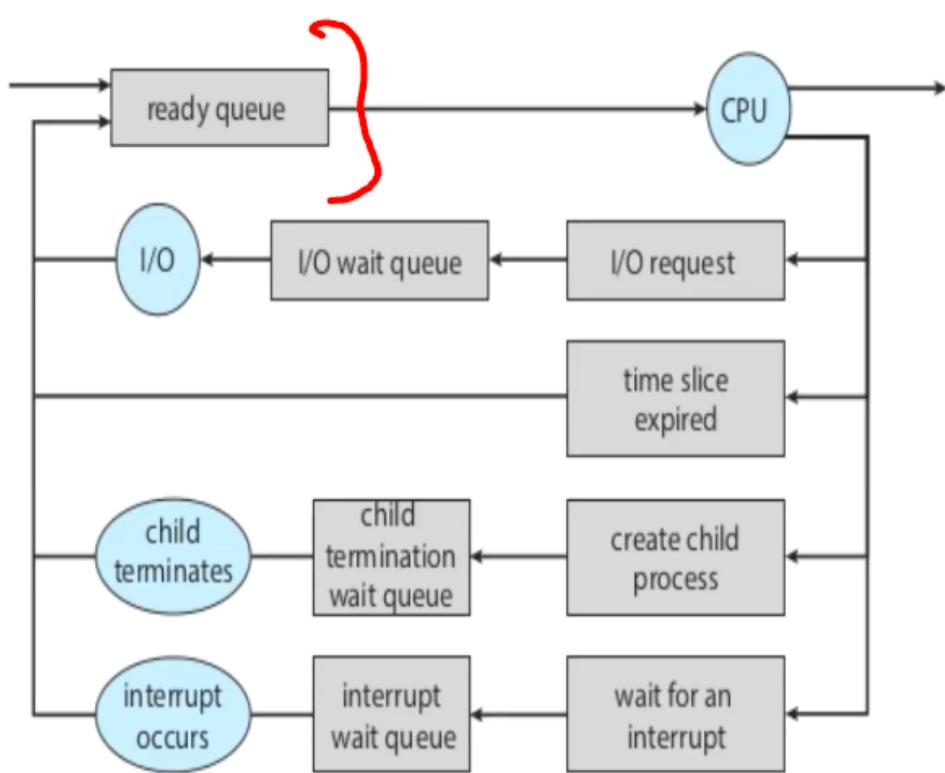


Figure 3.5 Queueing-diagram representation of process scheduling.

scheduler selects 1 process from ready queue and moved to cpu, if time slice expired, it is scheduled in ready queue at back, or if it does i/o it is put i/o wait queue, after I/O is done, it is pushed back in ready queue.

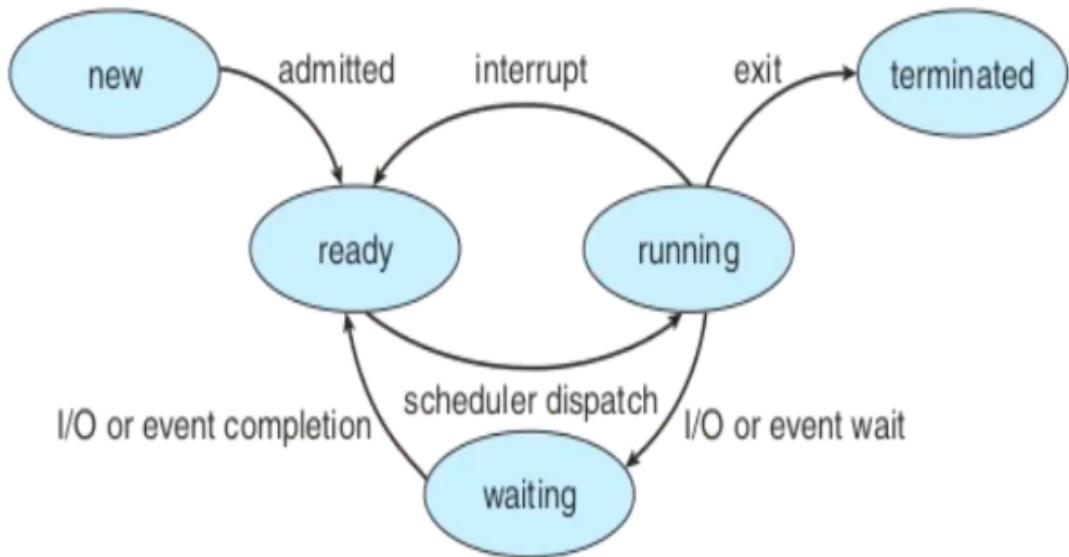


Figure 3.2 Diagram of process state.

Conceptual diagram

a process is in new state, when fork happened. once pcb is completely constructed, it is move to ready queue, and state marked as ready. while running it is in running state, then from running state it has various paths. process can exit only by exit() syscall. for a process to exit, it has to be running first. If timer interrupt occurs it again moves to ready state. if I/o event occurs, it is moved to wait queue (waiting state) when keyboard interrupt is handled it is again moved to ready state.

whatever is not shown in diagram is not possible, i.e. a waiting process can never terminate.

how does user interrupt terminate process(ctrl+c)?

ctrl+c provides a signal called as SIGINT, and then signal handler code gets executed, this diagram has nothing to do with user interrupt. the process is not terminated by exit() then, it is terminated by kernel's code. This diagram is for process's state transition for process's code. similarly for segfault kernal's code handles.

“Giving up” CPU by a process or blocking



```
int main() {  
    i = j + k;  
    scanf("%d", &k);  
}  
  
int scanf(char *x, ...) {  
    ...  
    read(0, ..., ...);  
}  
  
int read(int fd, char *buf, int len) {  
    ...  
    __asm__ { "int 0x80..." }  
    ...  
}
```

```
OS Syscall  
sys_read(int fd, char *buf, int len) {  
    file f = current->fdarray[fd];  
    int offset = f->position;  
    ...  
    disk_read(..., offset, ...);  
}  
  
}  
disk_read(..., offset, ...) {  
    __asm__("outb PORT ..");  
    return;  
}
```

after disk_read returns, interrupt listener is set for disk read ,move the process from ready queue to wait queue and call the scheduler code, this is called blocking.

context switch

execution context of a process includes:-



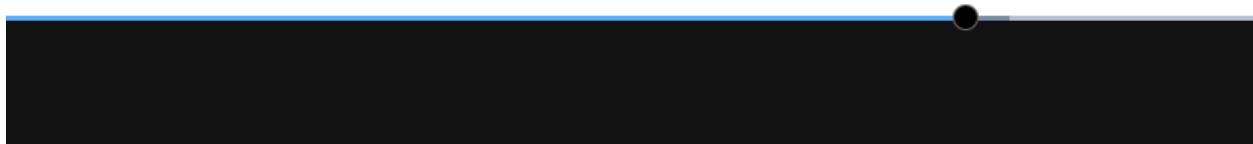
Context Switch

- Context
 - Execution context of a process
 - CPU registers, process state, memory management information, all configurations of the CPU that are specific to execution of a process/kernel
- Context Switch
 - Change the context from one process/OS to OS/another process
 - Need to save the old context and load new context
 - Where to save? --> PCB of the process



Context Switch

- Is an overhead
- No useful work happening while doing a context switch
- Time can vary from hardware to hardware
- Special instructions may be available to save a set of registers in one go



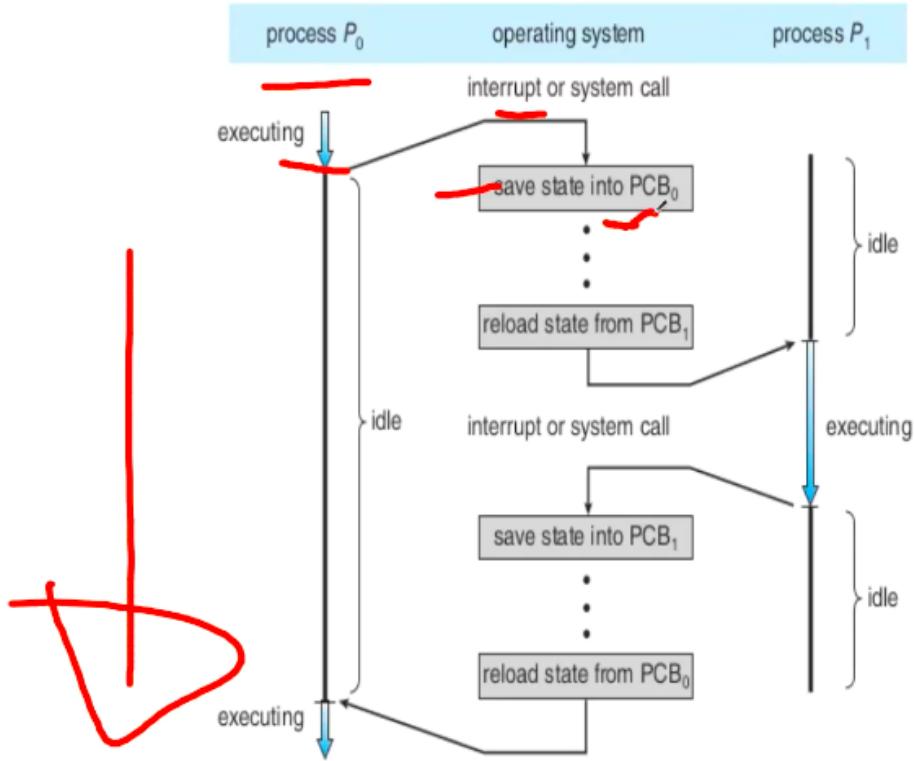


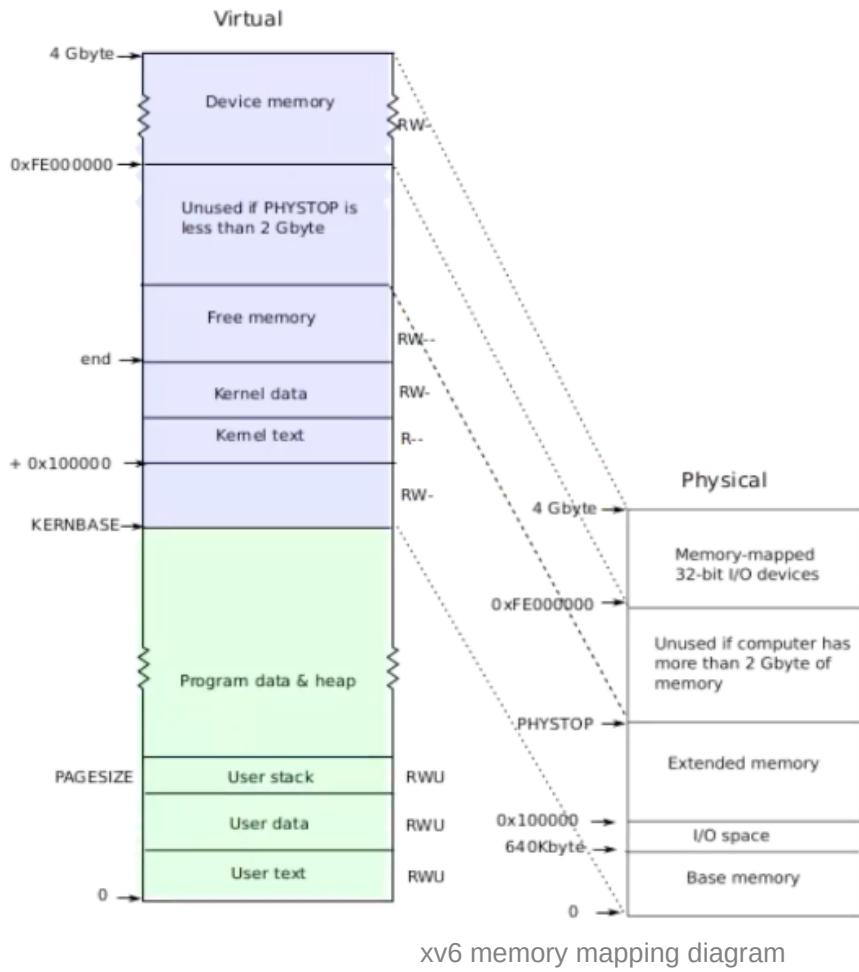
Figure 3.6 Diagram showing context switch from process to process.

Pecularity of context switch



- When a process is running, the function calls work in LIFO fashion
 - Made possible due to calling convention
- When an interrupt occurs
 - It can occur anytime
 - Context switch can happen in the middle of execution of any function
- After context switch
 - One process takes place of another
 - This “switch” is obviously not going to happen using calling convention, as no “call” is happening
 - Code for context switch must be in assembly!

xv6 code



Layout of process's VA space

xv6 schema!

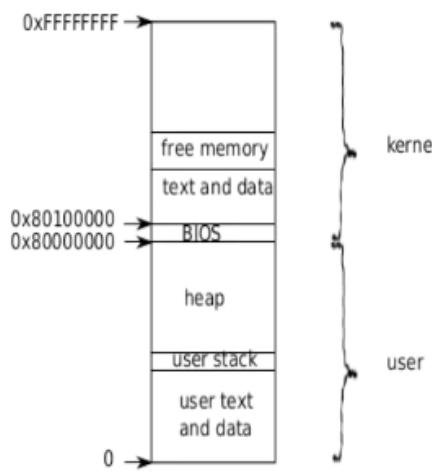
different from Linux

KERNBASE IS 0X80000000 i.e 2gb, kernel code starts from 0x80100000 is 2gb+1mb

PHYSTOP IS 224mb

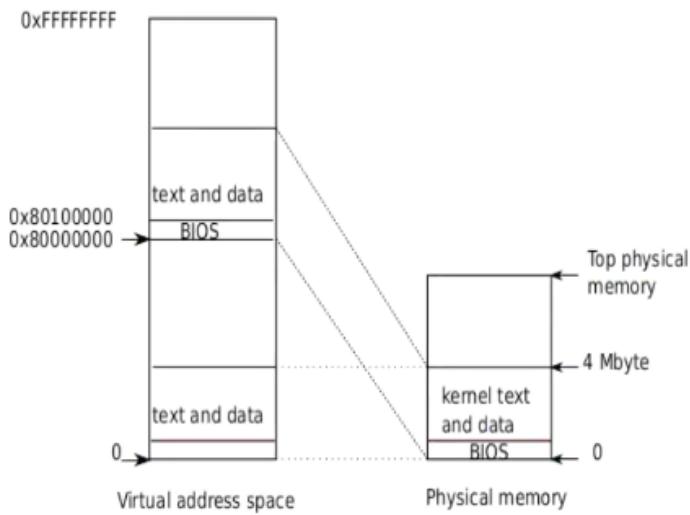
when a process is created and page table is setup the page table needs to have mapping of all these shown in above diagram.

Logical layout of memory for a process



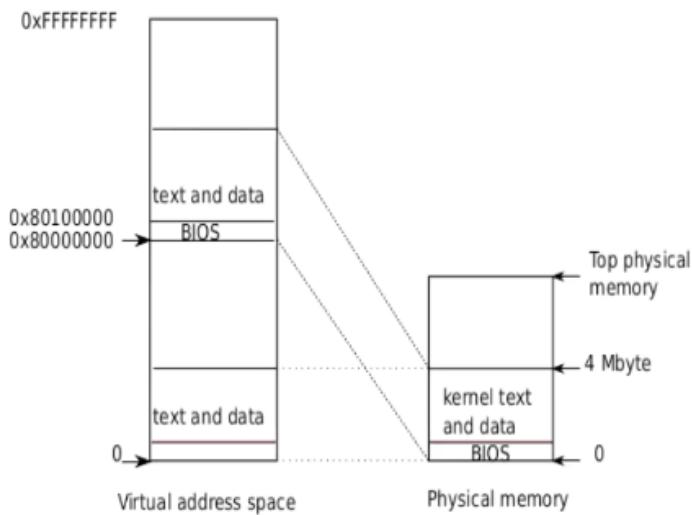
- Address 0: code
- Then globals
- Then stack
- Then heap
- Each process's address space maps kernel's text, data also --> so that system calls run with these mappings
- Kernel code can directly access user memory now

Kernel mappings in user address space actual location of kernel



- Kernel is loaded at 0x100000 physical address
- PA 0 to 0x100000 is BIOS and devices
- Process's page table will map VA 0x80000000 to PA 0x00000 and VA 0x80100000 to 0x100000

Kernel mappings in user address space actual location of kernel



- Kernel is not loaded at the PA **0x80000000** because some systems may not have that much memory
- **0x80000000** is called **KERNBASE** in **xv6**

in xv6 every process has 2 stacks, a user stack, a kernel stack. first for running application code, second for running kernel code(syscalls, exceptions) on behalf of application

Imp Concepts

- **A process has two stacks**
 - user stack: used when user code is running
 - kernel stack: used when kernel is running on behalf of a process
- **Note: there is a third stack also!**
 - The kernel stack used by the scheduler itself
 - Not a per process stack

Struct proc

```
// Per-process state
struct proc {
    uint sz;           // Size of process memory (bytes)
    pde_t* pgdir;     // Page table
    char *kstack;      // Bottom of kernel stack for this process
    enum procstate state; // Process state. allocated, ready to run, running, waiting for I/O, or exiting.
    int pid;          // Process ID
    struct proc *parent; // Parent process
    struct trapframe *tf; // Trap frame for current syscall
    struct context *context; // swtch() here to run process. Process's context
    void *chan;        // If non-zero, sleeping on chan. More when we discuss sleep, wakeup
    int killed;        // If non-zero, have been killed
    struct file *ofile[NOFILE]; // Open files, used by open(), read(),...
    struct inode *cwd; // Current directory, changed with "chdir()"
    char name[16];    // Process name (for debugging)
};
```

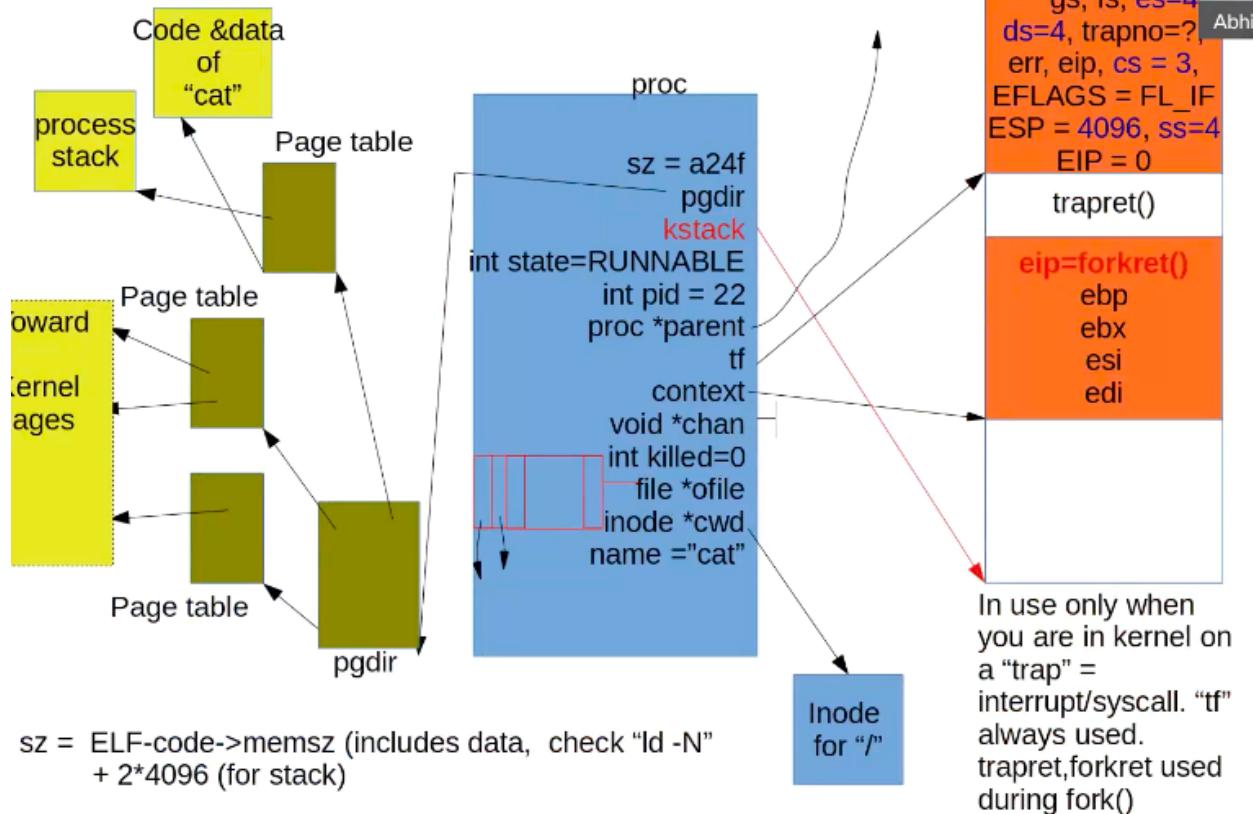
sz is the maximum virtual address, available in the elf file, sz has to be less than 2gb since process in xv6 has to be less than 2gb(KERNBASE=2gb see diagram).

pgdir is a pointer to base of page table, xv6 uses 2 level paging.this pointer will be loaded in cr3

kstack is pointer to the kernel stack(4k in size)

ofile is the array of file pointers pointing to a entry in file descriptor table

struct proc diagram: Very imp!



Memory Layout of a user process

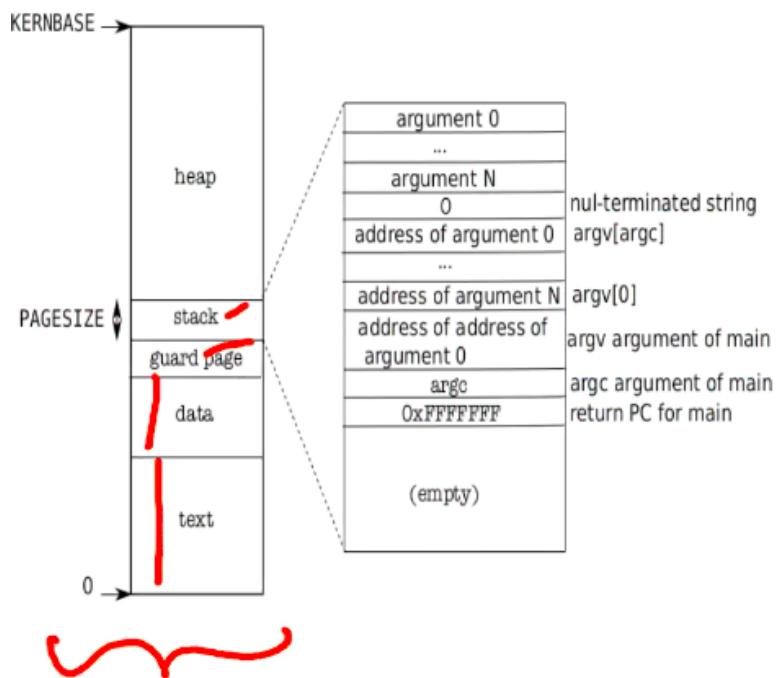


Memory Layout of a user process

After exec()

Note the argc, argv on stack

The “guard page” is just a mapping in page table. No frame allocated. It’s marked as invalid. So if stack grows (due to many function calls), then OS will detect it with an exception



all the pages that get allocated are from the free memory region as shown in xv6 memory mapping diagram. this free memory is around 200mb.

when a process is executing syscall, it is using kernal stack.

Handling traps

Handling traps

- **Transition from user mode to kernel mode**
 - On a system call
 - On a hardware interrupt
 - User program doing illegal work (exception)
- **Actions needed, particularly w.r.t. to hardware interrupts**
 - Change to kernel mode & switch to kernel stack
 - Kernel to work with devices, if needed
 - Kernel to understand interface of device

there are 2 types of kernel stacks, one process wala other general kernel stack

Handling traps

- **Actions needed on a trap**

- **Save the processor's registers (context) for future use**
- **Set up the system to run kernel code (kernel context) on kernel stack**
- **Start kernel in appropriate place (sys call, intr handler, etc)**
- **Kernel to get all info related to event (which block I/O done?, which sys call called, which process did exception and what type, get arguments to system call, etc)**

if a process was executing and using registers, on context switch some registers needs to be saved. then kernel's registers are loaded(kernel's context) kernel should get the info about the interrupt by the hardware

Privilege level

- The x86 has 4 protection levels, numbered 0 (most privilege) to 3 (least privilege).
- In practice, most operating systems use only 2 levels: 0 and 3, which are then called kernel mode and user mode, respectively.
- The current privilege level with which the x86 executes instructions is stored in %cs register, in the field CPL.

Privilege level

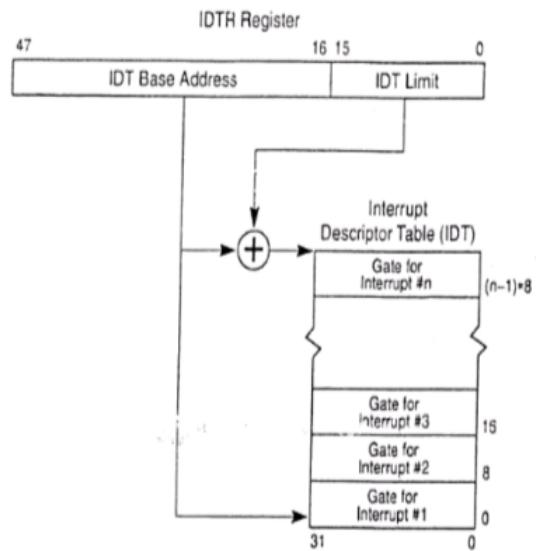
- Changes automatically on
 - “int” instruction
 - hardware interrupt
 - exception
- Changes back on
 - iret
- “int” 10 --> makes 10th hardware interrupt. S/w interrupt can be used to ‘create hardware interrupt’
- Xv6 uses “int 64” for actual system calls

in linux 0x80 is used for syscall

Interrupt Descriptor Table (IDT)

- **IDT defines interrupt handlers**
- **Has 256 entries**
 - each giving the %cs and %eip to be used when handling the corresponding interrupt.
- **Interrupts 0-31 are defined for software exceptions, like divide errors or attempts to access invalid memory addresses.**
- **Xv6 maps the 32 hardware interrupts to the range 32-63**
- **and uses interrupt 64 as the system call interrupt**

IDTR and IDT



entries in the idt table are called gates.

lets see xv6 code

```

136     ushort padding7;
137     ushort fs;
138     ushort padding8;
139     ushort gs;
140     ushort padding9;
141     ushort ldt;
142     ushort padding10;
143     ushort t;           // Trap on task switch
144     ushort iomb;        // I/O map base address
145 };
146
147 // Gate descriptors for interrupts and traps
148 struct gatedesc {
149     uint off_15_0 : 16;    // low 16 bits of offset in segment
150     uint cs : 16;         // code segment selector
151     uint args : 5;        // # args, 0 for interrupt/trap gates
152     uint rsv1 : 3;        // reserved(should be zero I guess)
153     uint type : 4;        // type(STS_{IG32,TG32})
154     uint s : 1;            // must be 0 (system)
155     uint dpl : 2;          // descriptor(meaning new) privilege level
156     uint p : 1;            // Present
157     uint off_31_16 : 16;   // high bits of offset in segment
158 };
159
160 // Set up a normal interrupt/trap gate descriptor.
"mmu.h" 181L, 6571C

```

148,1

this C structure is an entry in the IDT table

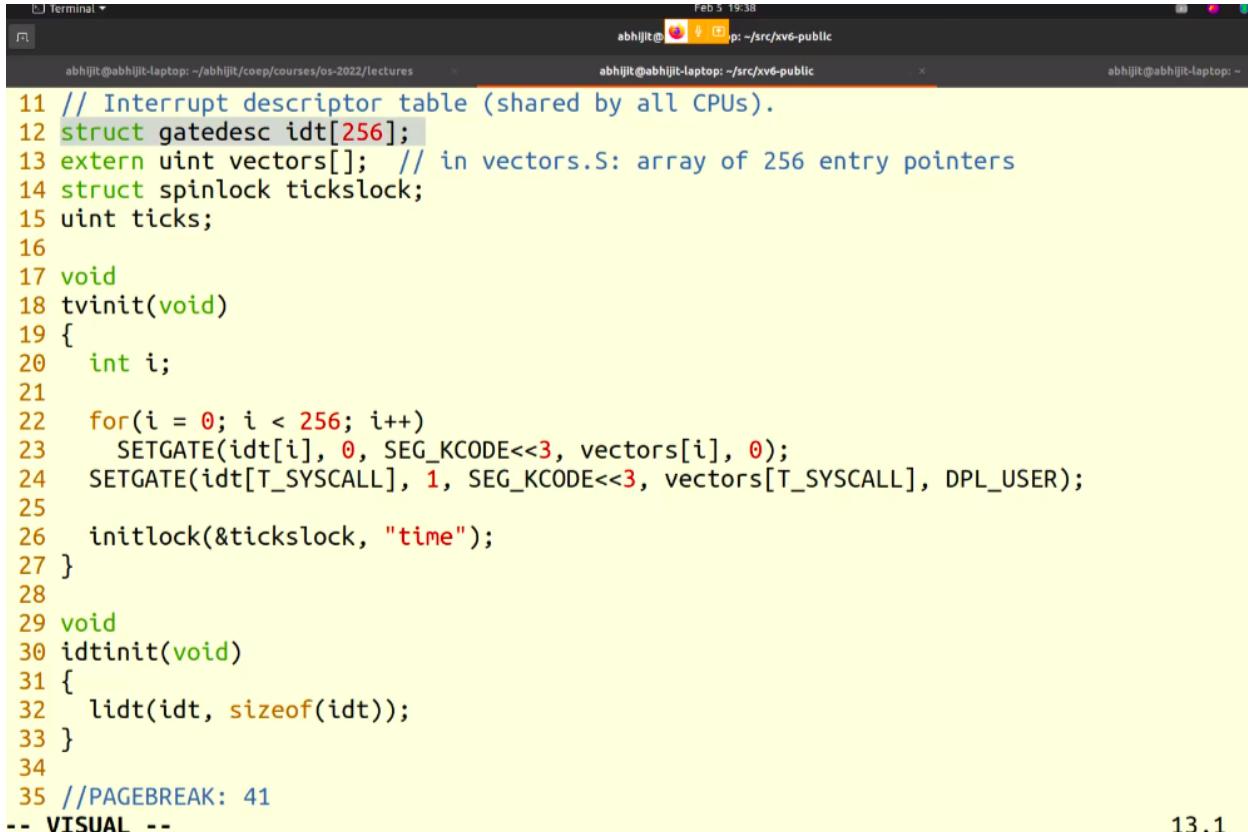
: 16 means field of 16 bits, and so on

Setting IDT entries

```

void
tvinit(void)
{
    int i;
    for(i = 0; i < 256; i++)
        SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);
        SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3,
                vectors[T_SYSCALL], DPL_USER);
        /* value 1 in second argument --> don't disable
        interrupts
           * DPL_USER means that processes can raise
        this interrupt. */
        initlock(&tickslock, "time");
}

```



The screenshot shows a terminal window with three tabs. The current tab displays the xv6 source code for setting IDT entries. The code defines a global variable `idt[256]` and initializes it in the `tvinit` function. It sets up 256 interrupt descriptor table entries, each pointing to a vector in the `vectors` array. The first 255 entries point to the kernel code at `SEG_KCODE<<3` with privilege level 0. The `T_SYSCALL` entry points to the user-space code at `DPL_USER`. The `idtinit` function is also shown, which uses the `lidt` instruction to load the IDT.

```

11 // Interrupt descriptor table (shared by all CPUs).
12 struct gatedesc idt[256];
13 extern uint vectors[]; // in vectors.S: array of 256 entry pointers
14 struct spinlock tickslock;
15 uint ticks;
16
17 void
18 tvinit(void)
19 {
20     int i;
21
22     for(i = 0; i < 256; i++)
23         SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);
24         SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);
25
26     initlock(&tickslock, "time");
27 }
28
29 void
30 idtinit(void)
31 {
32     lidt(idt, sizeof(idt));
33 }
34
35 //PAGEBREAK: 41
-- VISUAL --

```

13.1

line 12 has idt table, which is part of data part of kernel, it gets loaded when kernel was loaded in memory in bootmain

tvinits loads IDT table into memory

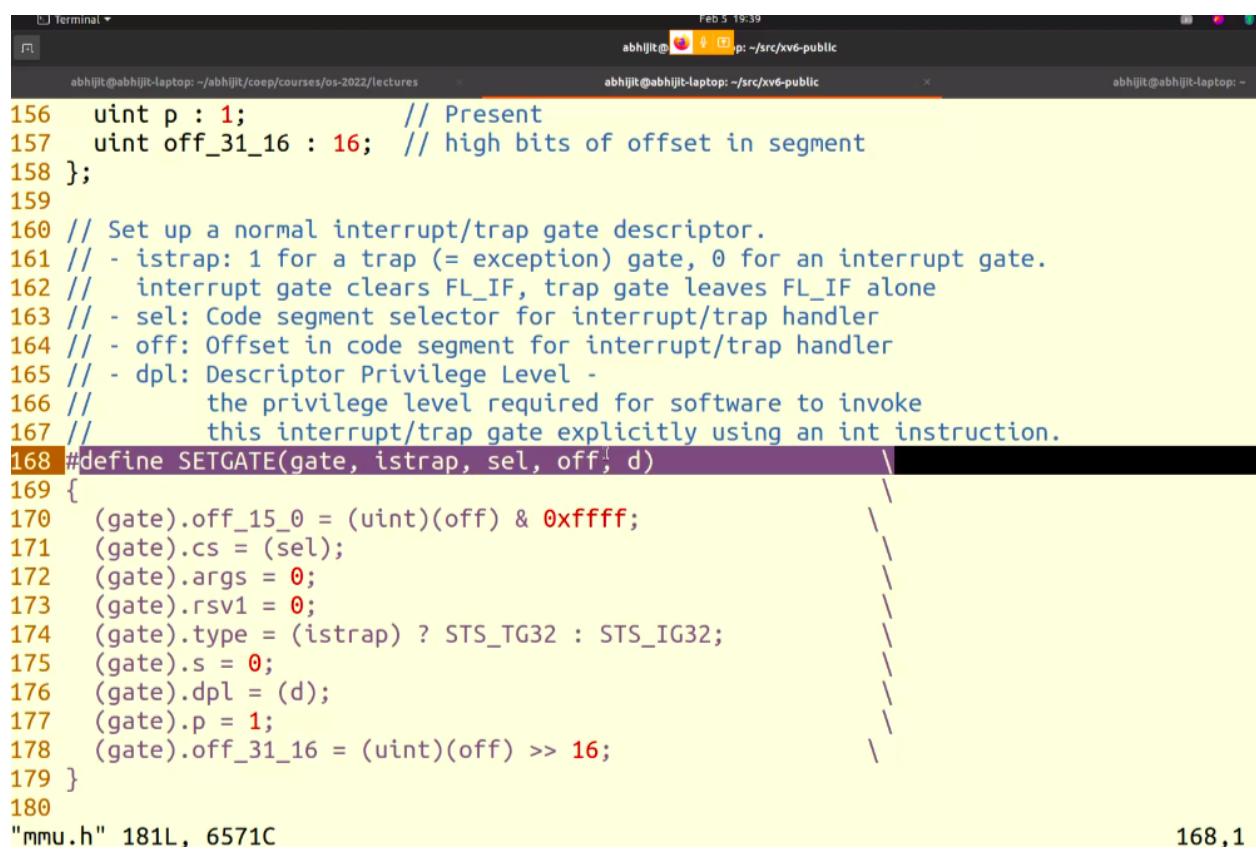
vectors[i] is the address of function that is to be executed, which is externed at line 13, defined in vectors.S

T_SYSCALL is #defined 64

DPL_USR is 0x3

for syscall we are overwriting istrap=1, and d=3 in SETGATE, i.e at line 24

SETGATE



```
156     uint p : 1;           // Present
157     uint off_31_16 : 16;   // high bits of offset in segment
158 };
159
160 // Set up a normal interrupt/trap gate descriptor.
161 // - istrap: 1 for a trap (= exception) gate, 0 for an interrupt gate.
162 //   interrupt gate clears FL_IF, trap gate leaves FL_IF alone
163 // - sel: Code segment selector for interrupt/trap handler
164 // - off: Offset in code segment for interrupt/trap handler
165 // - dpl: Descriptor Privilege Level -
166 //         the privilege level required for software to invoke
167 //         this interrupt/trap gate explicitly using an int instruction.
168 #define SETGATE(gate, istrap, sel, off, d)
169 {
170     (gate).off_15_0 = (uint)(off) & 0xffff;
171     (gate).cs = (sel);
172     (gate).args = 0;
173     (gate).rsv1 = 0;
174     (gate).type = (istrap) ? STS_TG32 : STS_IG32;
175     (gate).s = 0;
176     (gate).dpl = (d);
177     (gate).p = 1;
178     (gate).off_31_16 = (uint)(off) >> 16;
179 }
```

168,1

Vectors.S

```

1 # generated by vectors.pl - do not edit
2 # handlers
3 .globl _alltraps
4 .globl vector0
5 vector0:
6     pushl $0
7     pushl $0
8     jmp _alltraps
9 .globl vector1
10 vector1:
11    pushl $0
12    pushl $1
13    jmp _alltraps
14 .globl vector2
15 vector2:
16    pushl $0
17    pushl $2
18    jmp _alltraps
19 .globl vector3
20 vector3:
21    pushl $0
22    pushl $3
23    jmp _alltraps
24 .globl vector4
25 vector4:

```

"vectors.S" 1537L, 21356C 1,39

```

1277
1278 # vector table
1279 .data
1280 .globl vectors
1281 vectors:
1282     .long vector0
1283     .long vector1
1284     .long vector2
1285     .long vector3
1286     .long vector4
1287     .long vector5
1288     .long vector6
1289     .long vector7
1290     .long vector8
1291     .long vector9
1292     .long vector10
1293     .long vector11
1294     .long vector12
1295     .long vector13
1296     .long vector14
1297     .long vector15
1298     .long vector16
1299     .long vector17
1300     .long vector18
1301     .long vector19

```

-- VISUAL -- 1282,1

this is vectors array

a interrupt first pushes 0, then number of interrupt and then jmp to alltraps

tvinit() is called into main.c, a file which we have already seen.

On int instruction/interrupt the CPU does this:

- Fetch the n'th descriptor from the IDT, where n is the argument of int.
- Check that CPL in %cs is <= DPL, where DPL is the privilege level in the descriptor.
- Save %esp and %ss in CPU-internal registers, but only if the target segment selector's PL < CPL.
 - Switching from user mode to kernel mode. Hence save user code's SS and ESP
- Load %ss and %esp from a task segment descriptor.
 - Stack changes to kernel stack now. TS descriptor is on GDT, index given by TR register. See switchuvvm()

if a syscall is to be made, int 64, 64th IDT entry is loaded in hardware

2nd point means all levels are 0, except syscall which is 3, if application code was running, CPL=3, and DPL=3 so no problem, it is checking that you should go to lower level, why is this so, because process manufacturer always had 4 levels, OS used only 2 levels

if you are in syscall your cpl=3, and for other interrupts dpl=0,hence you cannot be interrupted in syscall?

ss and esp are saved in internal registers,only if you are going to lower level, i.e. transition from user to kernel mode, so you have to use kernel stack, processor gets the

kernel stacker using the task segment descriptor.

On int instruction/interrupt the CPU does this:

- Fetch the n'th descriptor from the IDT, where n is the argument of int.
- Check that CPL in %cs is <= DPL, where DPL is the privilege level in the descriptor.
- Save %esp and %ss in CPU-internal registers, but only if the target segment selector's PL < CPL.
 - Switching from user mode to kernel mode. Hence save user code's SS and ESP
- Load %ss and %esp from a task segment descriptor.
 - Stack changes to kernel stack now.
TS descriptor is on GDT, index given by TR register. See switchuvm()
- Push %ss. // optional
- Push %esp. // optional (also changes ss,esp using TSS)
- Push %eflags.
- Push %cs.
- Push %eip.
- Clear the IF bit in %eflags, but only on an interrupt.
- Set %cs and %eip to the values in the descriptor.

optional push of ss and esp is done on kernel stack, then eflags, cs, eip is pushed mandatory on kernel stack, then we push 0 and 64

clear the IF bit, only on hardware interrupt(interrupt), that is interrupts are disabled now.

After “int” ‘s job is done

- IDT was already set
 - Remember vectors.S
- So jump to 64th entry in vector’s

vector64:

```
pushl $0  
pushl $64  
jmp alltraps
```

- So now stack has ss, esp, eflags, cs, eip, 0 (for error code), 64
- Next run alltraps from trapasm.S

0 and 64 are pushed by our code, ss,esp,eflags cs,eip is loaded in stack by hardware.

0 will be replaced by errno, there is a single function that handles all traps, so we need to push trap number on stack as well

```
1 #include "mmu.h"
2
3 # vectors.S sends all traps here.
4 .globl alltraps
5 alltraps:
6 # Build trap frame.
7 pushl %ds
8 pushl %es
9 pushl %fs
10 pushl %gs
11 pushal
12
13 # Set up data segments.
14 movw $(SEG_KDATA<<3), %ax
15 movw %ax, %ds
16 movw %ax, %es
17
18 # Call trap(tf), where tf=%esp
19 pushl %esp
20 call trap
21 addl $4, %esp
22
23 # Return falls through to trapret...
24 .globl trapret
25 trapret:
-- VISUAL --
```

9,6

```
# Build trap frame.  
pushl %ds  
pushl %es  
pushl %fs  
pushl %gs  
pushal // push all gen purpose  
regs  
# Set up data segments.  
movw $(SEG_KDATA<<3), %ax  
movw %ax, %ds  
movw %ax, %es  
# Call trap(tf), where tf=%esp  
pushl %esp # first arg to trap()  
call trap  
addl $4, %esp
```

alltraps:

- Now stack contains
- ss, esp, eflags, cs, eip, 0 (for error code), 64, ds, es, fs, gs, eax, ecx, edx, ebx, oesp, ebp, esi, edi
 - This is the struct trapframe !
 - So the kernel stack now contains the trapframe
 - Trapframe is a part of kernel stack

trap is called on line 20 on alltraps

struct trapframe is all these values defined in reverse order

```

149 // hardware and by trapasm.S, and passed to trap().
150 struct trapframe {
151     // registers as pushed by pusha
152     uint edi;
153     uint esi;
154     uint ebp;
155     uint oesp;      // useless & ignored
156     uint ebx;
157     uint edx;
158     uint ecx;
159     uint eax;
160
161     // rest of trap frame
162     ushort gs;
163     ushort padding1;
164     ushort fs;
165     ushort padding2;
166     ushort es;
167     ushort padding3;
168     ushort ds;
169     ushort padding4;
170     uint trapno;
171
172     // below here defined by x86 hardware
173     uint err;
-- VISUAL --

```

158,1

before call trap, esp was pushed on stack, which now becomes the argument to trap function

```

void
trap(struct trapframe *tf)
{
    if(tf->trapno == T_SYSCALL){
        if(myproc()->killed)
            exit();
        myproc()->tf = tf;
        syscall();
        if(myproc()->killed)
            exit();
        return;
    }
    switch(tf->trapno){
        ....
}

```

trap()

- Argument is trapframe
- In alltraps
 - Before “call trap”, there was “push %esp” and stack had the trapframe
 - Remember calling convention --> when a function is called, the stack contains the arguments in reverse order (here only 1 arg)

```

36 void
37 trap(struct trapframe *tf)
38 {
39     if(tf->trapno == T_SYSCALL){
40         if(myproc()->killed)
41             exit();
42         myproc()->tf = tf;
43         syscall();
44         if(myproc()->killed)
45             exit();
46         return;
47     }
48
49     switch(tf->trapno){
50     case T_IRQ0 + IRQ_TIMER:
51         if(cpuid() == 0){
52             acquire(&tickslock);
53             ticks++;
54             wakeup(&ticks);
55             release(&tickslock);
56         }
57         lapiceoi();
58         break;
59     case T_IRQ0 + IRQ_IDE:
60         ideintr();
61     }
62 }
63
64 :e#
```

```

121 [SYS_uptime]    sys_uptime,
122 [SYS_open]      sys_open,
123 [SYS_write]     sys_write,
124 [SYS_mknod]     sys_mknod,
125 [SYS_unlink]    sys_unlink,
126 [SYS_link]      sys_link,
127 [SYS_mkdir]     sys_mkdir,
128 [SYS_close]     sys_close,
129 };
130
131 void
132 syscall(void)
133 {
134     int num;
135     struct proc *curproc = myproc();
136
137     num = curproc->tf->eax;
138     if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
139         curproc->tf->eax = syscalls[num]();
140     } else {
141         cprintf("%d %s: unknown sys call %d\n",
142                 curproc->pid, curproc->name, num);
143         curproc->tf->eax = -1;
144     }
145 }
```

-- VISUAL --

138,1

trap()

- Has a switch
 - `switch(tf->trapno)`
 - Q: who set this trapno?
 - Depending on the type of trap
 - Call interrupt handler
-
- Timer
 - `wakeup(&ticks)`
 - IDE: disk interrupt
 - `Ideintr()`
 - KBD
 - `Kbdintr()`
 - COM1
 - `Uatrintr()`
 - If Timer
 - Call `yield()` -- calls `sched()`
 - If process was killed (how is that done?
 - Call `exit()!`

trapno, was pushed in vectors.S for each vector

what happens when trap returns?

trap returns in alltraps

```
# Call trap(tf), where tf=%esp
pushl %esp
call trap
addl $4, %esp

# Return falls through to trapret...
.globl trapret
trapret:
```

VISUAL -- 22,0-1

when trap() returns

- #Back in alltraps
call trap
addl \$4, %esp

```
# Return falls through to trapret...
.globl trapret
trapret:
    popal
    popl %gs
    popl %fs
    popl %es
    popl %ds
    addl $0x8, %esp # trapno and errcode
    iret
```



Stack had (trapframe)

- ss, esp, eflags, cs, eip, 0 (for error code), 64, ds, es, fs, gs, eax, ecx, edx, ebx, oesp, ebp, esi, edi, esp
- add \$4 %esp
 - esp
- popal
 - eax, ecx, edx, ebx, oesp, ebp, esi, edi
- Then gs, fs, es, ds
- add \$0x8, %esp
 - 0 (for error code), 64
- iret
 - ss, esp, eflags, cs, eip,

after trap returns, we add 4 to esp, to pop the pushed esp, (removing stack pointer's value from stack,because it was just use as argument to trap(), now it is not needed)

then we call popal, which will pop all the general purpose registers from the stack

then add 8, will remove error code, and interrupt number

iret is opposite of int instruction so pop that 5 values, and uses ss and esp value,so the stack switches back to application stack,also cs and eip will be overwritten, so process will now resume

NOTE:struct trapframe{} in c file was definition of struct, while in assembly code we created a instance of that struct, we cannot create a instance of this struct in C code, because int instr will make lookup into idt and jump to some location which pushes some value on stack, then we push some other values to build the trapframe.

trapframe is all the essential info needed to handle a trap.

NOTE:the stack used here is per process kernel stack allocated in entry.S

compiler works with only general purpose registers in calling conventions(pushed by pushal)

how a syscall actually works?

in Usys.S, there is a macro which creates an entry for a syscall, and we create a entry for each syscall here.

lets consider fork, for example then

.globl fork;

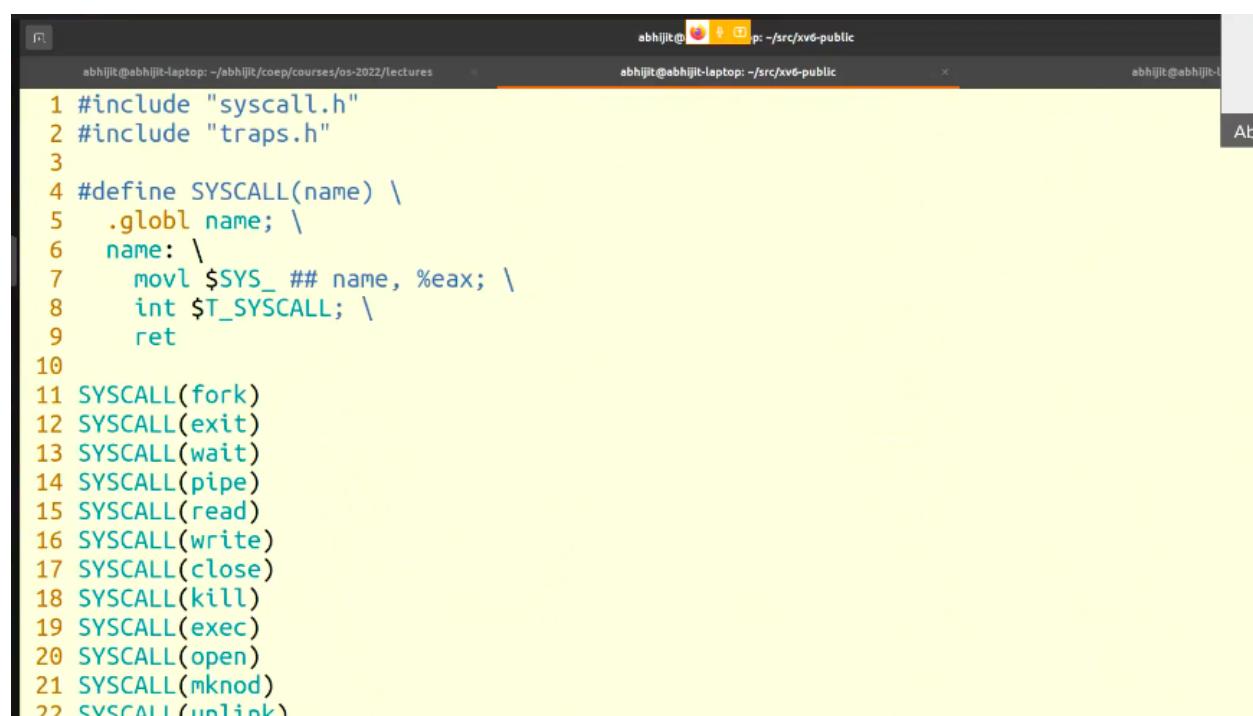
fork:

```
    movl $SYS_fork,eax;
```

```
    int 64;
```

```
    ret
```

such a code is generated for each syscall, here SYS_fork is a macro defined in file, which is basically a number that we push in eax, then we call int 64, which jumps into vectors.S, then push 0 push 64 then jmp to alltraps, push some values then call trap, which checks for syscall then call syscall and then inside the syscall() seen in syscall.c file, we access this eax value through the trapframe, as an index to a array of function pointers to the actual syscall functions which are written in sysfiles.c and sysproc.c



The screenshot shows a terminal window with three tabs. The active tab contains assembly code for system calls. The code defines a macro SYSCALL that takes a name and generates assembly instructions to push the name onto the stack, move it into eax, and then call the interrupt 64 (int 64). Below this, specific entries are provided for fork, exit, wait, pipe, read, write, close, kill, exec, open, mknod, and symlink.

```
1 #include "syscall.h"
2 #include "traps.h"
3
4 #define SYSCALL(name) \
5     .globl name; \
6     name: \
7         movl $SYS_## name, %eax; \
8         int $T_SYSCALL; \
9         ret
10
11 SYSCALL(fork)
12 SYSCALL(exit)
13 SYSCALL(wait)
14 SYSCALL(pipe)
15 SYSCALL(read)
16 SYSCALL(write)
17 SYSCALL(close)
18 SYSCALL(kill)
19 SYSCALL(exec)
20 SYSCALL(open)
21 SYSCALL(mknod)
22 SYSCALL(symlink)
```

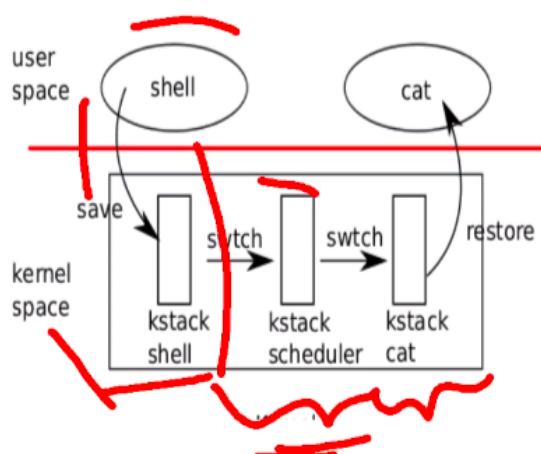
to sum up, tvinit() in main creates an idt table of type gatedesc of 256 entries, which are set using SETGATE macro, that maps each idt with a vector array containing the function to be called on a particular interrupt, this vector array is defined in vectors.S, which will push 0, push trapNumber, then jmp to alltraps, inside alltraps, we push other necessary values to generate the trapframe, which is used as an argument to the trap function, which has a switch for trapnumber, if it is a syscall then syscall function is called which checks the value in eax register(defined in usys.s) of trapframe, that value is mapped to array of function pointers that point to a function that needs to be executed for the syscall

Scheduler

Steps in scheduling

- Suppose you want to switch from P1 to P2 on a timer interrupt
 - P1 was doing
`F() { i++; j++; }`
 - P2 was doing
`G() { x--; y++; }`
 - P1 will experience a timer interrupt, switch to kernel (scheduler) and scheduler will schedule P2

4 stacks need to change!



- **User stack of process -> kernel stack of process**
 - Switch to kernel stack
 - The normal sequence on any interrupt !
- **Kernel stack of process -> kernel stack of scheduler**
 - Why?
- **Kernel stack of scheduler -> kernel stack of new process . Why?**
- **Kernel stack of new process -> user stack of new process**

when shell is running and their is timer interrupt, context of shell is need to be saved, it is saved in kernel stack of shell, scheduler is to be run after that, the context of shell is currently stored on top of kstack, scheduler needs lot of functions, so needs a stack of its own, so a different stack is used for scheduler, before scheduler starts it process, kernel stack is changed to scheduler stack, then now when scheduler wants to pass control to cat, then again stack needs to be changed to context of kernel stack of cat process. So first it was user stack of shell, then when timer interrupt occurred, context is stored on kernel stack of shell.,then kstack of scheduler then kstack of cat

inside main.c code of xv6

```

terminal - Feb 7 19:43 abhijit: ~src/xv6-public
19 {
20     kinit1(end, P2V(4*1024*1024)); // phys page allocator
21     kvmalloc(); // kernel page table
22     mpinit(); // detect other processors
23     lapicinit(); // interrupt controller
24     seginit(); // segment descriptors
25     picinit(); // disable pic
26     ioapicinit(); // another interrupt controller
27     consoleinit(); // console hardware
28     uartinit(); // serial port
29     pinit(); // process table
30     tvinit(); // trap vectors
31     binit(); // buffer cache
32     fileinit(); // file table
33     ideinit(); // disk
34     startothers(); // start other processors
35     // Abhijit: The full fledged page table has been mapped
36     // on all processors. Hence now reclaim all physical memory
37     // into free list from 4+2048 MB -> 224 + 2048 MB VM
38     // i.e. in kernel-data+memory region as per kmap[]
39     kinit2(P2V(4*1024*1024), P2V(PHYSTOP)); // must come after startothers()
40     userinit(); // first user process
41     mpmain(); // finish this processor's setup
42 }
43
44 // Other CPUs jump here from entryother.S.

```

40,4

function called mpinit was called which actually detected multiple processors and initialize them. the mpmain will start all the processor's execution, it will make all the processor run, in symmetric multiprocessing system. mpmain calls scheduler.

```
terminal - Viewing Abhijit's applica... 1e
Feb 7 19:44
45 static void
46 mpenter(void)
47 {
48     switchkvm();
49     seginit();
50     lapicinit();
51     mpmain();
52 }
53
54 // Common CPU setup code.
55 static void
56 mpmain(void)
57 {
58     cprintf("cpu%d: starting %d\n", cpuid(), cpuid());
59     idtinit();          // load idt register
60     xchg(&(mycpu()->started), 1); // tell startothers() we're up
61     scheduler();        // start running processes
62 }
63
64 pde_t entrypgdir[]; // For entry.S
65
66 // Start the non-boot (AP) processors.
67 static void
68 startothers(void)
69 {
70     extern uchar _binary_entryother_start[], _binary_entryother_size[];
61,23
```

```
terminal - Viewing Abhijit's applica... 1e
Feb 7 19:45
24 // Scheduler never returns. It loops, doing:
25 // - choose a process to run
26 // - swtch to start running that process
27 // - eventually that process transfers control
28 // via swtch back to the scheduler.
29 void
30 scheduler(void)
31 {
32     struct proc *p;
33     struct cpu *c = mycpu();
34     c->proc = 0;
35
36     for(;;){
37         // Enable interrupts on this processor.
38         /* Abhijit: the enabling of interrupts
39          * in the outer infinite loop is to ensure that
40          * if in the inner loop, no runnable process is found,
41          * which means all are waiting for I/O, then
42          * the interrupts can be received
43          */
44         sti();
45
46         // Loop over process table looking for process to run.
47         acquire(&ptable.lock);
48         for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
49             if(p->state != RUNNABLE)
```

349,1

it should array for proc,(a global variable ptable contains array of this proc) iterate over those processes that are ready and then allocate them.

this C code will be compiled according to calling conventions

sti() is using the gcc directive asm to embed assembly code in C code, to enable interrupts

```
terminal - Feb 7 19:47
1 abhijit@abhijit-OptiPlex-5070: ~/src/xv6-public
30 }
31
32 /* Abhijit: Clear Interrupt Flag */
33 static inline void
34 cli(void)
35 {
36     asm volatile("cli");
37 }
38
39 /* Abhijit: Set Interrupt Flag */
40 static inline void
41 sti(void)
42 {
43     asm volatile("sti");
44 }
45
46 /* Abhijit: Atomic Compare and Swap */
47 static inline uint
48 xchg(volatile uint *addr, uint newval)
49 {
50     uint result;
51
52     // The + in "+m" denotes a read-modify-write operand.
53     asm volatile("lock; xchgl %0, %1" :
54         "+m" (*addr), "=a" (result) :
55         "1" (newval) :
```

145,0-

```

terminal - Feb 7 19:48 abhijit: ~src/xv6-public
1
40      * if in the inner loop, no runnable process is found,
41      * which means all are waiting for I/O, then
42      * the interrupts can be received
43      */
44      sti();
45
46      // Loop over process table looking for process to run.
47      acquire(&ptable.lock);
48      for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
49          if(p->state != RUNNABLE)
50              continue;
51
52          // Switch to chosen process. It is the process's job
53          // to release ptable.lock and then reacquire it
54          // before jumping back to us.
55          c->proc = p;
56          switchuvm(p);
57          p->state = RUNNING;
58
59          swtch(&(c->scheduler), p->context);
60          /* Abhijit: The swtch() from sched() returns here
61             * Then switches to kvm below, goes in loop again
62             * finds a process to run and using the above switch
63             * statement switches to that process.
64             */
65          switchkvm();

```

347,7

RUNNABLE==READY, if process not ready skip, if RUNNABLE, it initializes $c \rightarrow proc = p$; (storing pointer to currently running process)

which was a cpu pointer.we are passing address to swtch() of $c \rightarrow scheduler$, since we are likely to change it. scheduler is instance of type struct context only

then calls switchuvm, changes state, and then calls swtch() with a argument $p \rightarrow context$, p is the process to be scheduled

```
terminal - Feb 7 19:50 Viewing Abhijit's applica... 1e
22 extern int ncpu;
23
24 //PAGEBREAK: 17
25 // Saved registers for kernel context switches.
26 // Don't need to save all the segment registers (%cs, etc),
27 // because they are constant across kernel contexts.
28 // Don't need to save %eax, %ecx, %edx, because the
29 // x86 convention is that the caller has saved them.
30 // Contexts are stored at the bottom of the stack they
31 // describe; the stack pointer is the address of the context.
32 // The layout of the context matches the layout of the stack in swtch.S
33 // at the "Switch stacks" comment. Switch doesn't save eip explicitly,
34 // but it is on the stack and allocproc() manipulates it.
35 struct context {
36     uint edi;
37     uint esi;
38     uint ebx;
39     uint ebp;
40     uint eip;■
41 };
42
43 enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };
44
45 // Per-process state
46 struct proc {
47     uint sz;                                // Size of process memory (bytes)
- VISUAL --
```

40,12

what is c? c is struct cpu

```
terminal - Feb 7 19:51 Viewing Abhijit's applica...
1 // Per-CPU state
2 struct cpu {
3     // Abhijit: apicid initialized in mpinit()
4     uchar apicid;           // Local APIC ID
5     // Abhijit:
6     struct context *scheduler; // swtch() here to enter scheduler
7     // Abhijit: ts is changed in switchuvm()
8     struct taskstate ts;      // Used by x86 to find stack for interrupt
9     // Abhijit: gdt initialized in seginit(), startothers->mpenter->seginit
10    struct segdesc gdt[NSEGS]; // x86 global descriptor table
11    // Abhijit: started initialized in mpmain()
12    volatile uint started;    // Has the CPU started?
13    // Abhijit: this is incr in every spinlock
14    int ncli;                // Depth of pushcli nesting.
15    // Abhijit: used to remember whether interrupts were enabled
16    int intena;              // Were interrupts enabled before pushcli?
17    // Abhijit: this gets assigned in every scheduler() call
18    struct proc *proc;        // The process running on this cpu or null
19 };
20
21 extern struct cpu cpus[NCPU];
22 extern int ncpu;
23
24 //PAGEBREAK: 17
25 // Saved registers for kernel context switches.
26 // Don't need to save all the segment registers (%cs, etc),
proc.h" 66L, 2660C
```

14,12

it has a pointer scheduler of context struct, lets see mycpu()

```

terminal - Feb 7 19:54
abhijit@: ~src/xv6-public

32     return mycpu()-cpus;
33 }
34
35 // Must be called with interrupts disabled to avoid the caller being
36 // rescheduled between reading lapicid and running through the loop.
37 struct cpu*
38 mycpu(void)
39 {
40     int apicid, i;
41
42     if(readeflags()&FL_IF) //Abhijit: confirm the assertion in comment above
43         panic("mycpu called with interrupts enabled\n");
44
45     apicid = lapicid();
46     // APIC IDs are not guaranteed to be contiguous. Maybe we should have
47     // a reverse map, or reserve a register to store &cpus[i].
48     for (i = 0; i < ncpu; ++i) {
49         if (cpus[i].apicid == apicid)
50             return &cpus[i];
51     }
52     panic("unknown apicid\n");
53 }
54
55 // Disable interrupts so that we are not rescheduled
56 // while reading proc from the cpu structure
57 struct proc*

```

42,10

it is assumed here, whichever process is to be scheduled its context is already present, while calling swtch

lets jump to switch which is defined into Switch.S

```

1 # Context switch
2 #
3 #   void swtch(struct context **old, struct context *new);
4 #
5 # Save the current registers on the stack, creating
6 # a struct context, and save its address in *old.
7 # Switch stacks to new and pop previously-saved registers.
8
9 # Abhijit: remember you dont' change anything in the contexts,
10 # i.e. **old is not changed
11 # old *old is changed and *new is changed
12 # swapping of the contexts, not what is within the contexts
13
14 .globl swtch
15 swtch:
16 #Abhijit: swtch was called through a function call.
17 #So %eip was saved on stack already
18 movl 4(%esp), %eax    # Abhijit: eax = old
19 movl 8(%esp), %edx    # Abhijit: edx = new
20
21 # Save old callee-saved registers
22 pushl %ebp
23 pushl %ebx
24 pushl %esi
25 pushl %edi      # Abhijit: esp = esp + 16
26 # old stack contains 4 new registers now

```

3,53

first argument is old(pointer to pointer of c → scheduler), 2nd is new(pointer to process context that is to be scheduled)

at top of stack %esp we have return value of function, at 4(esp) we have old(1st argument) and at 8(esp) we have 2nd argument new

ebp,ebx,esi,edi are the callee saved registers, which are pushed to stack, which stack? scheduler stack, in assembly .comm wala stack

```

12 # swapping of the contexts, not what is within the contexts
13
14 .globl swtch
15 swtch:
16     #Abhijit: swtch was called through a function call.
17     #So %eip was saved on stack already
18     movl 4(%esp), %eax    # Abhijit: eax = old
19     movl 8(%esp), %edx    # Abhijit: edx = new
20
21     # Save old callee-saved registers
22     pushl %ebp
23     pushl %ebx
24     pushl %esi
25     pushl %edi          # Abhijit: esp = esp + 16
26     # old stack contains 4 new registers now
27
28     # Switch stacks
29     movl %esp, (%eax) # Abhijit: *old = updated old stack
30     movl %edx, %esp    # Abhijit: esp = new
31
32     # Load new callee-saved registers
33     popl %edi
34     popl %esi
35     popl %ebx
36     popl %ebp          # Abhijit: newesp = newesp - 16, context restored
37     ret                 # Abhijit: will pop from esp now -> function where to return
-- VISUAL --

```

then line 29 does *old= esp's current value after pushing registers, then we now point esp to new stack i.e. p → context 's value, so now the stack is pointing to the kernel stack of new process, then we do pop i.e loading context of new process from stack to actual registers, then we call ret, which will pop value into eip, when this return happens it jumps into new process, we are pushing edi,esi,ebx and ebp explicitly while eip is being pushed and popped implicitly.(hence it aligns with defination of struct context)

the ret will pop eip of new process, so we will go into new process. So switching of context is not like function call, we never return to swtch() after its call

to know how is line 30 swtching stacks, see diagram of struct proc.

NOTE:SCHEDULER ENABLES INTERRUPTS, not disable (sti instruction)

scheduler()

- Disable interrupts
- Find a **RUNNABLE** process. Simple round-robin!
- **c->proc = p**
- **switchuvm(p) : Save TSS of scheduler's stack and make CR3 to point to new process pagedir**
- **p->state = RUNNING**
- **swtch(&(c->scheduler), p->context)**

scheduler()

- `swtch(&(c->scheduler), p->context)`
- Note that when scheduler() was called, when P1 was running
- After call to swtch() shown above
 - The call does NOT return!
 - The new process P2 given by 'p' starts running !
 - Let's review swtch() again

swtch(old, new)

- The magic function in swtch.S
 - Saves callee-save registers of old context
 - Switches esp to new-context's stack
 - Pop callee-save registers from new context
- ~~ret~~
- where? in the case of first process – returns to forkret() because stack was setup like that !
 - in case of other processes, return where?
 - Return address given on kernel stack. But what's that?
 - The EIP in p->context
 - When was EIP set in p->context ?

so the ret will return to eip in p → context, when was eip set in p → context, so to find that we try to look where was scheduler called, it is only called once in mpmain, but interestingly there is another sched() which is also a scheduler

scheduler()

- **Called from?**
 - `mpmain()`
 - **No where else!**
- **`sched()` is another scheduler function !**
 - **Who calls `sched()` ?**
 - `exit()` - a process exiting calls `sched()`
 - `yield()` - a process interrupted by timer calls `yield()`
 - `sleep()` - a process going to wait calls `sleep()`

we try to find where swtch was called, and find that it was called inside sched and that the arguments are reversed, just see 4th and 5th entry in the below image

```
ties Terminal Feb 7 20:13
abhijit:~/src/xv6-public
```

C symbol: swtch

File	Function	Line
0	swtch.S <global>	2 ##void swtch(struct context **old, struct context *new)
1	swtch.S <global>	9 .globl swtch
2	swtch.S <global>	10 swtch:
3	defs.h scheduler	125 void swtch(struct context**, struct context*);
4	proc.c scheduler	346 swtch(&(c->scheduler), p->context);
5	proc.c sched	380 swtch(&p->context, mycpu()->scheduler);

Find this C symbol:
Find this global definition:
Find functions called by this function:
Find functions calling this function:
Find this text string:
Change this text string:
Find this regex pattern:

exit() calls sched, which is obvious, since when process ends, it has to pass control back to kernel

sleep() when is sleep called, when it is above to be blocking for i/o operations

so our interest lies in yield, lets see who calls yield(). it is called in trap.c

```
92     }
93     // In user space, assume process misbehaved.
94     cprintf("pid %d %s: trap %d err %d on cpu %d "
95             "eip 0x%llx addr 0x%llx--kill proc\n",
96             myproc()->pid, myproc()->name, tf->trapno,
97             tf->err, cpuid(), tf->eip, rcr2());
98     myproc()->killed = 1;
99 }
100
101 // Force process exit if it has been killed and is in user space.
102 // (If it is still executing in the kernel, let it keep running
103 // until it gets to the regular system call return.)
104 if(myproc() && myproc()->killed && (tf->cs&3) == DPL_USER)
105     exit();
106
107 /* Abhijit: On a timer interrupt, call scheduler */
108 // Force process to give up CPU on clock tick.
109 // If interrupts were on while locks held, would need to check nlock.
110 if(myproc() && myproc()->state == RUNNING &&
111     tf->trapno == T_IRQ0+IRQ_TIMER)
112     yield(); // Abhijit: yield calls sched() switching to scheduler context
113
114 // Check if the process has been killed since we yielded
115 if(myproc() && myproc()->killed && (tf->cs&3) == DPL_USER)
116     exit();
117 }
-- VISUAL --
```

```
16:12:2017 abhijit@abhijit-OptiPlex-5070: ~/src/xv6-public
```

```
46 // APIC IDs are not guaranteed to be contiguous. Maybe we should have
47 // a reverse map, or reserve a register to store &cpus[i].
48 for (i = 0; i < ncpu; ++i) {
49     if (cpus[i].apicid == apicid)
50         return &cpus[i];
51 }
52 panic("unknown apicid\n");
53 }
54
55 // Disable interrupts so that we are not rescheduled
56 // while reading proc from the cpu structure
57 struct proc*
58 myproc(void) {
59     struct cpu *c;
60     struct proc *p;
61     pushcli();
62     /* Abhijit mycpu() must be called with interrupts disabled.*/
63     c = mycpu();
64     p = c->proc;
65     popcli();
66     return p;
67 }
68
69 //PAGEBREAK: 32
70 // Look in the process table for an UNUSED proc.
71 // If found, change state to EMBRYO and initialize
-- VISUAL --
```

65,:

(myproc is shown above,)if current proc, and its state is RUNNING, and timer interrupt occurred, it calls yield(which means to give up), (timer interrupt is a kind of hardware interrupt hence stack changes to kernel stack as a part of int instruction)

```

415 /* ?? Abhijit: return to whoever called sched() i.e.
416 * yield() or sleep()
417 */
418 }
419
420 // Give up the CPU for one scheduling round.
421 void
422 yield(void)
423 {
424     acquire(&ptable.lock); //DOC: yieldlock
425     myproc()->state = RUNNABLE;
426     sched();
427     /* Abhijit: you return here when scheduler()
428      * switches context using swtch(), and then
429      * sched() runs mycpu()->intena = intena and
430      * sched() returns.
431     */
432     release(&ptable.lock);
433 }
434
435 // A fork child's very first scheduling by scheduler()
436 // will swtch here. "Return" to user space.
437 /* Abhijit : This function exists to release the
438 * ptable.lock and to run initialization functions
439 * Else, the first process could return in trapret directly
440 */
-- VISUAL --

```

426

it marks current process's state as RUNNABLE from RUNNING (marked in scheduler) then calls sched, which calls swtch(&p → context, mycpu() → scheduler), currently running process context is pushed on its own kernel stack, when we came to kernel stack, we came to kernel stack of process when int instruction was executed, it will set p → context pointer to top of stack, after pushing 5 variables, and it will now load context from mycpu → scheduler(), from there it will load context of kernel, this is the same context that was saved when switch was called from scheduler(), the end result of calling this swtch() is that function now returns into the proc.c file scheduler() function, after swtch(), i.e at line 359 (seen 59 in below image)

```
terminal - Feb 7 19:48 abhijit@: ~src/xv6-public
1
40     * if in the inner loop, no runnable process is found,
41     * which means all are waiting for I/O, then
42     * the interrupts can be received
43     */
44     sti();
45
46     // Loop over process table looking for process to run.
47     acquire(&ptable.lock);
48     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
49         if(p->state != RUNNABLE)
50             continue;
51
52         // Switch to chosen process. It is the process's job
53         // to release ptable.lock and then reacquire it
54         // before jumping back to us.
55         c->proc = p;
56         switchuvm(p);
57         p->state = RUNNING;
58
59         swtch(&(c->scheduler), p->context);
60         /* Abhijit: The swtch() from sched() returns here
61          * Then switches to kvm below, goes in loop again
62          * finds a process to run and using the above switch
63          * statement switches to that process.
64          */
65         switchkvm();
```

347,7

now scheduler function will continue in the infinite loop.

sched()

Abhijit

```
void
sched(void)
{
    int intena;
    struct proc *p = myproc();
    if(!holding(&ptable.lock))
        panic("sched ptable.lock");
    if(mycpu()>ncli != 1)
        panic("sched locks");
    if(p->state == RUNNING)
        panic("sched running");
    if(readeflags()&FL_IF)
        panic("sched interruptible");
    intena = mycpu()>intena;
    swtch(&p->context, mycpu()-
>scheduler);
/*A*/ mycpu()>intena = intena;
}
```

- get current process
- Error checking code (ignore as of now)
- get interrupt enabled status on current CPU (ignore as of now)
- call to swtch
 - Note the arguments' order
 - p->context first, mycpu()->scheduler second
- swtch() is a function call
 - pushes address of /*A*/ on stack of current process p
 - switches stack to mycpu()->scheduler. Then pops EIP from that stack and jumps there.
 - when was mycpu()->scheduler set? Ans: during scheduler()!

sched() and scheduler()

```
 sched() {
```

```
...
```

```
    swtch(&p->context, mycpu()->scheduler); /* X */
```

```
}
```

```
 scheduler(void) {
```

```
...
```

```
    swtch(&(c->scheduler), p->context); /* Y */
```

```
}
```

- scheduler() saves context in c->scheduler, sched() saves context in p->context
- after swtch() call in sched(), the control jumps to Y in scheduler
 - Switch from process stack to scheduler's stack
- after swtch() call in scheduler(), the control jumps to X in sched()
 - Switch from scheduler's stack to new process's stack
- Set of co-operating functions

sched() and scheduler() as co-routines

Abhijit

- In sched()
`swtch(&p->context, mycpu()->scheduler);`
- In scheduler()
`swtch(&(c->scheduler), p->context);`
- These two keep switching between processes
- These two functions work together to achieve scheduling
- Using asynchronous jumps
- Hence they are co-routines

there are other 2 possibilities, that is when sched is called from exit() and sleep()

the above discussion only makes sense if the parameters are passed through stack, in calling conventions (and registers are not used for passing parameters) while calling swtch in sched and scheduler, hence we give an option while compiling the code that it will always pass parameters through stack.

why each process has its own kernel stack?

because each process makes its own syscall, so context of each process is to be stored on kernel stack, it is not accessible to application code, or the application could manipulate it and do illegal things.

how does kernel keeps track of each process's kstack?

there is a pointer inside struct proc for that

SUMMARY (combining traps and scheduler)

when timer interrupt, [as a part of int instruction process's user stack changes to process's kernel stack] occurs, it gets cs and eip from idt table inside gatedesc struct, from there it goes to vectors.S where it pushes 0, trap number, then alltraps which

creates trapframe and calls trap, which checks for timer interrupt and calls yield, which calls sched which switches user process's stack to scheduler stack,(inside swtch() i.e. in switch.S, that saves the old context and switches stack, loads context into registers[here process's kernel stack changes to scheduler's stack]) which returns inside scheduler(), inside that infinite loop, which checks for next RUNNABLE process and then switches scheduler's stack to new process's stack,[scheduler's stack changes to new process's kernel stack](which inside swtch.S stores scheduler's context on stack, updates esp, changes esp to process's stack, load context into registers) and return sched() and returns in yield inside trap, which returns in alltraps. In alltraps, which calls trapret, iret [that will change new process's kernel stack to process's user stack], process continues execution, then again a timer interrupt occurs and so on.