

Getting started with Linux Internals

Boot Sequence
OS Initialization
Building OS

xv6 to linux

xv6

xv6 - Intro, building & adding new commands

Basic feel of x86 assembly
building simple "helloworld" boot loader

Familiarity with xv6 codebase & syscall design-
Adding system calls - init 0x64 - to kernel mode
OS sys call design ring 0 ring 3 separation
Understanding boot sequence and initialization
Memory Mgt, pProcess Mgt

Understanding boot sequence and initialization

PC Linux & ARM 64 Linux

Building minimal linux system -kernel and busybox

Understanding & debugging Linux Boot Sequence (OS Initialization)

Adding System Call To The Linux Kernel (5.11.9) In Ubuntu (20.04 LTS)

Linux Embedded, basic components -kernel, root filesystem
-Buildroot build system to build ARM linux

x86 Evolution (@21 mins)

PC Hardware & Its Working

Operating Systems Lectures

How intel processors have evolved from 16 bit 8088/8086 to 64 bit processors

64 bit intel processors are backward compatible - Implies can run in 16 bit or 32 bit mode

X86 Assembly

Addressing modes

Immediate mode -

```
mov $0x5, dst /* $val is constant
```

Register mode -

```
mov %eax, %edx /* %R is register
```

Direct mode

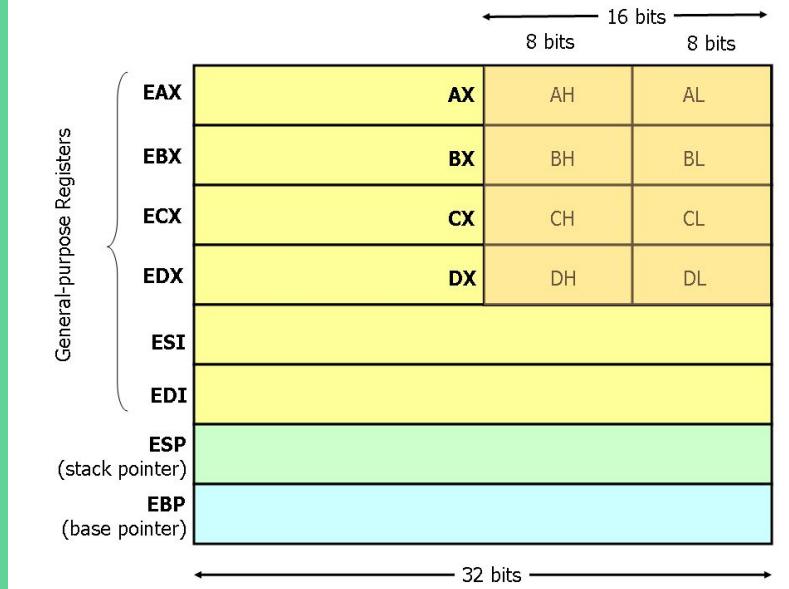
```
mov 0x4033d0, dst /* Oxaddr source  
read from Mem[0xaddr]
```

Indirect mode

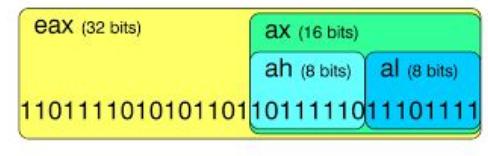
```
mov (%eax), dst /* (%R) R is register source read from Memory [%R]
```

Indirect displacement

```
mov 8(%eax), dst /* D(%R) R is register D is displacement source read from  
Mem[%R + D]
```



Register aliasing / sub-registers



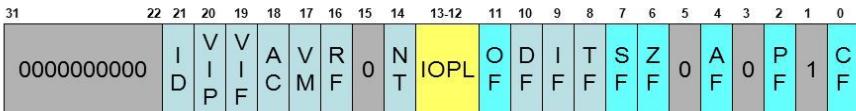
X86 Instructions

- Data movement- mov, push, pop
 - Instruction suffixes b byte w word (2 bytes) l long q quad /doubleword (4 bytes)- size of operand e.g.

```
movl $2, %ebx /* Move 32-bit integer rep of 2 into the 4 bytes to EBX. */  
movw %ax %bx or mov %ax %bx are same (suffix is inferable from operands)  
In some cases explicit suffix is must - movw $123 0x123 is diff from movl $123 0x123
```
- Arithmetic - add, sub, inc, dec, imul (integer multiply)
 - ```
sub $216, %eax - subtract 216 from the value stored in EAX
```
- Bitwise logical and, or, xor
  - ```
and $0x0f, %eax - clear all but the last 4 bits of EAX
```
- Arithmetic - add, sub - effects the flags p.s. Next slide
 - <https://flint.cs.yale.edu/cs421/papers/x86-asm/asm.html>

EFlags register stores - bits from last arithmetic operation. For example, one bit of this word indicates if the last result was zero. Another indicates if the last result was negative. Based on these condition codes, a number of conditional jumps can be performed. For example, the `jz` instruction performs a jump to the specified operand label if the result of the last arithmetic operation was zero. Otherwise, control proceeds to the next instruction in sequence.

The x86 EFLAGS register



Legend:

IOPL = I/O Privilege-Level (0,1,2,3)

AC = Alignment-Check (1=yes, 0=no)

NT = Nested-Task (1=yes, 0=no)

RF = Resume Flag (1=yes, 0=no)

VM = Virtual-8086 Mode (1=yes, 0=no)

VIF = 'Virtual' Interrupt-Flag VIP = 'Virtual' Interrupt is Pending

ID = the CPUID-instruction is implemented if this bit can be 'toggled'

ZF = Zero Flag

SF = Sign Flag

CF = Carry Flag

PF = Parity Flag

OF = Overflow Flag

AF = Auxiliary Flag

Jump/conditional jump

jmp label

jump to label (unconditional)

je label

jump equal ZF=1

jne label

jump not equal ZF=0

js label

jump negative SF=1

jns label

jump not negative SF=0

jg label

jump > (signed) ZF=0 and

SF=OF

jump >= (signed) SF=OF

jl label

jump < (signed) SF!=OF

jle label

jump <= (signed) ZF=1 or

SF!=OF

jump > (unsigned) CF=0 and

ja label

jump >= (unsigned) CF=0

ZF=0

jump < (unsigned) CF=1

jae label

jump <= (unsigned) CF=1 or

jb label

jump >= (unsigned) CF=1

jbe label

if result of last instruction is 0

ZF=1

jz label

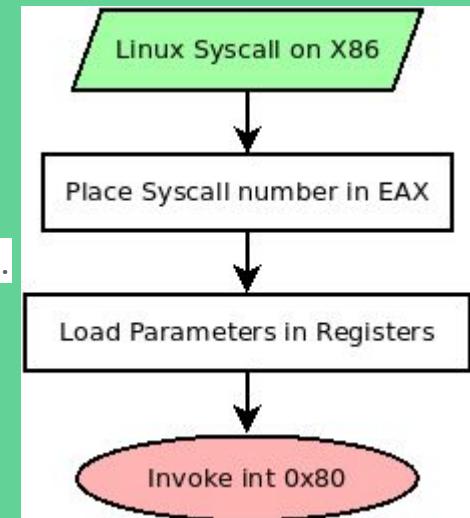
X86 Control Flow & System instructions

Control Flow - jmp, je, jl...

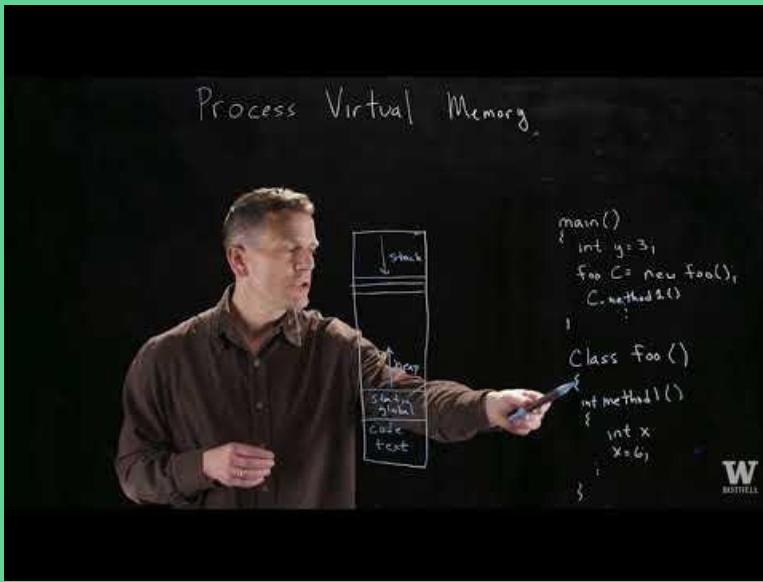
test - no change in operands - just dipes test

System - INT 0x64 - emulates software interrupt,
iret

AT&T x86 syntax (since that is what GDB as well as xv6 & unix code uses).
This means that the destination register comes last

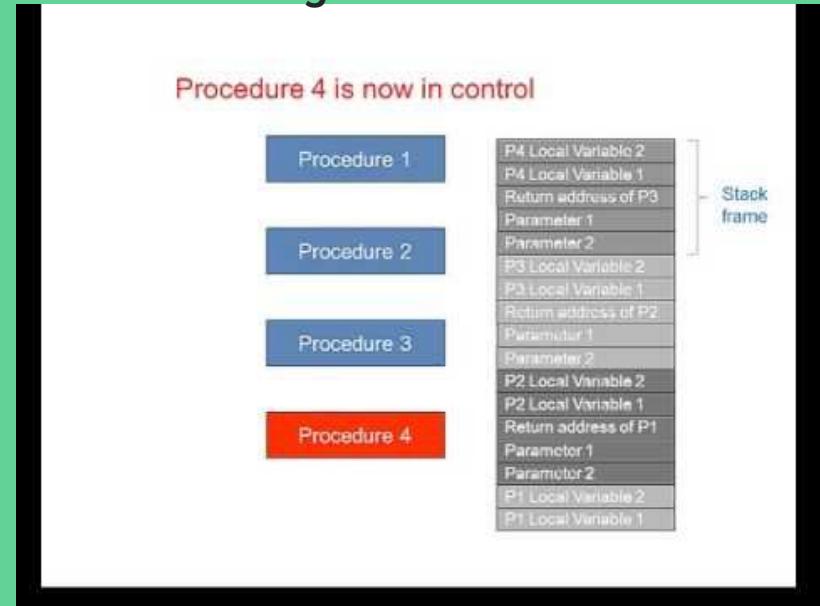


Process VM layout



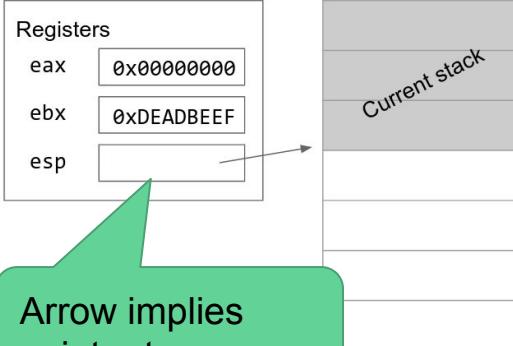
Process virtual addr. space is made up of 4 components-
a) code/text b) static/global vars c) heap d) stack- last 2
sections grows & shrinks as process runs
C++ program how it is mapped to diff segments-
specifically global vars, object allocations and local
variables

How call stack grows & shrinks on func call



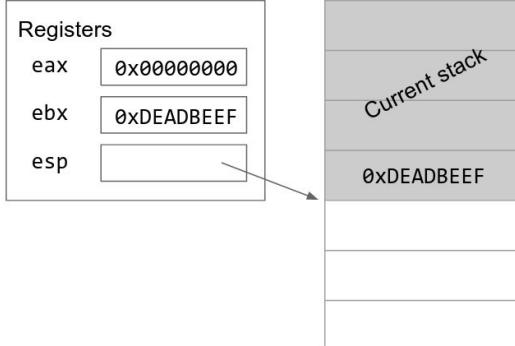
How stack frame is built when function is call & unwinds when we return from a function

X86 Stack operations - push pop



Arrow implies
pointer to mem

Before push %ebx



After push %ebx

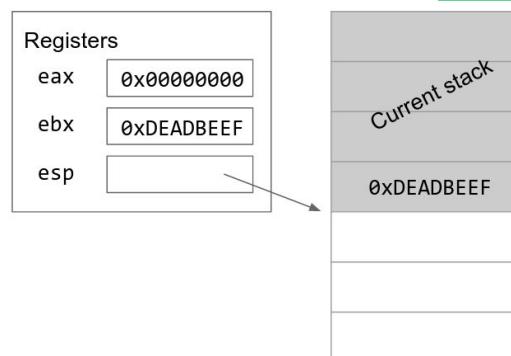
2. x86 Assembly and Call Stack - Computer Security

[https://gabrieletolomei.wordpress.com/misCELLANEA/operating-systems/in-memory-layout/](https://gabrieletolomei.wordpress.com/miscellanea/operating-systems/in-memory-layout/)

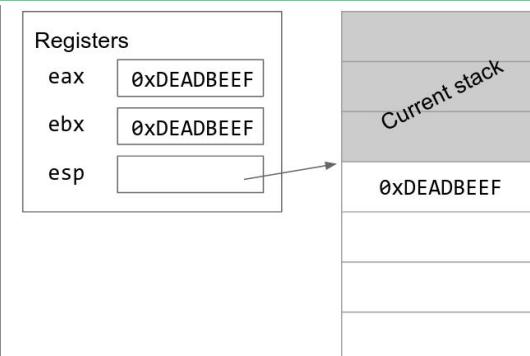
Pushing to stack

- allocate space on the stack by decrementing the esp
- store the value in the newly allocated space

"push" does both of these steps



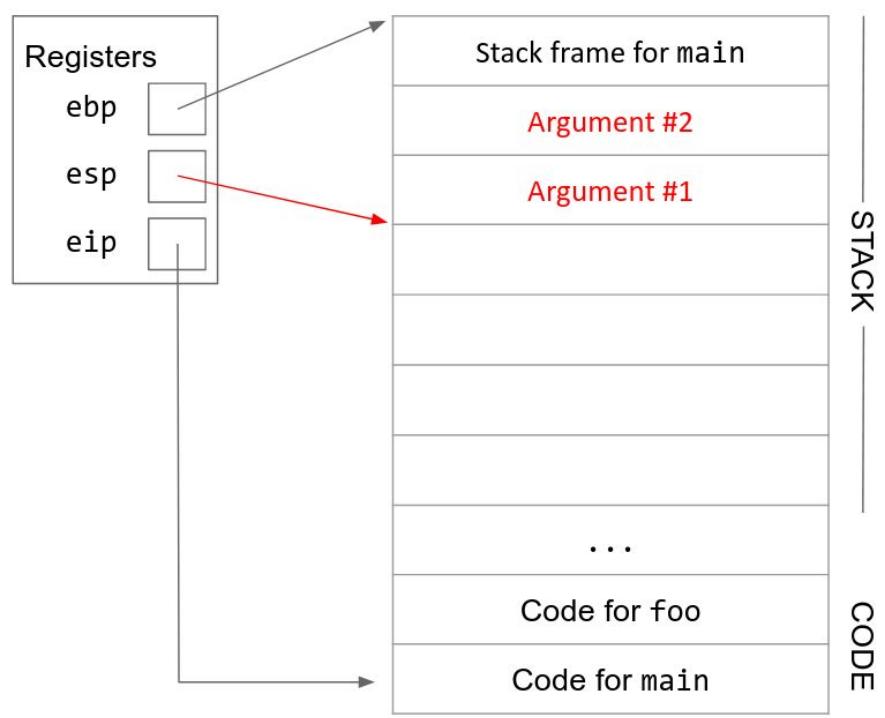
Before pop %eax



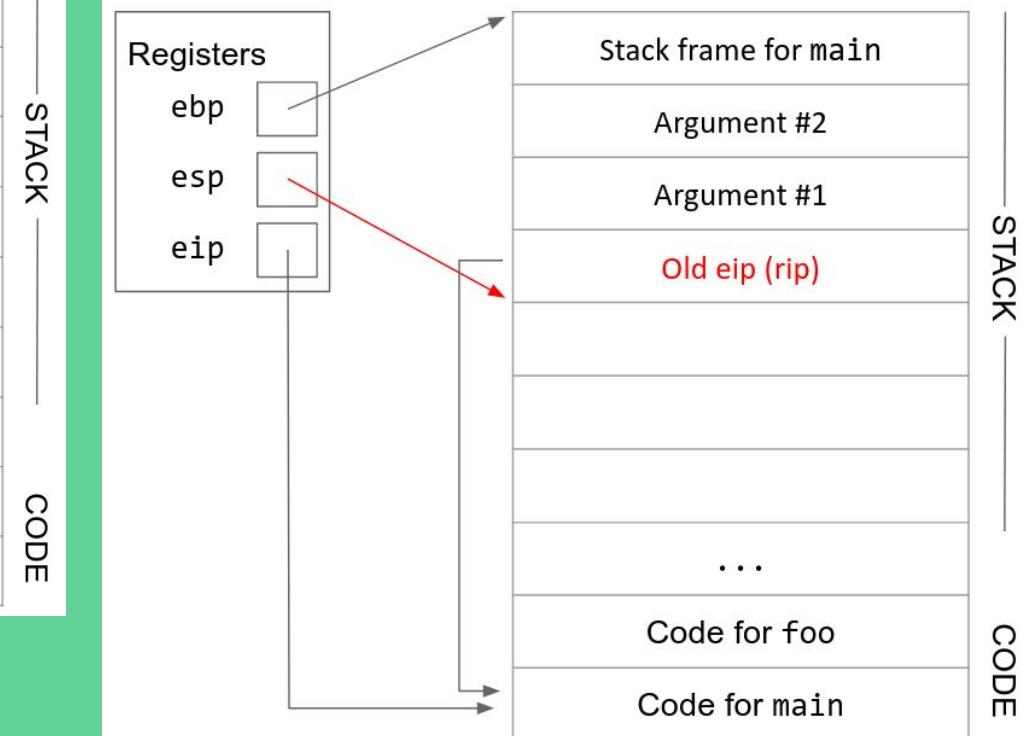
After pop %eax

11 steps to calling an x86 function and returning- main calls function foo

1. Push arguments onto the stack (done in reverse order)

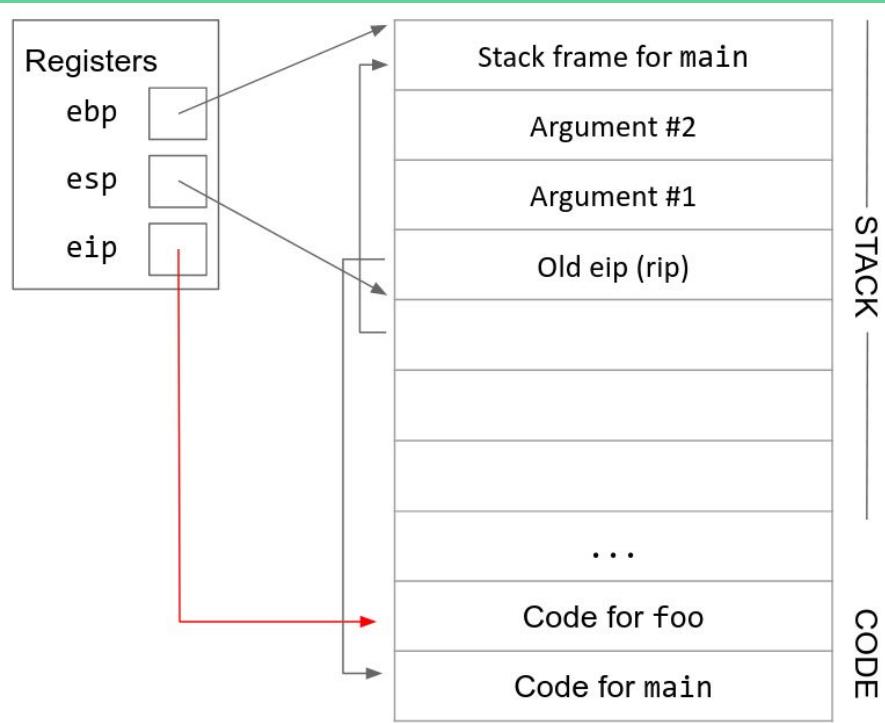


2. Push the old eip (return ip) on the stack

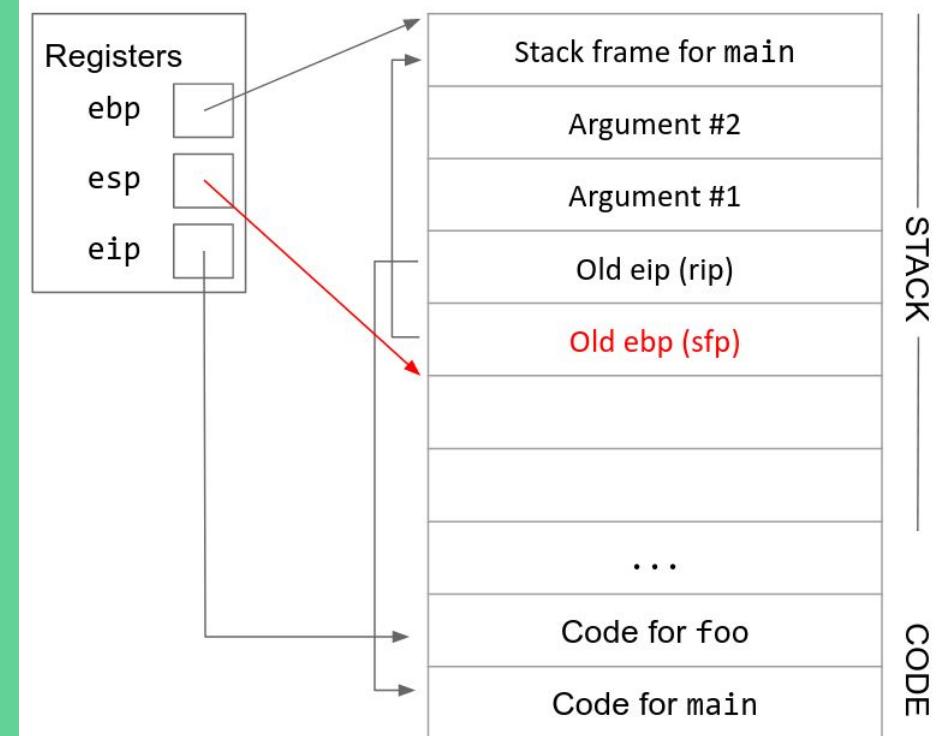


11 steps to calling an x86 function and returning- main calls the foo function-

3. Move EIP (since old value of eip is saved)

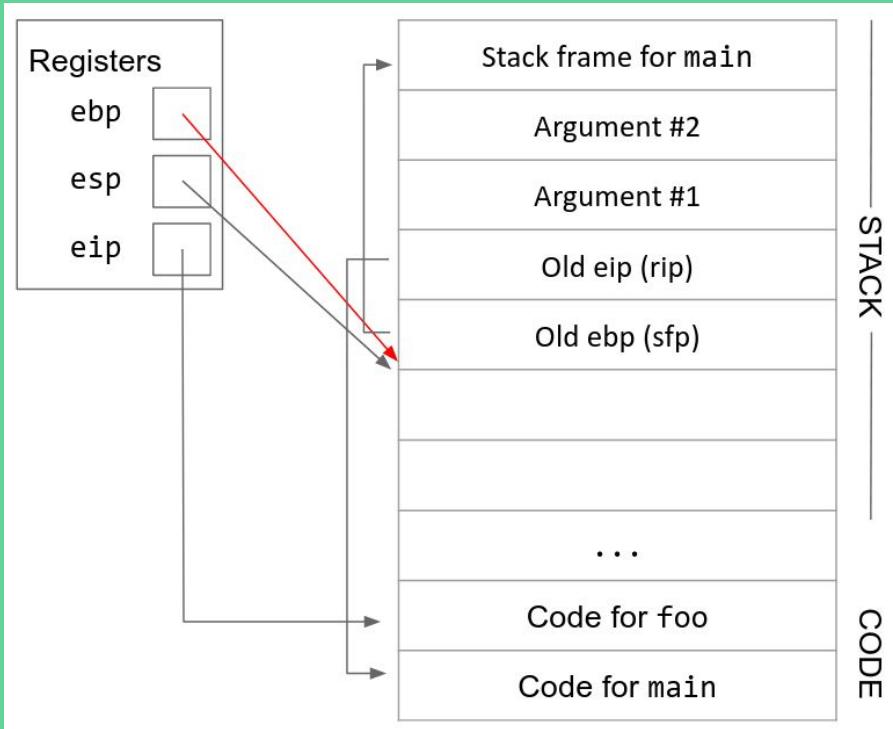


4. Push the old ebp (sfp) on the stack

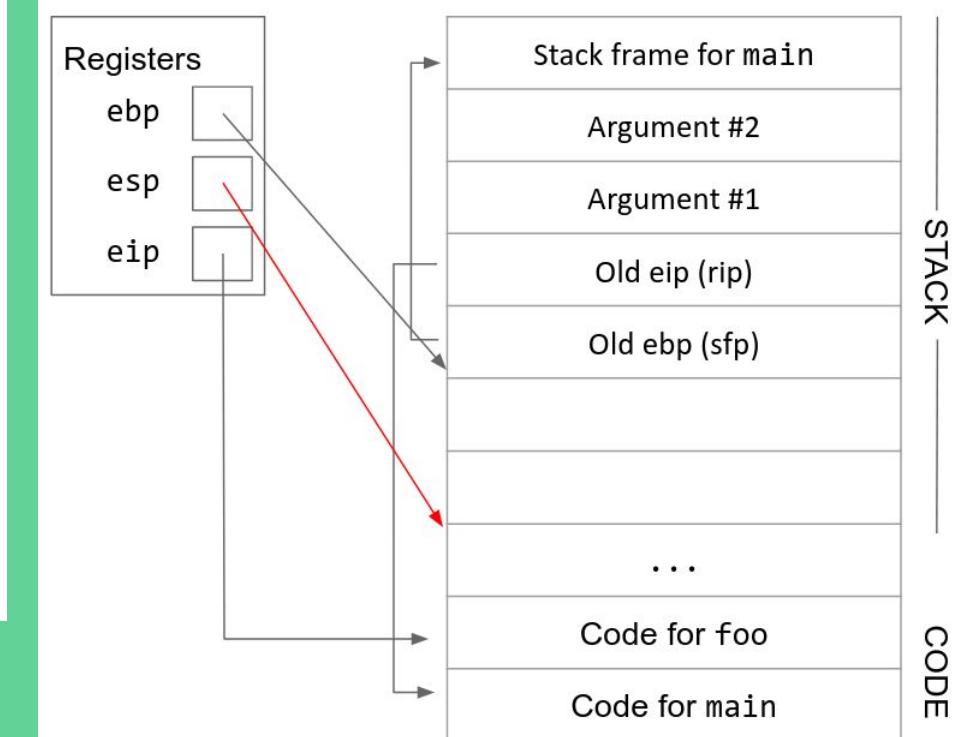


11 steps to calling an x86 function and returning- main calls the foo function-

5. Move ebp as old value of ebp is saved (safely point to top of new stack frame)

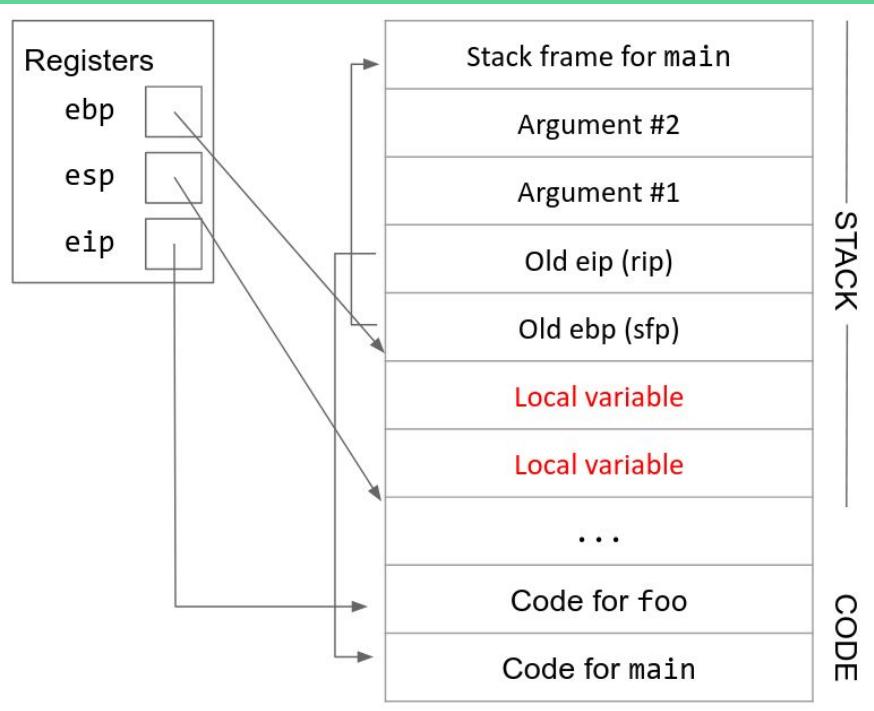


6 dec. ESP to alloc space for new stck fm

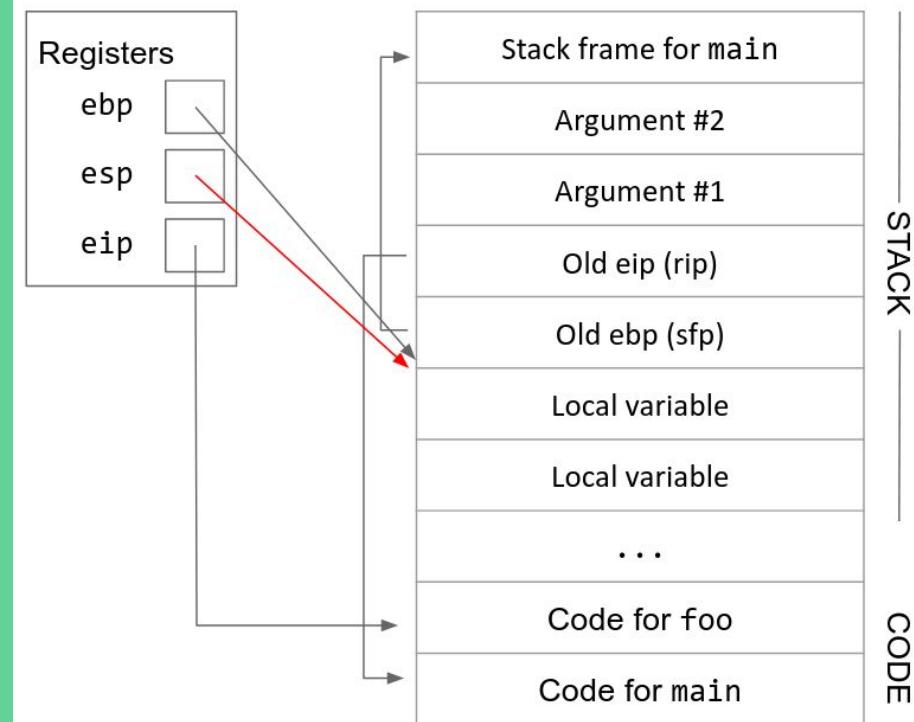


11 steps to calling an x86 function and returning- main calls the foo function-

7. Execute the function

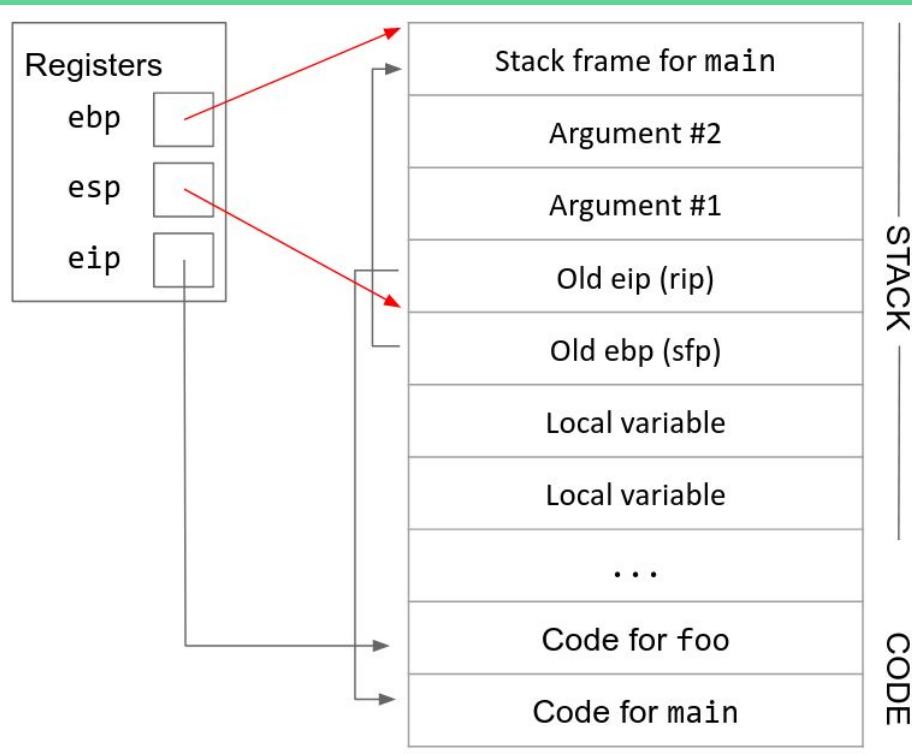


8. To return inc. esp to pt to top of stack

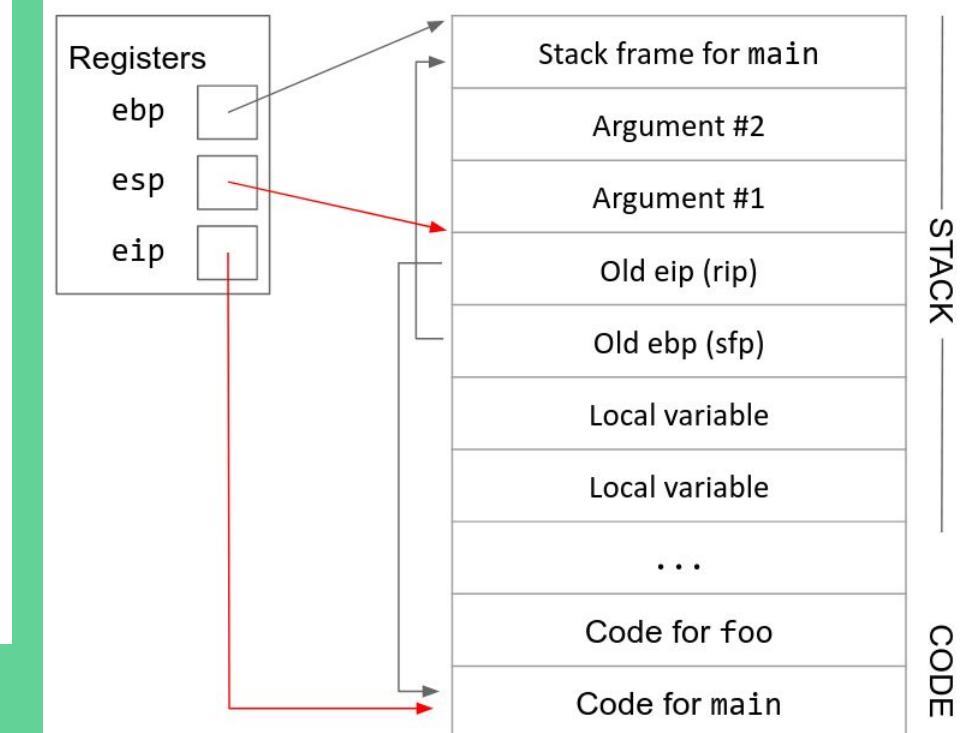


11 steps to calling an x86 function and returning- main calls the foo function-

9. Restore the old ebp (sfp)

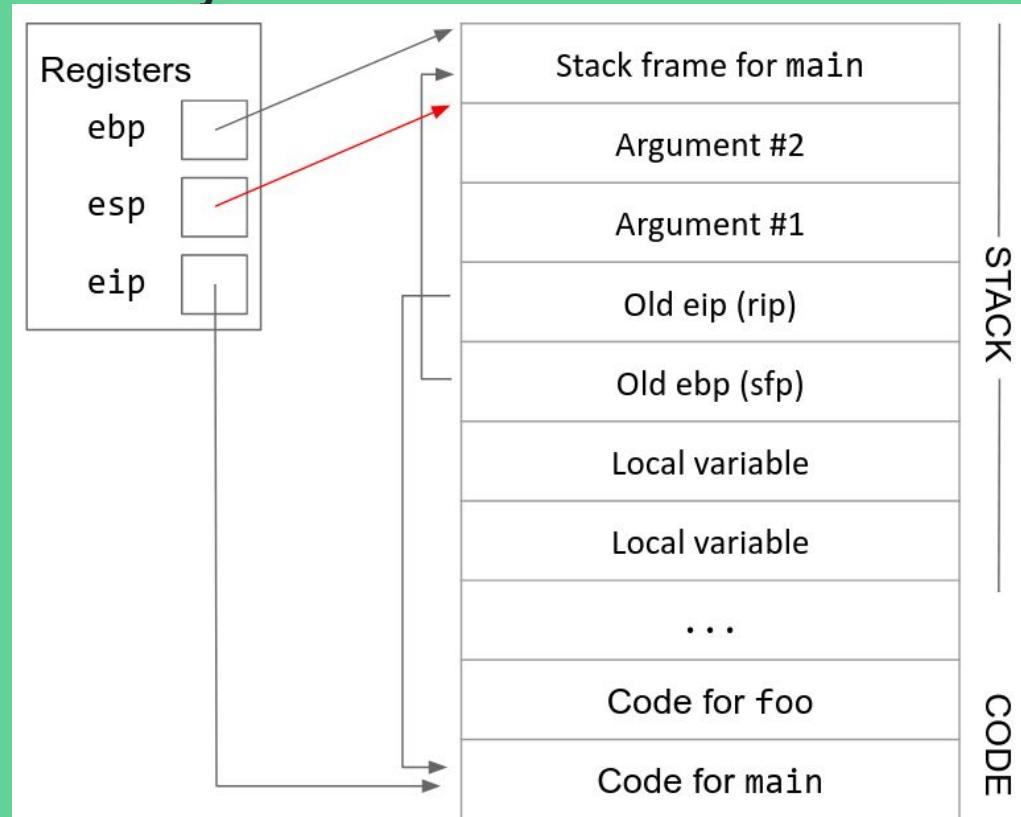


10. Restore the old eip (rip)



11 steps to calling an x86 function and returning- main calls the foo function-

11. Remove arguments from the stack. Since the function call is over, we don't need to store the arguments anymore



```
int main(void) {  
    foo(1, 2);  
}
```

```
void foo(int a, int b) {  
    int bar[4];  
}
```

```
main:  
    # Step 1. Push arguments on the stack in reverse order  
    push $2  
    push $1  
  
    # Steps 2-3. Save old eip (rip) on the stack and change eip  
    call foo  
  
    # Execution changes to foo now. After returning from foo:  
  
    # Step 11: Remove arguments from stack  
    add $8, %esp
```

```
foo:  
    # Step 4. Push old ebp (sfp) on the stack  
    push %ebp  
  
    # Step 5. Move ebp down to esp  
    mov %esp, %ebp  
  
    # Step 6. Move esp down  
    sub $16, %esp  
  
    # Step 7. Execute the function (omitted here)  
  
    # Step 8. Move esp  
    mov %ebp, %esp  
  
    # Step 9. Restore old ebp (sfp)  
    pop %ebp  
  
    # Step 10. Restore old eip (rip)  
    pop %eip
```

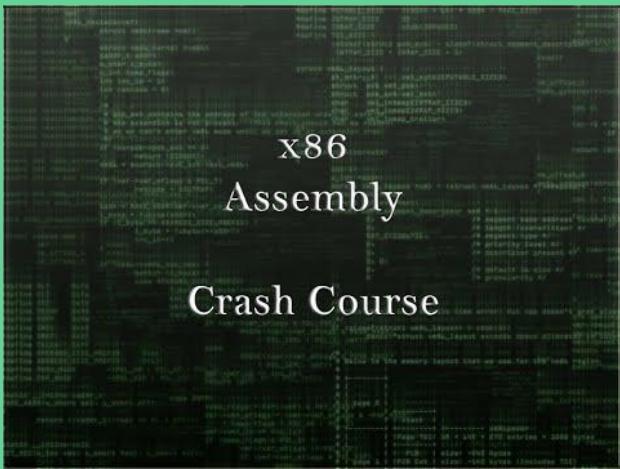
Example of Disassembled code



Disassembling a Program

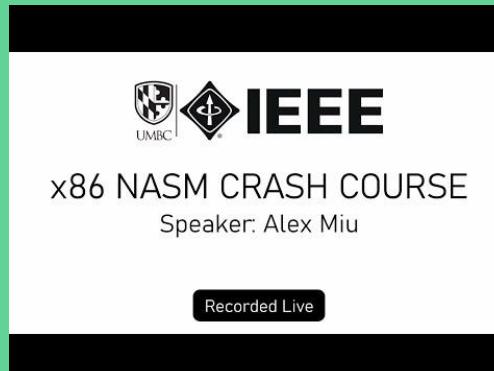
Professor Vijay Chidambaram

TEXAS



Assembly

Crash Course



Hands on

Adding new command to xv6 (done)

[Xv6 Operating System -add a user program - GeeksforGeeks](#)

Adding new system call to xv6 (pending)

<https://viduniwickramarachchi.medium.com/add-a-new-system-call-in-xv6-5486c2437573>

<https://www.geeksforgeeks.org/xv6-operating-system-add-a-user-program/?ref=lbp>

<https://01siddharth.blogspot.com/2018/04/adding-system-call-in-xv6-os.html>

Implementing PS command and required sys call

<https://www.youtube.com/watch?v=a6p-E9JCZc>

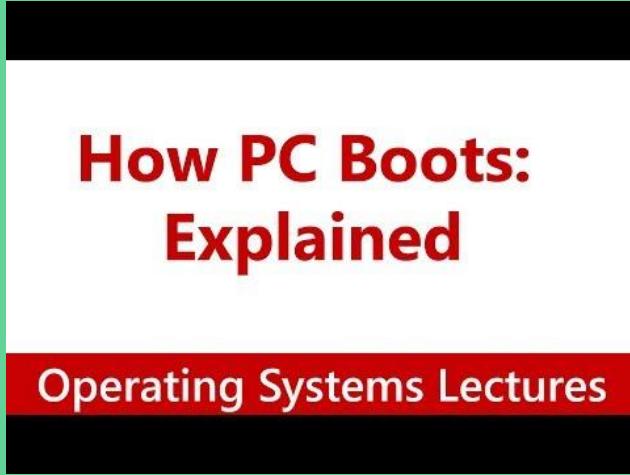
https://www.youtube.com/watch?v=hIXRrv-cBA4&ab_channel=FooSo

Day 2

Day 2 agenda

1. How PC boots - understand significance of 0x7c00, MBR signature 0x55AA
BIOS->MBR-->grub/bootloader->kernel
2. Coding hello-world boot loader
3. Diff between boot loader and linux assembly program helloworld in IA32 linux
4. Assignments -
 - I. 16 bit MBR version of helloworld in x86 assembly
 - II. 32 bit linux version of hello world in 32 bit linux assembly [\(using blog- slide 25\)](#)
 - III. Recreate gdb debug setup for MBR

How xv6 boots- why MBR code is placed at 0x7c00, significance of sign 0x55AA placed as last word



1. On Poweron PC starts at Physical address(CS<<4)+IP= 0xfffff0
2. BIOS is present in a small chip connected to processor
3. Sector 0 of the bootable device disk called MBR is loaded @0x7c00
4. MBR loads the bootloader
5. Bootloader loads OS -covers xv6 bootloader
6. Simplified view of SMP linux boot process

Bootloader printing hello world

```
.code16 # use 16 bits
.global init

init:
    mov $msg, %si # loads the address of msg into si
    mov $0xe, %ah # loads 0xe (function number for int 0x10) into ah
print_char:
    lodsb # loads the byte from the address in si into al and increments si
    cmp $0, %al # compares content in AL with zero
    je done # if al == 0, go to "done"
    int $0x10 # prints the character in al to screen
    jmp print_char # repeat with next byte
done:
    hlt # stop execution

msg: .asciz "Hello world!"

.fill 510-(.-init), 1, 0 # add zeroes to make it 510 bytes long
.word 0xaa55 # magic bytes that tell BIOS that this is bootable
```

Bootloader printing helloworld ..

```
hscuser@GGNLABLVMTRN02:~/pm$ as -o boot.o boot.s
hscuser@GGNLABLVMTRN02:~/pm$ ld -o boot.bin --oformat binary -e init -Ttext 0x7c00 -o boot.bin boot.o
hscuser@GGNLABLVMTRN02:~/pm$ qemu-system-x86_64 boot.bin
WARNING: Image format was not specified for 'boot.bin' and probing guessed raw.
Automatically detecting the format is dangerous for raw images, write operations on block 0 will be restricted.
Specify the 'raw' format explicitly to remove the restrictions.
```

<https://stackoverflow.com/questions/32508919/how-to-produce-a-minimal-bios-hello-world-boot-sector-with-gcc-that-works-from-a>

<https://50lineeofco.de/post/2018-02-28-writing-an-x86-hello-world-bootloader-with-assembly>

BIOS Service 10H- video write char

ostad.nit.ac.ir/payaidea/ospic/file1615.pdf

ostad.nit.ac.ir/payaidea/ospic/file1615.pdf

13 / 53 | - 100% + | ☰

Abacus **Appendix B. BIOS Interrupts and Functions**

Interrupt 10H, function 0EH **BIOS**
Video: Write character

Writes a character at the current cursor position in the current display page. The new character uses the color of the character that was previously in this position on the screen.

Input: AH = 0EH
AL = ASCII code of the character
BL = Foreground color of the character (graphic mode only)

Output: No output

Remarks: This function executes control codes (e.g., bell, carriage return) instead of reading them as ASCII codes. For example, the function sounds a beep instead of printing the bell character.

After this function displays a character, the cursor position increments so that the next character appears at the next position on the screen. If the function reaches the last display position, the display scrolls up one line and output continues in the first column of the last screen line.

The foreground color parameter depends on the current graphic mode. 640x200 bitmapped mode only permits the values 0 and 1. In the 320x200 bitmapped mode, the values 0 to 3 are permitted, which generates a certain color according to the chosen color palette. 0 represents the selected background color; 1 represents the first color of the selected color palette; 2 represents the second color of the color palette, etc.

The contents of the BX, CX, DX registers and the SS, CS and DS segment registers are not affected by this function. The contents of all other registers may change, especially the SI and DI registers.

<https://ostad.nit.ac.ir/payaidea/ospic/file1615.pdf>

<https://stackoverflow.com/questions/22054578/how-to-run-a-program-without-an-operating-system>

MBR assembly version and linux asm helloworld

Source [Linux X86 Assembly – How to Build a Hello World Program in NASM - Security Boulevard](https://securityboulevard.com/2021/05/linux-x86-assembly-how-to-build-a-hello-world-program-in-nasm/)

The screenshot shows a web page from securityboulevard.com. The URL in the address bar is https://securityboulevard.com/2021/05/linux-x86-assembly-how-to-build-a-hello-world-program-in-nasm/. The page has a dark header with various navigation links like ANALYTICS, APPSEC, CISO, CLOUD, DEVOPS, GRC, IDENTITY, INCIDENT RESPONSE, IOT / ICS, and THREATS / BREACH. On the left, there's a vertical sidebar with social sharing icons for Twitter, LinkedIn, Facebook, Reddit, and Email. The main content area contains the assembly code for a hello world program:

```
;#####
; syscall - exit(0);
;#####
mov al, 1      ; Syscall for Exit()
mov ebx, 0      ; The status code we want to provide.
int 0x80        ; Poke kernel. This will end the program
```

The text below the code explains that the comments help explain things, and if removed, the code would be much smaller:

```
global _start
section .data
    msg: db "Hello, World!",0xa
    len: equ $-msg
section .text
_start:
    mov eax, 4
    mov ebx, 1
    mov ecx, msg
    mov edx, len
    int 0x80
    mov al, 1
    mov ebx, 0
    int 0x80
```

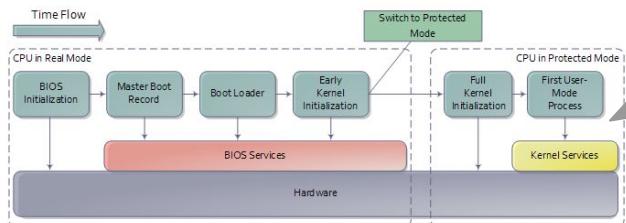
Bootloader “hello world”-bios interrupt, bios services lifespan, boot order, boot sector...

infosecwriteups.com/writing-a-bootloader-931da062f25b

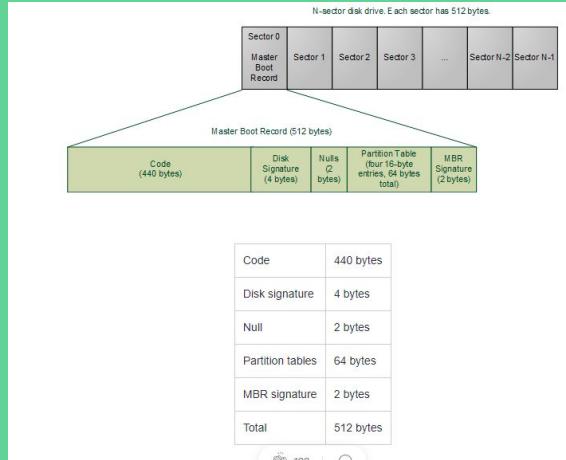
Maps w Java exercises... Dysphagia - Di... (33) Spring Tu... (33) Important... Insta...

What is a Bootloader?

A bootloader is a special program that is executed each time a bootable device is initialized by the computer during its power on or reset that will load the kernel image into the memory. This application is very close to hardware and to the architecture of the CPU. All x86 PCs boot in Real Mode. In this mode, you have only 16-bit instructions. Our bootloader runs in Real Mode and our bootloader is a 16-bit program.



Bios interrupts help OS and application invoke the facilities of the BIOS. This is loaded before the bootloader and it is very helpful in communicating with the I/O. Since we don't have OS-level interrupts (till kernel gets initialized)



<https://infosecwriteups.com/writing-a-bootloader-931da062f25b>

Day 3

Understanding xv6 bootloader - assembly part

Day 3 - xv6 MBR has bootloader - inits to protected mode & loads kernel

0. Big picture - xv6 startup

BIOS-> MBR(bootloader/bootblock.o)

MBR/bootblock -> kernel

1. Understanding xv6 makefile - how bootloader (specifically bootblock) is stored as MBR comprises of -

bootasm.S -code walkthrough is done in 3 parts

bootmain.c (loading the kernel)

2. bootasm.S Part 1 - code walkthrough

3. bootasm.S Part 2 - enable 21 bit addressing -can be skipped

4. bootasm.S remaining code, before part 3-understand x86-32 bit memory mgt arch.

a) paging

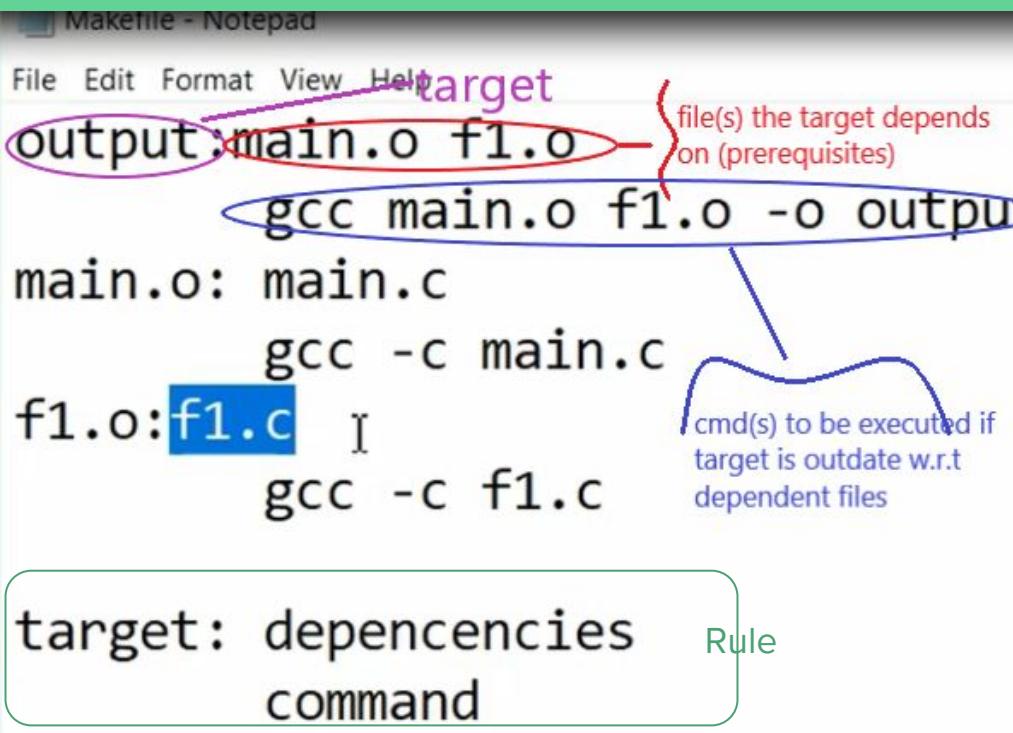
b) multilevel paging c) how segmentation and paging used together

5. bootasm.S part 3

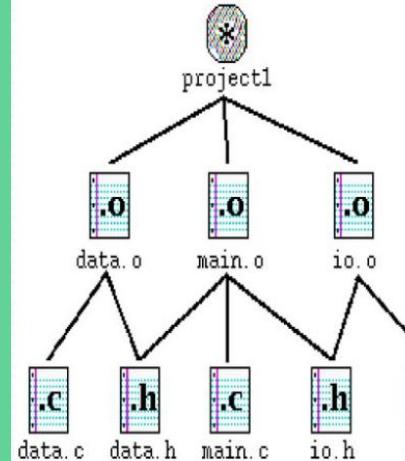
6. bootmain.c part 4 (to be covered in day 4)

Makefile layout - generic

[Makefile Tutorial CIS5027 Prof: Dr. Shu-Ching Chen - ppt download](#)



Dependency graph



Sample Makefile

```
project1: data.o main.o io.o
cc data.o main.o io.o -o project1
data.o: data.c data.h
cc -c data.c
main.o: data.h io.h main.c
cc -c main.c
io.o: io.h io.c
cc -c io.c
```

xv6 Makefile- xv6.img comprises of bootblock & kernel

Objcopy -j text implies only pickup text segment -as
every thing in bootblock is packed in text segment
including manually created gdt table - slide 39 bullet 7
Sign.pl to add sign 0x55aa

```
93 xv6.img: bootblock kernel
94     dd if=/dev/zero of=xv6.img count=10000
95     dd if=bootblock of=xv6.img conv=notrunc
96     dd if=kernel of=xv6.img seek=1 conv=notrunc
97
98 xv6memfs.img: bootblock kernelmemfs
99     dd if=/dev/zero of=xv6memfs.img count=10000
100    dd if=bootblock of=xv6memfs.img conv=notrunc
101   dd if=kernelmemfs of=xv6memfs.img seek=1 conv=notrunc
102
103 bootblock: bootasm.S bootmain.c
104     $(CC) $(CFLAGS) -fno-pic -O -nostdinc -I. -c bootmain.c
105     $(CC) $(CFLAGS) -fno-pic -nostdinc -I. -c bootasm.S
106     $(LD) $(LDFLAGS) -N -e start -Ttext 0x7C00 -o bootblock.o bootmain.o
107     $(OBJDUMP) -S bootblock.o > bootblock.asm
108     $(OBJCOPY) -S -O binary -j .text bootblock.o bootblock
109     ./sign.pl bootblock
110
111 entryother: entryother.S
112     $(CC) $(CFLAGS) -fno-pic -nostdinc -I. -c entryother.S
113     $(LD) $(LDFLAGS) -N -e start -Ttext 0x7000 -o bootblockother.o entryother.o
114     $(OBJCOPY) -S -O binary -j .text bootblockother.o entryother
115     $(OBJDUMP) -S bootblockother.o > entryother.asm
116
117 initcode: initcode.S
118     $(CC) $(CFLAGS) -nostdinc -I. -c initcode.S
119     $(LD) $(LDFLAGS) -N -e start -Ttext 0 -o initcode.out initcode.o
120     $(OBJCOPY) -S -O binary initcode.out initcode
121     $(OBJDUMP) -S initcode.o > initcode.asm
122
123 kernel: $(OBJS) entry.o entryother initcode kernel.ld
124     $(LD) $(LDFLAGS) -T kernel.ld -o kernel entry.o $(OBJS) -b binary initcode entryother
125     $(OBJDUMP) -S kernel > kernel.asm
126     $(OBJDUMP) -t kernel | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$/d' > kernel.sym
127
```

count=10K x 512 bytes implies create file filled with 0s of size ~5 Mb

```
hscuser@GGNLABLVMTRN25:~/neeraj/pub_xv6$ ls -hl xv6.img
-rw-rw-r-- 1 hscuser hscuser 4.9M Nov 26 14:54 xv6.img
hscuser@GGNLABLVMTRN25:~/neeraj/pub_xv6$
```

The **notrunc** conversion option means do not truncate the output file — that is, if the output file already exists, just replace the specified bytes and leave the rest of the output file alone.

Source : [dd - why do people use "notrunc" when making an ISO from a DVD? - Unix & Linux Stack Exchange](https://unix.stackexchange.com/questions/103333/dd-why-do-people-use-notrunc-when-making-an-iso-from-a-dvd)

bootblock is located @0x7c00 & line 109
adding required signature

Makefile layout - xv6 makefile - 1

Understanding code block from XV6 makefile

Asked 7 years, 8 months ago Modified 7 years, 8 months ago Viewed 614 times

I'm trying to understand the following code block from XV6 makefile :

```
1 ULIB = ulib.o usys.o printf.o umalloc.o  
2  
3 %: %.o $(ULIB)  
4     $(LD) $(LDFLAGS) -N -e main -Ttext 0 -o $@ $^  
5     $(OBJDUMP) -S $@ > $*.asm  
6     $(OBJDUMP) -t $@ | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$/d' > $*.sym
```



Automatic variables help in removing redundancy p.s. video

\$(LD) \$(LDFLAGS) -N -e main -Ttext 0 -o \$@ \$^

Is, if I understand it right, a mixture of `bash` and `make` syntax:

- `$(LD)` is replaced by the `make` variable `LD`, which most likely holds the name of the linker executable (usually `ld`).
- `$(LDFLAGS)` is like the above, with the difference that it holds the `flags` to pass to the executable named in `LD`.
- `-N -e main -Ttext 0 -o` are just arguments to `LD`
- `$@` is replaced by the target
- `$^` is replaced by a space-separated list of all dependencies

Makefile layout - xv6 makefile - 2

I have three files in my XV6: testmain.c, foo.h, and foo.c :

0 foo.h:

```
extern void myfunction(void)
```

1 foo.c:

```
#include "foo.h"
void myfunction(void){
    printf(1, "HelloWorld"); }
```

2 testmain.c:

```
#include "foo.h"
int main(void){
    myfunction();
    return 0 ; }
```

You need several changes in `Makefile`:

- Indicate that you want to create `_testmain` program,
- Tell what `_testmain` dependencies are (if apply).

add `_testmain` in programs list:

```
UPROGS=\
    _testmain\
    _cat\
    _crash\
    _echo\
    _factor\
    ....
```

`_testmain` dependencies:

Since your `_testmain` program depends on two files, you must create a special rule telling that to builder (I make this rule from `%.o: %.c $(ULIB)` rule):

```
_testmain: testmain.o foo.o $(ULIB)

$(LD) $(LDFLAGS) -N -e main -Ttext 0x1000 -o $@ $^
$(OBJDUMP) -S $@ > $*.asm
$(OBJDUMP) -t $@ | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$/d' > $*.sym
```

Line by line analysis- xv6 makefile -3

The Makefile of Xv6 is a file used to organize the compilation process of the source code. The Makefile describes the interdependence between each source file, and simplifies the original tedious compilation process to the cooperation of the makefile and the make command. use. The general structure of Makefile is a tree structure, that is, a target is divided into multiple sub-targets. Make takes the first target of the makefile as the final target, and all rules that this rule depends on will be executed.

Compilation process:

- (1) First OBJS specifies the .c file that needs to be compiled
- (2) Then configure some environment and tool information
- (3) xv6.img, bootblock, kernel depend on each other to generate the system kernel and image
- (4) Generate the file system and Executable files inside the system
- (5) Configuration startup information

Source code comment and analysis:

- (1) Compilation target definition

```
1 OBJS = \
2     bio.o\
3     console.o\
4     exec.o\
```

bootasm.S

Zero out sg regs
code to enable above 20 bit
addressing -can be skipped

```
#include "asm.h"
#include "memlayout.h"
#include "mmu.h"

# Start the first CPU: switch to 32-bit protected mode, jump into C.
# The BIOS loads this code from the first sector of the hard disk into
# memory at physical address 0x7c00 and starts executing in real mode
# with %cs=0 %ip=7c00.

.code16          # Assemble for 16-bit mode
.globl start
start:
    cli           # BIOS enabled interrupts; disable
    # Zero data segment registers DS, ES, and SS.
    xorw %ax,%ax      # Set %ax to zero
    movw %ax,%ds      # -> Data Segment
    movw %ax,%es      # -> Extra Segment
    movw %ax,%ss      # -> Stack Segment
    # Physical address line A20 is tied to zero so that the first PCs
    # with 2 MB would run software that assumed 1 MB. Undo that.
seta20.1:
    inb $0x64,%al      # Wait for not busy
    testb $0x2,%al
    jnz seta20.1
    movb $0xd1,%al      # 0xd1 -> port 0x64
    outb %al,$0x64

seta20.2:
    inb $0x64,%al      # Wait for not busy
    testb $0x2,%al
    jnz seta20.2
    movb $0xdf,%al      # 0xdf -> port 0x60
    outb %al,$0x60

# Switch from real to protected mode. Use a bootstrap GDT that makes
bootasm.S
```

```
# Switch from real to protected mode. Use a bootstrap GDT that makes
# virtual addresses map directly to physical addresses so that the
# effective memory map doesn't change during the transition.
lgdt gdtdesc
movl %cr0, %eax
orl $CR0_PE, %eax
movl %eax, %cr0

//PAGEBREAK!
# Complete the transition to 32-bit protected mode by using a long jmp
# to reload %cs and %ip. The segment descriptors are set up with no
# translation, so that the mapping is still the identity mapping.
ljmp $(SEG_KCODE<<3), $start32

.code32 # Tell assembler to generate 32-bit code now.
start32:
    # Set up the protected-mode data segment registers
    movw $(SEG_KDATA<<3), %ax    # Our data segment selector
    movw %ax, %ds      # -> DS: Data Segment
    movw %ax, %es      # -> ES: Extra Segment
    movw %ax, %ss      # -> SS: Stack Segment
    movw $0, %ax        # Zero segments not ready for use
    movw %ax, %fs      # -> FS
    movw %ax, %gs      # -> GS

    # Set up the stack pointer and call into C.
    movl $start, %esp
    call bootmain

    # If bootmain returns (it shouldn't), trigger a Bochs
    # breakpoint if running under Bochs, then loop.
    movw $0x8a00, %ax      # 0x8a00 -> port 0x8a00
    movw %ax, %dx
    outw %ax, %dx
    movw $0x8ae0, %ax      # 0x8ae0 -> port 0x8ae0
    outw %ax, %dx
spin:
    jmp spin
```

This part of the code
is meant for entering
into protected mode

To understand this we
need to understand
x86 memory mgt arch

Memory Management - virtual memory

Virtual Memory

Operating Systems Lectures

Working of Virtual Memory

Operating Systems Lectures

- 1 **virtual memory** RAM is split into fixed size partitions called page frame
For a running process current page(s) & entire page table should be in ram
- 2.1-1 mapping not possible & not required as processes can start & stop in random order.
- 3 Every memory access has additional overhead of first page table access
& then actual memory location, -h/w mechanism TLB cache to mitigate this overhead. At any pt. in time pages from multiple processes will be there
- Q. Can we read process page table using user space program?
4. Meets requirement of separate/ isolated per process address space -protection@6:30
5. **On process start** page tbl is created & only 1-2 pages are loaded, rest marked as absent - lazy loading of pages on demand@9:50, else leads to poor UI, ideally RAM should sized w.r.t workload
- 6 Page table has presence, dirty bit & protection bits -rw swap in on need basis -@15

1 Virtual address space of Process - contiguous address space in ELF (objdump -ph)

- 2 MMU translates -VA to PA before raising read request on memory bus-using page table of process
- 3.4K page so 32 bit address is split into 12 bit offset & remaining 20 bit as page index
4. $2^{20} \Rightarrow 1\text{MB}$ entries of 32 bits, 4MB
of contiguous memory is an issue
Thus 2 levels outer (pg dir) and inner (pg tbl) entry pointing to l2 table
5. Working of VM loading page through page fault

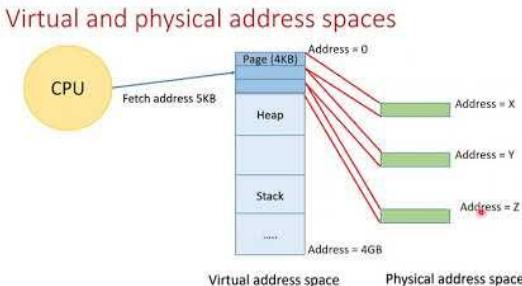


Memory Management - segmentation

Segmentation & Fragmentation

Operating Systems Lectures

- 1 segmentation - to split program into various segments -text, data, stack, heap etc- logical view to physical view address mapping
- 2.Need 3 registers a)Ptr to descriptor table Gdtr ,b)segment selector CS & c)offset register IP AND Segment descriptor table- base address, limit
- 3.Each 64 bit descriptor entry comprises bit base, bit limit value & access rights 0-3
- 4.**Segment selector is 16 register - CS for code access, DS for data...points to offset in GDT , e.g. code access CS has index into gdt table and register IP as offset**
5. **Mapping is -Logical to linear to Virtual address -Some what like real mode (e.g. CS<<4+IP), in protected mode CS & IP work together- CS acts as selector or index in gdt table**
6. Segmentation leads to fragmentation
7. @8:30 -Bit confusing 2 diff ways of splitting memory - segmentation and paging - anticimex segmentation in xv6 is kind of disabled - next vblog



- 1 segmentation - to split program into various segments -text, data, stack, heap etc- logical view to physical view address mapping
- Segment descriptor table- base address, limit
- values of base is 0, limit is 4GB, CS, DS...thus no effect of base and limit. Two entries in segment descriptor table for flat segments - Segmentation doesn't change the address - only paging changes. **Only value add checking privileges 1st segment descriptor table entry is for code (with +x) and second for code- thus segmentation is only used for permissions/has limited use**
- 3 Modern OSs use dummy flat segments but have diff segments for diff privileges (details later)

Real mode Vs protected mode

- Real mode 16 bit registers 
- Protected mode
 - Enables segmentation + Paging both
 - No longer seg*16+offset calculations
 - Segment registers is index into segment descriptor table. But segment:offset pairs continue
 - `mov %esp, $32 # SS will be used with esp`
 - More in next few slides
 - Other segment registers need to be explicitly mentioned in instructions
 - `Mov FS:$200, 30`
 - 32 bit registers
 - can address upto 2^{32} memory

X86 segmentation

code access selector is CS and offset is IP,
for address mentioned as part of instruction selector is DS..

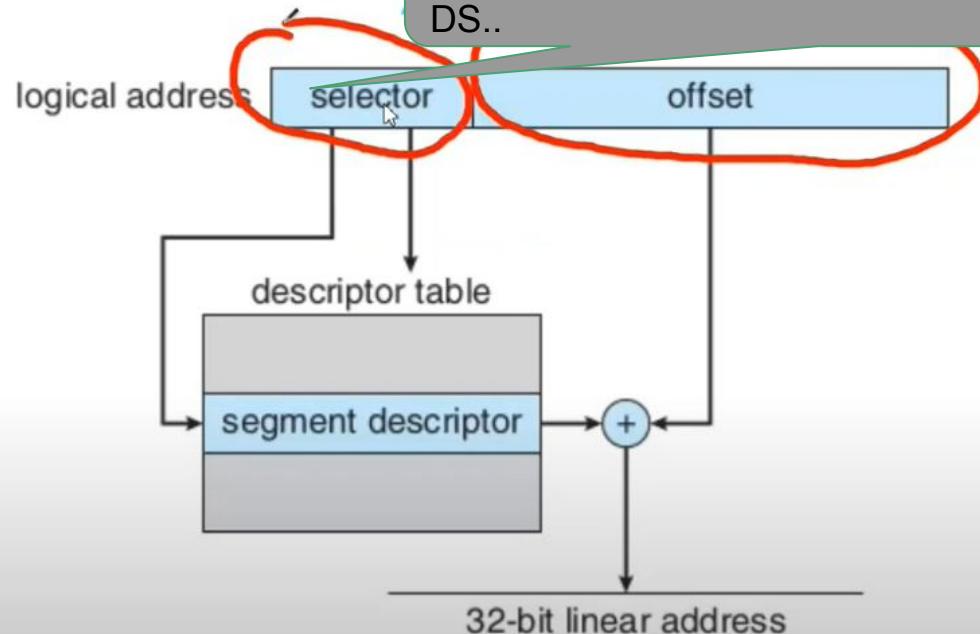
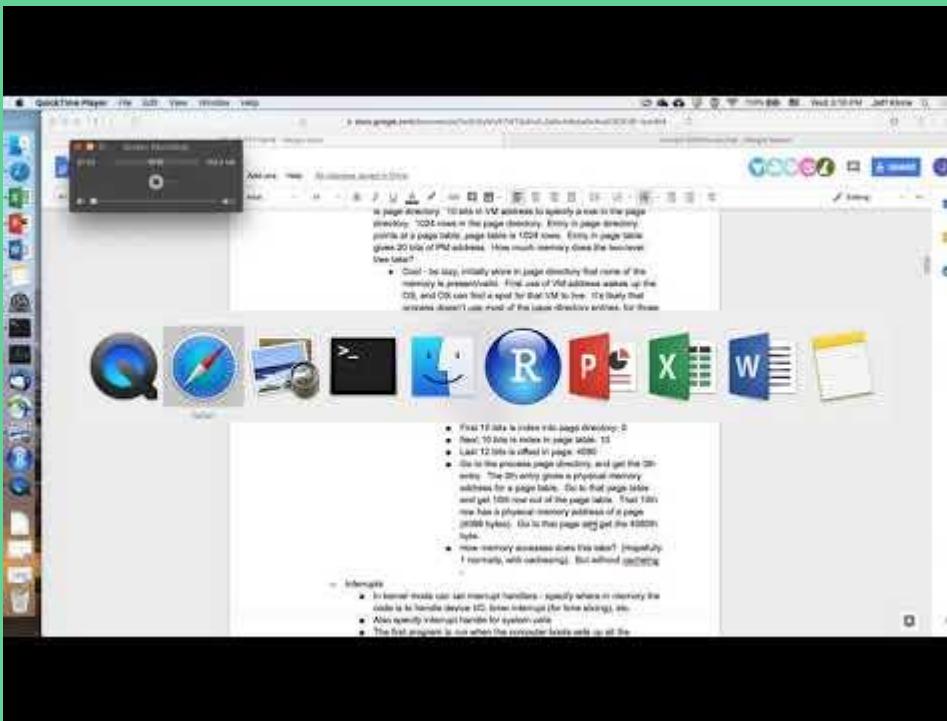


Figure 8.22 IA-32 segmentation.

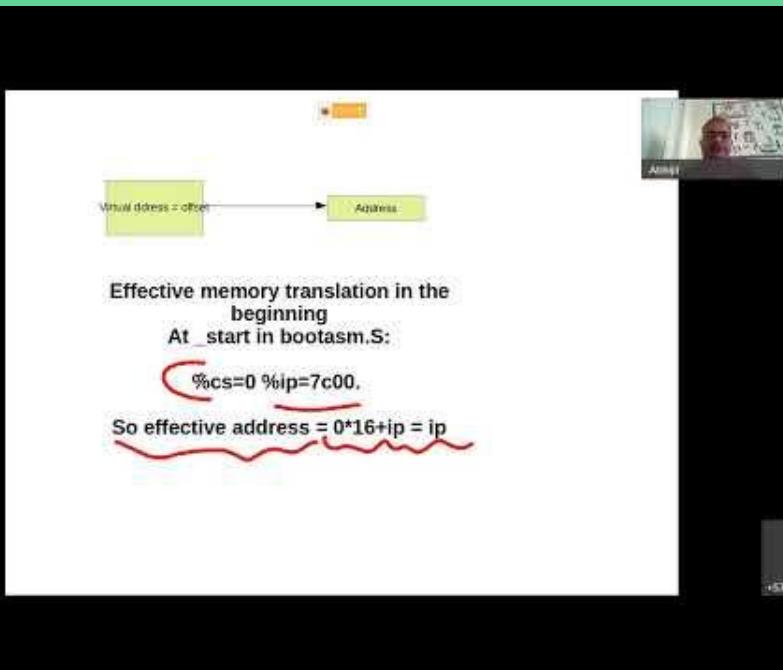
Assignment -Memory Management to walk a pointer VA (virtual address) to PA (physical address) mapping - first 12 mins



[Adding System call in xv6. what is xv6 ? | by Mahima Kothari | Medium](#)

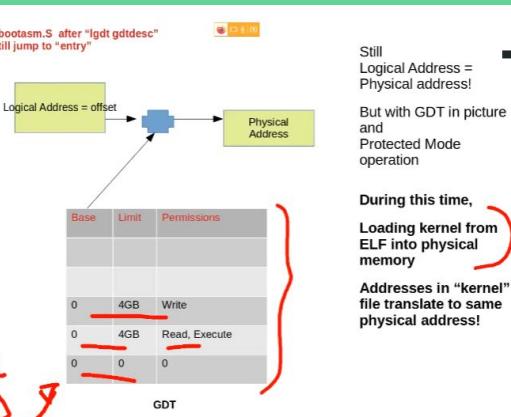
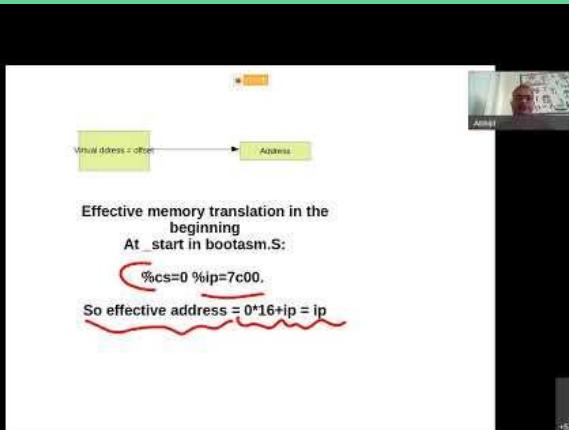
Walkthrough of virtual address to physical address mapping for variable - page directory, page table and physical address

Bootloader part 1 & 2 - what does it do 1/2



1. xv6 Makefile bootblock made up of bootasm.S & bootmain.c
2. bootasm.S is first binary and is meant to be located @0x7c00
3. cli -clear all interrupts - no ISRs are installed/expected as well.
4. Some code to enable above 20 (21) bit addressing -can be skipped
5. Line 42 -51 - are meant to switch from real to protected mode and for that we need to apply our understanding of memory management arch of intel processor
6. Before that lets see real vs protected mode - [slide 35](#)
 - Segment register (CS) and offset reg IP pair is still used but not like cs<<4+ IP, but in different way
 - Logical -> linear-> physical
 - gdtr register
 - Concept of paging how it gets extended to -2 level paging for 4k pages, other option is 1 level paging for 4 Mb pages
 - Level 2 - level 1 is called PD & level 2 is called PT
 - PDE PTE formats @26.26
 - @27:15 CR3 gives us base of page directory
 - @27:30 scheme of memory mgt with 4mb pages
 - @29:20 role of kernel is to setup gdt, gdtr, cr3 and PD

Bootloader Part 2 - segmentation setup & switch to protected mode



1. Segmentation setup base 0 limit 4G (is ignored)
2. During loading of kernel paging is not enabled (i.e. when bootloader is running) on this dummy segmentation is in use
3. After kernel is loaded initially 4MB pages are used & later kernel is switched to 4KB pages
4. 64 bit GDT entry layout & SEGM_ASM macro for populating GDT entry as per split bitmap
5. Segment selector in protected mode - meaning of 16 bits
6. @36:50 equipped with intel MM concepts - let's return to code
7. @39:30 instructions and data are combined into .text
8. p.s image @42:50 after ldgt instruction - how gdtr & descriptor array is setup (array has 3 entries- null entry, 0-4G r x, 0-4G w)-**how seg is effectively disabled?**
9. entire 4G memory as single segment
10. Enable protected mode by setting LSB/bit of CR0
11. Execute ljmp \$(SEG_KCODE<<3), \$start32
 - a. @46:29 to populate seg register by skipping lower 3 bits
 - b. Transition to 32 bit (started with CR0 bit) completes with ljmp inst.
 - c. DS, SS, ES initialized to 2
 - d. 0x7c00 is now a scratch pad area and can be used as stack area for bootmain() - C code that loads kernel ELF from disk into memory

Day 4 - 12/16
Understanding xv6
bootloader - C part
Kernel bringup

Day 4 - xv6 MBR Bootmain.c loads kernel and kernel init

0. Big picture - xv6 startup

BIOS-> MBR(loader/bootblock.o)

MBR/loader -> kernel

1. xv6 makefile

....

5. bootasm.S part 3 (covered in day 3 e.g lgdt)-segmentation setup & switch to protected mode ([slide 41](#))

6. bootmain.c part 4

7. Kernel

Initialization- assembly part -entry.S

Main.c -virtual memory setup -identity mapping

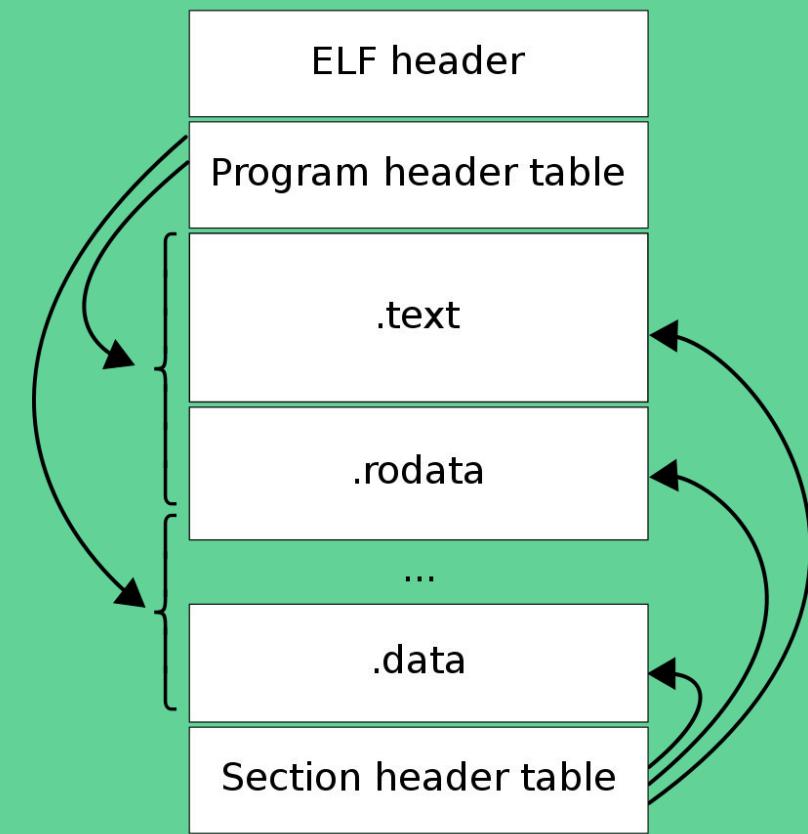
8. Writing first system call

C bootmain is loader for kernel, initial Assembly code of kernel -enabled paging but in a smart way

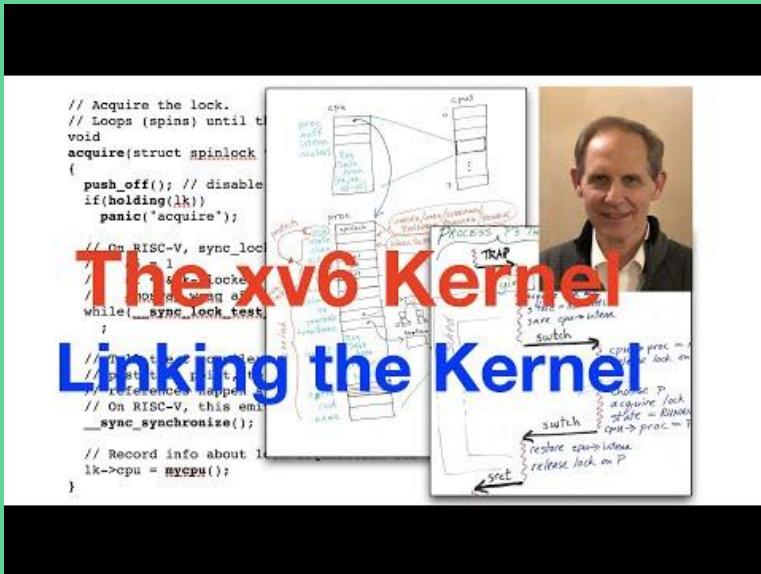


1. Recap of bootblock (bootasm.S) assembly part
 - a. ldgldr -glb desc tbl is array of 3 entries & ldgtr inst. make gdtr pt to that array **Q how segmentation is effectively disabled?**
 2. Bootmain loader is doing one run of for loop for each segment of kernel program
 3. objdump -p @21:05, above 0x100,000 (1MB) u have real memory
 4. objdump -f kernel -1st instruction to be executed
 5. Walkthrough of entry() assembly code
 - a. Load entrypgdir into page table & enable 4MB paging
 6. Contents of Entrypgdir- just 2 valid entries from possible 2^{20} entries - both identity mappings (p.s. pic)
 7. Earlier I couldn't access address 2GB+0x10 now with VM enabled & paging entry mapped will result in valid access
 8. All data and stack are living in physical 0- 4MB, as hardcoded 2 entries ensures 0-4MB is accessible through virtual addressing as well
 - a. Carpet is not pulled under my feet, addresses are still valid
 - b. pulling off carpet under the feet could have resulted w/o entrypgdir initialization or specifically 2 manual entries
 9. Entrypgdir is temporary page table for switching to virtual addressing
 10. Prepare stack and jmp to main() - c entry point of kernel

ELF file format



Linker script for xv6 kernel - risc v 64



1. Linkers is something that we don't think about, most of the user programs default works- but not for kernel
 2. How kernel is laid out in memory -different markers
 3. Walkthrough of linker file for risc 64 bit build
 4. Concepts are similar

Day 5

- Why learning xv6 to understand linux kernel
- vscode & gef GDB for debugging xv6 startup code

xv6 - toy operating system to learn how kernel works?



1. Intro

Educational OS built by porting Unix Kernel

- **Implementations - x86 (32 bit) & RISC-V (64 bit)**
x86 -32 bit, 64 bit
- **Emulated using QEMU**
Meant to run on bare metal (physical hardware)
- **Simple code base in C & assembly**
~ 6000 of code max in C, (300 lines -startup/context switch in assembly)
Simple well written clean code- meant to learn kernel implementation
- **Multicore OS QEMU can emulate multicore h/w**

xv6 Features

Processes with virtual address spaces Page tables for physical address mapping

Files directory
Pipes

Multitasking Time Slicing

21 system calls
commercial linux 300+

Usr Programs

sh
cat
echo
grep
kill
ln
ls
mkdir rm
wc- only 10 (real linux lots of apps)

Missing

Real OS x 100 code
No user ids
No file protection
Paging to disk
Mountable FS
Socket/networking
Device drivers
-couple (kbd/console, disk, timer..)

```
neeraj.arora@IBS-LAP-095:~/bak_xv6/xv6-public$ ls *.c
bio.c      echo.c     fs.c      ioapic.c   lapi.c    main.c      mp.c      proc.c      spinlock.c   sysfile.c   ulib.c      wc.c
bootmain.c exec.c     grep.c     kalloc.c   ln.c      memide.c   picirq.c   rm.c      stressfs.c   sysproc.c   umalloc.c   zombie.c
cat.c      file.c     ide.c     kbd.c     log.c      mkdir.c    pipe.c    sh.c      string.c    trap.c     usertests.c
console.c  forktest.c init.c    kill.c    ls.c      mkfs.c    printf.c  sleeplock.c syscall.c   uart.c     vm.c
neeraj.arora@IBS-LAP-095:~/bak_xv6/xv6-public$ grep "main(int" *.c
cat.c:main(int argc, char *argv[])
echo.c:main(int argc, char *argv[])
grep.c:main(int argc, char *argv[])
kill.c:main(int argc, char **argv)
ln.c:main(int argc, char *argv[])
ls.c:main(int argc, char *argv[])
mkdir.c:main(int argc, char *argv[])
mkfs.c:main(int argc, char *argv[])
rm.c:main(int argc, char *argv[])
stressfs.c:main(int argc, char *argv[])
usertests.c:main(int argc, char *argv[])
wc.c:main(int argc, char *argv[])
neeraj.arora@IBS-LAP-095:~/bak_xv6/xv6-public$ █
```

Day 6

- Day 5 recap
- xv6 completing kernel init C part -startup
- What are system calls
 - Code walkthrough of system call interface, Interrupt handling, initial process creation...
- Process management code base xv6
 - https://namsick96.github.io/os/OS_process/
- System calls in xv6- hello world homework?
- [xv6 -Implementing ps, nice system calls and priority scheduling | by Harshal Shree | Medium](#)

Summary view

The main thing is to load the kernel , Kernel is a elf file from sector 1 & onwards

- 3 Points -
- 3 disk functions-
- ELF loader
- call entry()

On power/REST
CS=0xf000 IP=0xffff0

Sector 0/MBR

bootblock.o

Bootasm.S

Bootmain.c

Starts execution in 16 bit real mode.

Mainly did one thing : Enter into protection mode, There are 4 main steps :

1. open A20 ->
2. Build load GDT (`l1gdt gdtdesc`) ->
3. Set up CR0 register (protected mode) ->
4. start32 & call bootmain

(written in assembly init stack, env. for C)

```
movl    $start, %esp    # 0x7c00 Set top of stack
call    bootmain
```

Mainly did one thing , Turn on paging & jump to main, There are four main steps :

Build page tables -> Load page table -> Set up CR3 register -> Jump to main (written in C)

kernel

entry.S

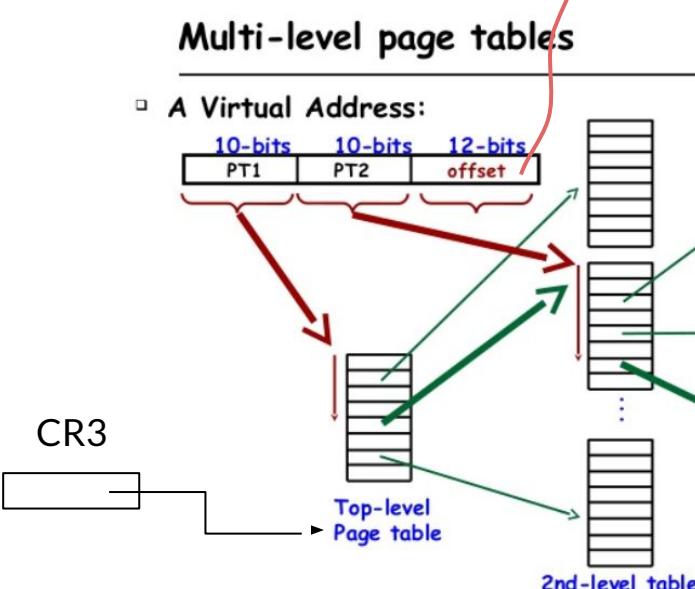
main() init and start entire world of user processes

```
1217 main(void)
1218 {
1219     kinit(end, P2V(4*1024*1024)); // phys page allocator
1220     kvmalloc(); // kernel page table
1221     mpminit(); // collect info about this machine
1222     lapicinit();
1223     seginit(); // set up segments
1224     cprif("ncpu%d: starting xv6\n\n", cpu->id);
1225     picinit(); // interrupt controller
1226     ioapicinit(); // another interrupt controller
1227     consoleinit(); // I/O devices & their interrupts
1228     uartinit(); // serial port
1229     pinit(); // process table
1230     tvinit(); // trap vectors
1231     binit(); // buffer cache
1232     fileinit(); // file table
1233     iinit(); // inode cache
1234     ideinit(); // disk
1235     if(lismp)
1236         timerinit(); // uniprocessor timer
1237     startothers(); // start other processors
1238     kinit(P2V(4*1024*1024), P2V(PHYSTOP)); // must come after startothers()
1239     userinit(); // first user process
1240     // setting up this processor in mpmain.
```

Multi level Page table

stackoverflow.com/questions/29837520/does-an-entry-of-page-table-represents-a-page-or-a-linear-address

stackoverflow About Products For Teams Search...
Questions Tags Users Companies
COLLECTIVES Explore Collectives TEAMS
Stack Overflow for Teams – Start collaborating and sharing organizational knowledge.
Free Create a free Team Why Teams?



Remember, each page table entry holds a virtual address. It is the responsibility of the operating system to translate virtual addresses to physical addresses (the benefits of which are outside of this particular topic).

Most paging systems also maintain a frame table that keeps track of used and unused frames. The

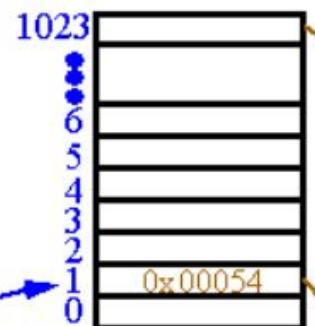
Which location in page

Does a page table represent a physical address? - Stack Overflow

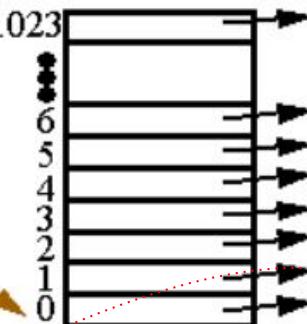
Multilevel Page Tables

- Instead of using only one level of indirection, use two levels.

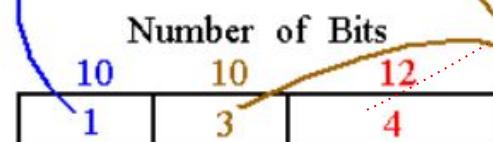
Top-level page table



4K
Page
Frames



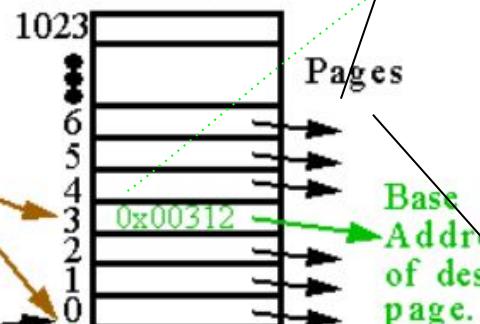
● Second-level
● page tables



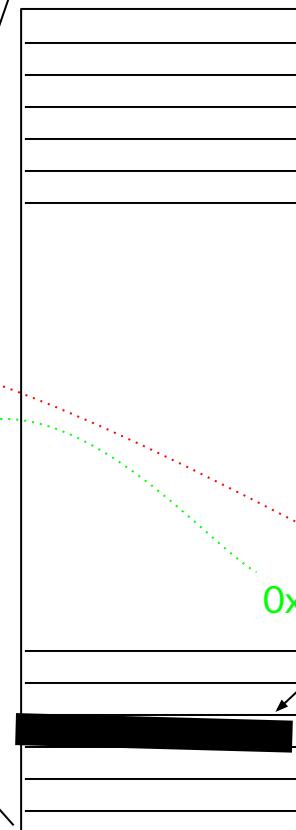
32-bit virtual address
0x00403004
0b00000000010000000011 0x004

Page base addr => 20 bit addr & last
12 bits (12 bit aligned)

Base Address
0x00054000



Which location in page



0x00312000

setupkvm()

kmap - map table/array of 4 kernel regions

~~Pgdir = kmalloc() //malloc top level dir\\
//alloc 1 page for outer directory~~
for loop iterate over each entry in **kap**
call **mappages()** routine to build page
table foreach kernel segment

mappages()

For a given kmap region walk (using **walkpgdir()** outer and inner page tables and populate page table entries

- Page table entries added by “mappages”
 - Arguments: page directory, range of virtual addresses, physical addresses to map to, permissions of the pages
 - For each page, walks page table, get pointer to PTE via function “walkpgdir”, fills it with physical address and permissions
 - Function “walkpgdir” walks page table, returns PTE of a virtual address
 - Can allocate inner page table if it doesn’t exist

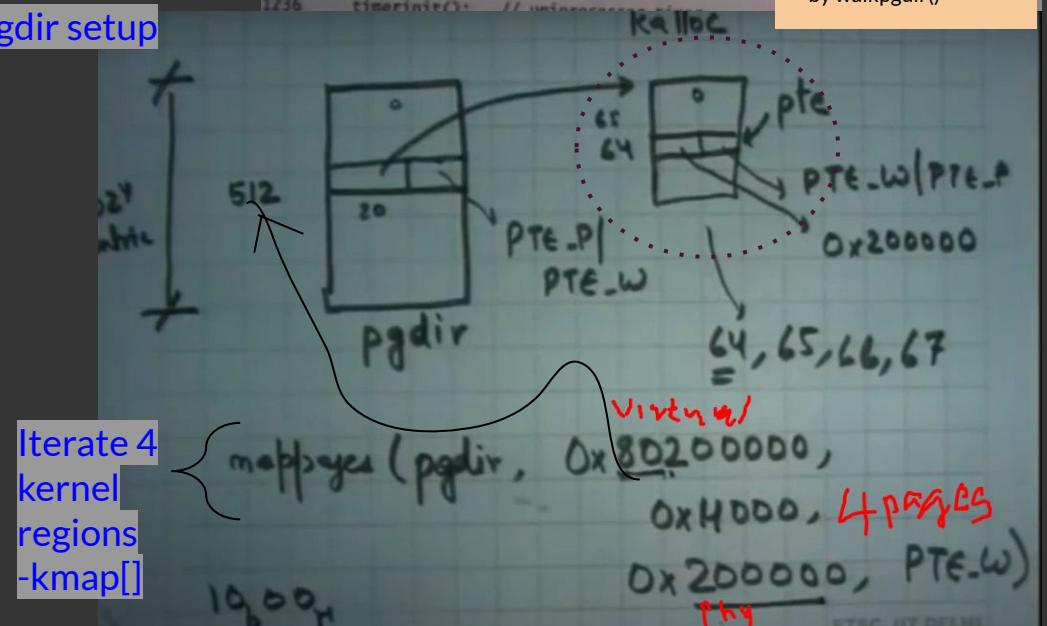
Add entries in outer page table & call walkpgdir to build for alloc/build inner TBL

```
1757 // Physical address at pa. va and size might not
1758 // be page-aligned.
1759 static int
1760 mappings(pde_t *pgdir, void *va, uint size, uint pa, int perm)
1761 {
1762     char *a, *last;
1763     pte_t *pte;
1764
1765     last = (char*)PCROUNDOWN((uint)va);
1766     last -= (char*)PCROUNDOWN(((uint)va) + size - 1);
1767     for(; last >= va; last -= PTE_SIZE)
1768     {
1769         if(!pte_walkpgdir(pgdir, a, 1))
1770             return -1;
1771         if(*pte == PTE_P)
1772             panic("remap");
1773         *pte |= pa | perm | PTE_P;
1774     }
1775     return 0;
1776 }
```

1731 // Return the address of the PTE in page table pdptr
1732 // that corresponds to virtual address va. If alloc==0,
1733 // create any required page table pages.
1734 static pte_t *
1735 pte_walkpgdir(pde_t *pgdir, const void *va, int alloc)
1736 {
1737 pde_t *pde;
1738 pte_t *ptab;
1739 pte_t *pte;
1740
1741 pde = pgdir+PDX(va));
1742 if(*pde & PTE_P)
1743 {
1744 ptab = (pte_t*)PVIPE_ADDR(*pde);
1745 } else
1746 {
1747 if(alloc)
1748 kalloc();
1749 if(!ptab)
1750 return 0;
1751 memset(ptab, 0, PSEZIZE);
1752 // Make sure all these PTE_Bits are zero.
1753 pte = ptab;
1754 // The permissions here are overly generous, but they can
1755 // always be modified by the permissions in the page table
1756 // entries, if necessary.
1757 *pte = PTE_P | PTE_U | PTE_W | PTE_H;

```
1217 main(void)
1218 {
1219     kpgdir = setupkvm();
1220     switchkvm();
1221 }
1222 // Switch h/w page tab
1223 // for when no process
1224 void
1225 switchkvm(void)
1226 {
1227     lcr3(v2p(kpgdir));
1228 }
1229
1230 kinit(end, P2V(4*1024*1024)); // phys page allocator
1231 kmalloc(); // kernel page table
1232 mpinit(); // collect info about this machine
1233 lapicinit();
1234 seginit(); // set up segments
1235 cprintf("ncpu%d: starting xv6\n\n", cpu->id);
1236 picinit(); // interrupt controller
1237 ioapicinit(); // another interrupt controller
1238 consoleinit(); // I/O devices & their interrupts
1239 uartinit(); // serial port
1240 pinit(); // process table
1241 tvinit(); // trap vectors
1242 binit(); // buffer cache
1243 fileinit(); // file table
1244 iinit(); // inode cache
1245 ideinit(); // disk
1246 if(ifismp)
1247     timerinit(); // uniprocessor timer
1248 }
```

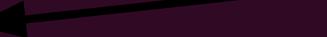
Inner page table entry-
Ether alloc & populate
Or just populate
- by walkpgdir()



```
neeraj.arora@IBS-LAP-095:~/xv6/unix_space/xv6-public$ objdump -f kernel
```

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

entry()



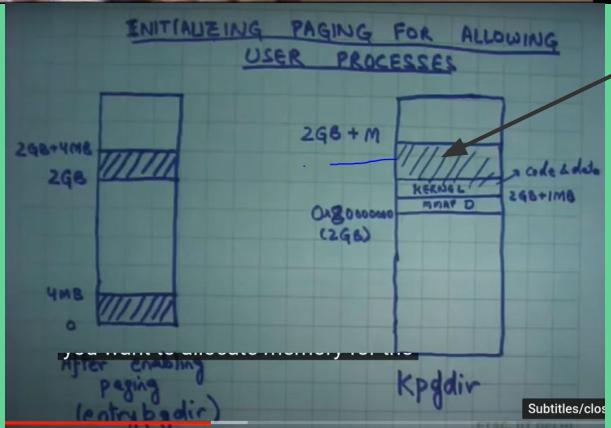
```
neeraj.arora@IBS-LAP-095:~/xv6/unix_space/xv6-public$ objdump -p kernel
```

```
kernel:      file format elf32-i386
```

Program Header:

LOAD	off 0x00001000	vaddr 0x80100000	paddr 0x00100000	align 2**12
	filesz 0x00007aab	memsz 0x00007aab	flags r-x	
LOAD	off 0x00009000	vaddr 0x80108000	paddr 0x00108000	align 2**12
	filesz 0x00002516	memsz 0x0000d4a8	flags rw-	
STACK	off 0x00000000	vaddr 0x00000000	paddr 0x00000000	align 2**4
	filesz 0x00000000	memsz 0x00000000	flags rwx	

Setting up page table in for kernel - changing from entrydir to kernelpgdir

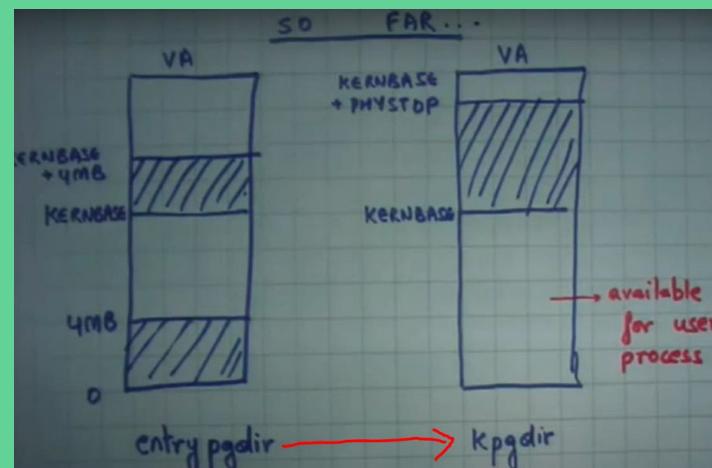


1. Recap of virtual memory subsystem -from bootup till call of main()
 - a. At startup no virtual memory flat address space
 - b. Next setup gdt table, make gdtr point to this & enable segmentation - 32 bit protected only segmentation no paging
 - c. The above segmentation is virtually disabled
 - d. Boot sectors loads kernel starting @1MB
 - e. How to change from entrypgdir regime to kernelpgdir
 - i. Entire phy memory 0 to 234MB is mapped from 2GB+234MB
 - ii. All process control blocks (PCBs, inodes for open files) all kernel data structures are allocated by kernel in this mem area using kalloc()/kernel heap - even when processes are loaded/started physical memory pages required to map process virtual address space are mapped using this kernel heap and process's page table will have that as double mapping (0 -2GB) user segments are 0-2GB - so this leads to duplicate memory mapping - kernel allocator as well as user space process having mapping for same kernel page Converts into physical pointer & loads into CR3
 - iv. When kalloc() can fail - kernel size is big & no headroom till 2GB+4MB
 - v. Kmap - kernel regions that needs mapping - per region one entry

Setting up page table in for kernel - changing from entrydir(4Mb) to kernelpgdir(4k) regime - continued



1. Recap of virtual memory subsystem -from bootup till call of main()
 - a. At startup no virtual memory flat address space
 - b. Next setup gdt table, make gdtr point to this & enable segmentation - 32 bit protected only segmentation no paging
2. Live run through of mappages & walkpgdir combination to construct 2 level page table for kernel virtual memory



Process Management

Process table (ptable) in xv6

```
2409 struct {  
2410     struct spinlock lock;  
2411     struct proc proc[NPROC];  
2412 } ptable;
```



- ptable: Fixed-size array of all processes
 - Real kernels have dynamic-sized data structures
- CPU scheduler in the OS loops over all runnable processes, picks one, and sets it running on the CPU

```
2768     // Loop over process table looking for process to run.  
2769     acquire(&ptable.lock);  
2770     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){  
2771         if(p->state != RUNNABLE)  
2772             continue;  
2773  
2774         // Switch to chosen process.  It is the process's job  
2775         // to release ptable.lock and then reacquire it  
2776         // before jumping back to us.  
2777         c->proc = p;  
2778         switchuvn(p);  
2779         p->state = RUNNING;
```

Process states

Basic structure of PCB

List of processes in ptable[]

Round robin scheduler

State transitions

Anatomy of System Call

[stack overflow.com/questions/29656136/is-there-a-system-call-service-routine-in-the-interrupt-vector](https://stackoverflow.com/questions/29656136/is-there-a-system-call-service-routine-in-the-interrupt-vector)



About Products For Teams

Search...

Home

PUBLIC

Questions

Tags

Users

Companies

COLLECTIVES

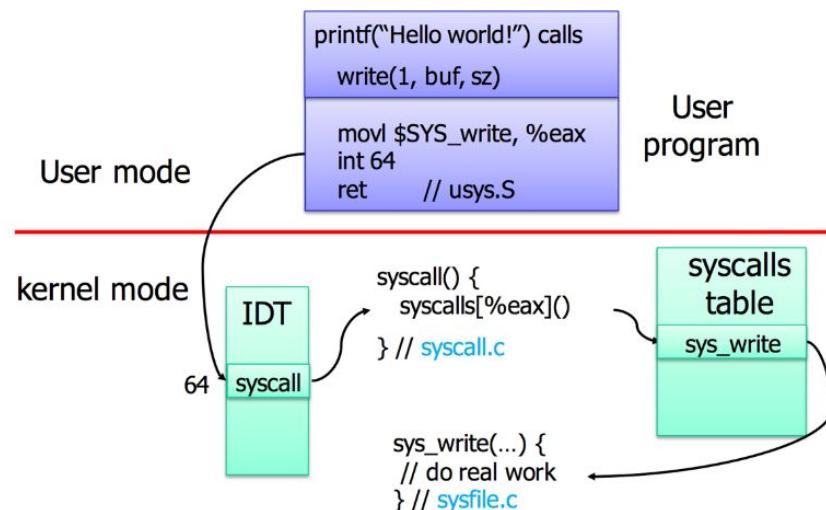
Explore Collectives

TEAMS

Stack Overflow for
Teams – Start
collaborating and
sharing organizational
knowledge.



Are system calls also stored in a vector of function pointers ? Like shown on this picture ?



If so why there is a possibility to add your own system call and there is no possibility to add your own interrupt handler ? Why interrupt-vector is fixed-size and system call vector not ?

Quote from Silberschatz Operating Systems Concepts:

<https://stackoverflow.com/questions/29656136/is-there-a-system-call-service-routine-in-the-interrupt-vector>

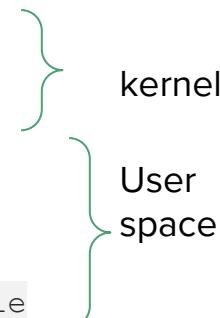
[Adding System call in xv6. what is xv6 ? | by Mahima Kothari | Medium](#)

Typos -

Make 2 changes -
EXTRA=\ should have hello.c
added (missed out in blog)

we need to modify following files :

- 1) syscall.c
- 2) syscall.h
- 3) sysproc.c
- 4) usys.S
- 5) user.h
- 6) user prog
- 7) change Makefile



Interrupts, Exceptions & System Calls

System call starting point

- sh.c writing its "\$ " prompt

```
int getcmd(char *buf, int nbuf)
{
    printf(2, "$ ");
    ...
}
```

sh.c

```
int write(int, const void*, int);
...
void printf(int, const char*, ...);
```

user.h

```
static void putc(int fd, char c)
{
    write(fd, &c, 1);
}
```

```
void printf(int fd, const char *fmt, ...)
{
    ...
    putc(fd, c);
}
```

printf.c

```
#define SYSCALL(name) \
.globl name; \
name: \
    movl $SYS_ ## name, %eax; \
    int $T_SYSCALL; \
    ret
...
SYSCALL(write)
```

usys.s

```
#define SYS_write 16
...
syscall.h
```

```
00000cec <write>:
SYSCALL(write)
    cec:          mov    $0x10,%eax
    cf1:          int    $0x40
    cf3:          ret
```

sh.asm

Code walk through

Trap frame on the kernel stack

- Trap frame: state is pushed on kernel stack during trap handling
 - CPU context of where execution stopped is saved, so that it can be resumed after trap
 - Some extra information needed by trap handler is also saved
- The "int n" instruction has so far only pushed the bottom few entries of trap frame
 - The kernel code we are about to see next will push the rest

```
3630 // Layout of the trap frame built on the stack by the
3631 // hardware and by trapsys.c, and passed to trap().
3632 struct trapframe
3633 {
3634     /* registers are mapped by trapsys
3635      * 3635    user: rax;
3636      * 3636    user: rsi;
3637      * 3638    user: rbp; */ // useless & unused
3639      * 3639    user: rdi;
3640      * 3641    user: rdx;
3642      * 3643    user: rcx;
3644      * 3645    user: rbx;
3646      * 3647    user: rbp; */
3647      * 3648    user: padding[2];
3649      * 3650    user: rsi;
3651      * 3652    user: padding[2];
3653      * 3654    user: rdi;
3655      * 3656    user: padding[2];
3656      * 3657    user: rax;
3657      * 3658    user: padding[2];
3658      * 3659    user: rbp; */
3659      * 3660    user: padding[2];
3660      * 3661    user: rsi;
3661      * 3662    user: padding[2];
3662      * 3663    user: rdi;
3663      * 3664    user: padding[2];
3664      * 3665    user: rax;
3665      * 3666    user: padding[2];
3666      * 3667    user: rbp; */
3667      * 3668    user: padding[2];
3668      * 3669    user: rsi;
3669      * 3670    user: padding[2];
3670      * 3671    user: rdi;
3671      * 3672    user: padding[2];
3672      * 3673    user: rax;
3673      * 3674    user: padding[2];
3674      * 3675    user: rbp; */
3675      * 3676    user: padding[2];
3676      * 3677    user: rsi;
3677      * 3678    user: padding[2];
3678      * 3679    user: rdi;
3679      * 3680    user: padding[2];
3680      * 3681    user: rax;
3681      * 3682    user: padding[2];
3682      * 3683    user: rbp; */
3683      * 3684    user: padding[2];
3684      * 3685    user: rsi;
3685      * 3686    user: padding[2];
3686      * 3687    user: rdi;
3687      * 3688    user: padding[2];
3688      * 3689    user: rax;
3689      * 3690    user: padding[2];
3690      * 3691    user: rbp; */
3691      * 3692    user: padding[2];
3692      * 3693    user: rsi;
3693      * 3694    user: padding[2];
3694      * 3695    user: rdi;
3695      * 3696    user: padding[2];
3696      * 3697    user: rax;
3697      * 3698    user: padding[2];
3698      * 3699    user: rbp; */
3699      * 3700    user: padding[2];
3700      * 3701    user: rsi;
3701      * 3702    user: padding[2];
3702      * 3703    user: rdi;
3703      * 3704    user: padding[2];
3704      * 3705    user: rax;
3705      * 3706    user: padding[2];
3706      * 3707    user: rbp; */
3707      * 3708    user: padding[2];
3708      * 3709    user: rsi;
3709      * 3710    user: padding[2];
3710      * 3711    user: rdi;
3711      * 3712    user: padding[2];
3712      * 3713    user: rax;
3713      * 3714    user: padding[2];
3714      * 3715    user: rbp; */
3715      * 3716    user: padding[2];
3716      * 3717    user: rsi;
3717      * 3718    user: padding[2];
3718      * 3719    user: rdi;
3719      * 3720    user: padding[2];
3720      * 3721    user: rax;
3721      * 3722    user: padding[2];
3722      * 3723    user: rbp; */
3723      * 3724    user: padding[2];
3724      * 3725    user: rsi;
3725      * 3726    user: padding[2];
3726      * 3727    user: rdi;
3727      * 3728    user: padding[2];
3728      * 3729    user: rax;
3729      * 3730    user: padding[2];
3730      * 3731    user: rbp; */
3731      * 3732    user: padding[2];
3732      * 3733    user: rsi;
3733      * 3734    user: padding[2];
3734      * 3735    user: rdi;
3735      * 3736    user: padding[2];
3736      * 3737    user: rax;
3737      * 3738    user: padding[2];
3738      * 3739    user: rbp; */
3739      * 3740    user: padding[2];
3740      * 3741    user: rsi;
3741      * 3742    user: padding[2];
3742      * 3743    user: rdi;
3743      * 3744    user: padding[2];
3744      * 3745    user: rax;
3745      * 3746    user: padding[2];
3746      * 3747    user: rbp; */
3747      * 3748    user: padding[2];
3748      * 3749    user: rsi;
3749      * 3750    user: padding[2];
3750      * 3751    user: rdi;
3751      * 3752    user: padding[2];
3752      * 3753    user: rax;
3753      * 3754    user: padding[2];
3754      * 3755    user: rbp; */
3755      * 3756    user: padding[2];
3756      * 3757    user: rsi;
3757      * 3758    user: padding[2];
3758      * 3759    user: rdi;
3759      * 3760    user: padding[2];
3760      * 3761    user: rax;
3761      * 3762    user: padding[2];
3762      * 3763    user: rbp; */
3763      * 3764    user: padding[2];
3764      * 3765    user: rsi;
3765      * 3766    user: padding[2];
3766      * 3767    user: rdi;
3767      * 3768    user: padding[2];
3768      * 3769    user: rax;
3769      * 3770    user: padding[2];
3770      * 3771    user: rbp; */
3771      * 3772    user: padding[2];
3772      * 3773    user: rsi;
3773      * 3774    user: padding[2];
3774      * 3775    user: rdi;
3775      * 3776    user: padding[2];
3776      * 3777    user: rax;
3777      * 3778    user: padding[2];
3778      * 3779    user: rbp; */
3779      * 3780    user: padding[2];
3780      * 3781    user: rsi;
3781      * 3782    user: padding[2];
3782      * 3783    user: rdi;
3783      * 3784    user: padding[2];
3784      * 3785    user: rax;
3785      * 3786    user: padding[2];
3786      * 3787    user: rbp; */
3787      * 3788    user: padding[2];
3788      * 3789    user: rsi;
3789      * 3790    user: padding[2];
3790      * 3791    user: rdi;
3791      * 3792    user: padding[2];
3792      * 3793    user: rax;
3793      * 3794    user: padding[2];
3794      * 3795    user: rbp; */
3795      * 3796    user: padding[2];
3796      * 3797    user: rsi;
3797      * 3798    user: padding[2];
3798      * 3799    user: rdi;
3799      * 3800    user: padding[2];
3800      * 3801    user: rax;
3801      * 3802    user: padding[2];
3802      * 3803    user: rbp; */
3803      * 3804    user: padding[2];
3804      * 3805    user: rsi;
3805      * 3806    user: padding[2];
3806      * 3807    user: rdi;
3807      * 3808    user: padding[2];
3808      * 3809    user: rax;
3809      * 3810    user: padding[2];
3810      * 3811    user: rbp; */
3811      * 3812    user: padding[2];
3812      * 3813    user: rsi;
3813      * 3814    user: padding[2];
3814      * 3815    user: rdi;
3815      * 3816    user: padding[2];
3816      * 3817    user: rax;
3817      * 3818    user: padding[2];
3818      * 3819    user: rbp; */
3819      * 3820    user: padding[2];
3820      * 3821    user: rsi;
3821      * 3822    user: padding[2];
3822      * 3823    user: rdi;
3823      * 3824    user: padding[2];
3824      * 3825    user: rax;
3825      * 3826    user: padding[2];
3826      * 3827    user: rbp; */
3827      * 3828    user: padding[2];
3828      * 3829    user: rsi;
3829      * 3830    user: padding[2];
3830      * 3831    user: rdi;
3831      * 3832    user: padding[2];
3832      * 3833    user: rax;
3833      * 3834    user: padding[2];
3834      * 3835    user: rbp; */
3835      * 3836    user: padding[2];
3836      * 3837    user: rsi;
3837      * 3838    user: padding[2];
3838      * 3839    user: rdi;
3839      * 3840    user: padding[2];
3840      * 3841    user: rax;
3841      * 3842    user: padding[2];
3842      * 3843    user: rbp; */
3843      * 3844    user: padding[2];
3844      * 3845    user: rsi;
3845      * 3846    user: padding[2];
3846      * 3847    user: rdi;
3847      * 3848    user: padding[2];
3848      * 3849    user: rax;
3849      * 3850    user: padding[2];
3850      * 3851    user: rbp; */
3851      * 3852    user: padding[2];
3852      * 3853    user: rsi;
3853      * 3854    user: padding[2];
3854      * 3855    user: rdi;
3855      * 3856    user: padding[2];
3856      * 3857    user: rax;
3857      * 3858    user: padding[2];
3858      * 3859    user: rbp; */
3859      * 3860    user: padding[2];
3860      * 3861    user: rsi;
3861      * 3862    user: padding[2];
3862      * 3863    user: rdi;
3863      * 3864    user: padding[2];
3864      * 3865    user: rax;
3865      * 3866    user: padding[2];
3866      * 3867    user: rbp; */
3867      * 3868    user: padding[2];
3868      * 3869    user: rsi;
3869      * 3870    user: padding[2];
3870      * 3871    user: rdi;
3871      * 3872    user: padding[2];
3872      * 3873    user: rax;
3873      * 3874    user: padding[2];
3874      * 3875    user: rbp; */
3875      * 3876    user: padding[2];
3876      * 3877    user: rsi;
3877      * 3878    user: padding[2];
3878      * 3879    user: rdi;
3879      * 3880    user: padding[2];
3880      * 3881    user: rax;
3881      * 3882    user: padding[2];
3882      * 3883    user: rbp; */
3883      * 3884    user: padding[2];
3884      * 3885    user: rsi;
3885      * 3886    user: padding[2];
3886      * 3887    user: rdi;
3887      * 3888    user: padding[2];
3888      * 3889    user: rax;
3889      * 3890    user: padding[2];
3890      * 3891    user: rbp; */
3891      * 3892    user: padding[2];
3892      * 3893    user: rsi;
3893      * 3894    user: padding[2];
3894      * 3895    user: rdi;
3895      * 3896    user: padding[2];
3896      * 3897    user: rax;
3897      * 3898    user: padding[2];
3898      * 3899    user: rbp; */
3899      * 3900    user: padding[2];
3900      * 3901    user: rsi;
3901      * 3902    user: padding[2];
3902      * 3903    user: rdi;
3903      * 3904    user: padding[2];
3904      * 3905    user: rax;
3905      * 3906    user: padding[2];
3906      * 3907    user: rbp; */
3907      * 3908    user: padding[2];
3908      * 3909    user: rsi;
3909      * 3910    user: padding[2];
3910      * 3911    user: rdi;
3911      * 3912    user: padding[2];
3912      * 3913    user: rax;
3913      * 3914    user: padding[2];
3914      * 3915    user: rbp; */
3915      * 3916    user: padding[2];
3916      * 3917    user: rsi;
3917      * 3918    user: padding[2];
3918      * 3919    user: rdi;
3919      * 3920    user: padding[2];
3920      * 3921    user: rax;
3921      * 3922    user: padding[2];
3922      * 3923    user: rbp; */
3923      * 3924    user: padding[2];
3924      * 3925    user: rsi;
3925      * 3926    user: padding[2];
3926      * 3927    user: rdi;
3927      * 3928    user: padding[2];
3928      * 3929    user: rax;
3929      * 3930    user: padding[2];
3930      * 3931    user: rbp; */
3931      * 3932    user: padding[2];
3932      * 3933    user: rsi;
3933      * 3934    user: padding[2];
3934      * 3935    user: rdi;
3935      * 3936    user: padding[2];
3936      * 3937    user: rax;
3937      * 3938    user: padding[2];
3938      * 3939    user: rbp; */
3939      * 3940    user: padding[2];
3940      * 3941    user: rsi;
3941      * 3942    user: padding[2];
3942      * 3943    user: rdi;
3943      * 3944    user: padding[2];
3944      * 3945    user: rax;
3945      * 3946    user: padding[2];
3946      * 3947    user: rbp; */
3947      * 3948    user: padding[2];
3948      * 3949    user: rsi;
3949      * 3950    user: padding[2];
3950      * 3951    user: rdi;
3951      * 3952    user: padding[2];
3952      * 3953    user: rax;
3953      * 3954    user: padding[2];
3954      * 3955    user: rbp; */
3955      * 3956    user: padding[2];
3956      * 3957    user: rsi;
3957      * 3958    user: padding[2];
3958      * 3959    user: rdi;
3959      * 3960    user: padding[2];
3960      * 3961    user: rax;
3961      * 3962    user: padding[2];
3962      * 3963    user: rbp; */
3963      * 3964    user: padding[2];
3964      * 3965    user: rsi;
3965      * 3966    user: padding[2];
3966      * 3967    user: rdi;
3967      * 3968    user: padding[2];
3968      * 3969    user: rax;
3969      * 3970    user: padding[2];
3970      * 3971    user: rbp; */
3971      * 3972    user: padding[2];
3972      * 3973    user: rsi;
3973      * 3974    user: padding[2];
3974      * 3975    user: rdi;
3975      * 3976    user: padding[2];
3976      * 3977    user: rax;
3977      * 3978    user: padding[2];
3978      * 3979    user: rbp; */
3979      * 3980    user: padding[2];
3980      * 3981    user: rsi;
3981      * 3982    user: padding[2];
3982      * 3983    user: rdi;
3983      * 3984    user: padding[2];
3984      * 3985    user: rax;
3985      * 3986    user: padding[2];
3986      * 3987    user: rbp; */
3987      * 3988    user: padding[2];
3988      * 3989    user: rsi;
3989      * 3990    user: padding[2];
3990      * 3991    user: rdi;
3991      * 3992    user: padding[2];
3992      * 3993    user: rax;
3993      * 3994    user: padding[2];
3994      * 3995    user: rbp; */
3995      * 3996    user: padding[2];
3996      * 3997    user: rsi;
3997      * 3998    user: padding[2];
3998      * 3999    user: rdi;
3999      * 4000    user: padding[2];
4000      * 4001    user: rax;
4001      * 4002    user: padding[2];
4002      * 4003    user: rbp; */
4003      * 4004    user: padding[2];
4004      * 4005    user: rsi;
4005      * 4006    user: padding[2];
4006      * 4007    user: rdi;
4007      * 4008    user: padding[2];
4008      * 4009    user: rax;
4009      * 4010    user: padding[2];
4010      * 4011    user: rbp; */
4011      * 4012    user: padding[2];
4012      * 4013    user: rsi;
4013      * 4014    user: padding[2];
4014      * 4015    user: rdi;
4015      * 4016    user: padding[2];
4016      * 4017    user: rax;
4017      * 4018    user: padding[2];
4018      * 4019    user: rbp; */
4019      * 4020    user: padding[2];
4020      * 4021    user: rsi;
4021      * 4022    user: padding[2];
4022      * 4023    user: rdi;
4023      * 4024    user: padding[2];
4024      * 4025    user: rax;
4025      * 4026    user: padding[2];
4026      * 4027    user: rbp; */
4027      * 4028    user: padding[2];
4028      * 4029    user: rsi;
4029      * 4030    user: padding[2];
4030      * 4031    user: rdi;
4031      * 4032    user: padding[2];
4032      * 4033    user: rax;
4033      * 4034    user: padding[2];
4034      * 4035    user: rbp; */
4035      * 4036    user: padding[2];
4036      * 4037    user: rsi;
4037      * 4038    user: padding[2];
4038      * 4039    user: rdi;
4039      * 4040    user: padding[2];
4040      * 4041    user: rax;
4041      * 4042    user: padding[2];
4042      * 4043    user: rbp; */
4043      * 4044    user: padding[2];
4044      * 4045    user: rsi;
4045      * 4046    user: padding[2];
4046      * 4047    user: rdi;
4047      * 4048    user: padding[2];
4048      * 4049    user: rax;
4049      * 4050    user: padding[2];
4050      * 4051    user: rbp; */
4051      * 4052    user: padding[2];
4052      * 4053    user: rsi;
4053      * 4054    user: padding[2];
4054      * 4055    user: rdi;
4055      * 4056    user: padding[2];
4056      * 4057    user: rax;
4057      * 4058    user: padding[2];
4058      * 4059    user: rbp; */
4059      * 4060    user: padding[2];
4060      * 4061    user: rsi;
4061      * 4062    user: padding[2];
4062      * 4063    user: rdi;
4063      * 4064    user: padding[2];
4064      * 4065    user: rax;
4065      * 4066    user: padding[2];
4066      * 4067    user: rbp; */
4067      * 4068    user: padding[2];
4068      * 4069    user: rsi;
4069      * 4070    user: padding[2];
4070      * 4071    user: rdi;
4071      * 4072    user: padding[2];
4072      * 4073    user: rax;
4073      * 4074    user: padding[2];
4074      * 4075    user: rbp; */
4075      * 4076    user: padding[2];
4076      * 4077    user: rsi;
4077      * 4078    user: padding[2];
4078      * 4079    user: rdi;
4079      * 4080    user: padding[2];
4080      * 4081    user: rax;
4081      * 4082    user: padding[2];
4082      * 4083    user: rbp; */
4083      * 4084    user: padding[2];
4084      * 4085    user: rsi;
4085      * 4086    user: padding[2];
4086      * 4087    user: rdi;
4087      * 4088    user: padding[2];
4088      * 4089    user: rax;
4089      * 4090    user: padding[2];
4090      * 4091    user: rbp; */
4091      * 4092    user: padding[2];
4092      * 4093    user: rsi;
4093      * 4094    user: padding[2];
4094      * 4095    user: rdi;
4095      * 4096    user: padding[2];
4096      * 4097    user: rax;
4097      * 4098    user: padding[2];
4098      * 4099    user: rbp; */
4099      * 4100    user: padding[2];
4100      * 4101    user: rsi;
4101      * 4102    user: padding[2];
4102      * 4103    user: rdi;
4103      * 4104    user: padding[2];
4104      * 4105    user: rax;
4105      * 4106    user: padding[2];
4106      * 4107    user: rbp; */
4107      * 4108    user: padding[2];
4108      * 4109    user: rsi;
4109      * 4110    user: padding[2];
4110      * 4111    user: rdi;
4111      * 4112    user: padding[2];
4112      * 4113    user: rax;
4113      * 4114    user: padding[2];
4114      * 4115    user: rbp; */
4115      * 4116    user: padding[2];
4116      * 4117    user: rsi;
4117      * 4118    user: padding[2];
4118      * 4119    user: rdi;
4119      * 4120    user: padding[2];
4120      * 4121    user: rax;
4121      * 4122    user: padding[2];
4122      * 4123    user: rbp; */
4123      * 4124    user: padding[2];
4124      * 4125    user: rsi;
4125      * 4126    user: padding[2];
4126      * 4127    user: rdi;
4127      * 4128    user: padding[2];
4128      * 4129    user: rax;
4129      * 4130    user: padding[2];
4130      * 4131    user: rbp; */
4131      * 4132    user: padding[2];
4132      * 4133    user: rsi;
4133      * 4134    user: padding[2];
4134      * 4135    user: rdi;
4135      * 4136    user: padding[2];
4136      * 4137    user: rax;
4137      * 4138    user: padding[2];
4138      * 4139    user: rbp; */
4139      * 4140    user: padding[2];
4140      * 4141    user: rsi;
4141      * 4142    user: padding[2];
4142      * 4143    user: rdi;
4143      * 4144    user: padding[2];
4144      * 4145    user: rax;
4145      * 4146    user: padding[2];
4146      * 4147    user: rbp; */
4147      * 4148    user: padding[2];
4148      * 4149    user: rsi;
4149      * 4150    user: padding[2];
4150      * 4151    user: rdi;
4151      * 4152    user: padding[2];
4152      * 4153    user: rax;
4153      * 4154    user: padding[2];
4154      * 4155    user: rbp; */
4155      * 4156    user: padding[2];
4156      * 4157    user: rsi;
4157      * 4158    user: padding[2];
4158      * 4159    user: rdi;
4159      * 4160    user: padding[2];
4160      * 4161    user: rax;
4161      * 4162    user: padding[2];
4162      * 4163    user: rbp; */
4163      * 4164    user: padding[2];
4164      * 4165    user: rsi;
4165      * 4166    user: padding[2];
4166      * 4167    user: rdi;
4167      * 4168    user: padding[2];
4168      * 4169    user: rax;
4169      * 4170    user: padding[2];
4170      * 4171    user: rbp; */
4171      * 4172    user: padding[2];
4172      * 4173    user: rsi;
4173      * 4174    user: padding[2];
4174      * 4175    user: rdi;
4175      * 4176    user: padding[2];
4176      * 4177    user: rax;
4177      * 4178    user: padding[2];
4178      * 4179    user: rbp; */
4179      * 4180    user: padding[2];
4180      * 4181    user: rsi;
4181      * 4182    user: padding[2];
4182      * 4183    user: rdi;
4183      * 4184    user: padding[2];
4184      * 4185    user: rax;
4185      * 4186    user: padding[2];
4186      * 4187    user: rbp; */
4187      * 4188    user: padding[2];
4188      * 4189    user: rsi;
4189      * 4190    user: padding[2];
4190      * 4191    user: rdi;
4191      * 4192    user: padding[2];
4192      * 4193    user: rax;
4193      * 4194    user: padding[2];
4194      * 4195    user: rbp; */
4195      * 4196    user: padding[2];
4196      * 4197    user: rsi;
4197      * 4198    user: padding[2];
4198      * 4199    user: rdi;
4199      * 4200    user: padding[2];
4200      * 4201    user: rax;
4201      * 4202    user: padding[2];
4202      * 4203    user: rbp; */
4203      * 4204    user: padding[2];
4204      * 4205    user: rsi;
4205      * 4206    user: padding[2];
4206      * 4207    user: rdi;
4207      * 4208    user: padding[2];
4208      * 4209    user: rax;
4209      * 4210    user: padding[2];
4210      * 4211    user: rbp; */
4211      * 4212    user: padding[2];
4212      * 4213    user: rsi;
4213      * 4214    user: padding[2];
4214      * 4215    user: rdi;
4215      * 4216    user: padding[2];
4216      * 4217    user: rax;
4217      * 4218    user: padding[2];
4218      * 4219    user: rbp; */
4219      * 4220    user: padding[2];
4220      * 4221    user: rsi;
4221      * 4222    user: padding[2];
4222      * 4223    user: rdi;
4223      * 4224    user: padding[2];
4224      * 4225    user: rax;
4225      * 4226    user: padding[2];
4226      * 4227    user: rbp; */
4227      * 4228    user: padding[2];
4228      * 4229    user: rsi;
4229      * 4230    user: padding[2];
4230      * 4231    user: rdi;
4231      * 4232    user: padding[2];
4232      * 4233    user: rax;
4233      * 4234    user: padding[2];
4234      * 4235    user: rbp; */
4235      * 4236    user: padding[2];
4236      * 4237    user: rsi;
4237      * 4238    user: padding[2];
4238      * 4239    user: rdi;
4239      * 4240    user: padding[2];
4240      * 4241    user: rax;
4241      * 4242    user: padding[2];
4242      * 4243    user: rbp; */
4243      * 4244    user: padding[2];
4244      * 4245    user: rsi;
4245      * 4246    user: padding[2];
4246      * 4247    user: rdi;
4247      * 4248    user: padding[2];
4248      * 4249    user: rax;
4249      * 4250    user: padding[2];
4250      * 4251    user: rbp; */
4251      * 4252    user: padding[2];
4252      * 4253    user: rsi;
4253      * 4254    user: padding[2];
4254      * 4255    user: rdi;
4255      * 4256    user: padding[2];
4256      * 4257    user: rax;
4257      * 4258    user: padding[2];
4258      * 4259    user: rbp; */
4259      * 4260    user: padding[2];
4260      * 4261    user: rsi;
4261      * 4262    user: padding[2];
4262      * 4263    user: rdi;
4263      * 4264    user: padding[2];
4264      * 4265    user: rax;
4265      * 4266    user: padding[2];
4266      * 4267    user: rbp; */
4267      * 4268    user: padding[2];
4268      * 4269    user: rsi;
4269      * 4270    user: padding[2];
4270      * 4271    user: rdi;
4271      * 4272    user: padding[2];
4272      * 4273    user: rax;
4273      * 4274    user: padding[2];
4274      * 4275    user: rbp; */
4275      * 4276    user: padding[2];
4276      * 4277    user: rsi;
4277      * 4278    user: padding[2];
4278      * 4279    user: rdi;
4279      * 4280    user: padding[2];
4280      * 4281    user: rax;
4281      * 4282    user: padding[2];
4282      * 4283    user: rbp; */
4283      * 4284    user: padding[2];
4284      * 4285    user: rsi;
4285      * 4286    user: padding[2];
4286      * 4287    user: rdi;
4287      * 4288    user: padding[2];
4288      * 4289    user: rax;
4289      * 4290    user: padding[2];
4290      * 4291    user: rbp; */
4291      * 4292    user: padding[2];
4292      * 4293    user: rsi;
4293      * 4294    user: padding[2];
4294      * 4295    user: rdi;
4295      * 4296    user: padding[2];
4296      * 4297    user: rax;
4297      * 4298    user: padding[2];
4298      * 4299    user: rbp; */
4299      * 4300    user: padding[2];
4300      * 4301    user: rsi;
4301      * 4302    user: padding[2];
4302      * 4303    user: rdi;
4303      * 4304    user: padding[2];
4304      * 4305    user: rax;
4305      * 4306    user: padding[2];
4306      * 4307    user: rbp; */
4307      * 4308    user: padding[2];
4308      * 4309    user: rsi;
4309      * 4310    user: padding[2];
4310      * 4311    user: rdi;
4311      * 4312    user: padding[2];
4312      * 4313    user: rax;
4313      * 4314    user: padding[2];
4314      * 4315    user: rbp; */
4315      * 4316    user: padding[2];
4316      * 4317    user: rsi;
4317      * 4318    user: padding[2];
4318      * 4319    user: rdi;
4319      * 4320    user: padding[2];
4320      * 4321    user: rax;
4321      * 4322    user: padding[2];
4322      * 4323    user: rbp; */
4323      * 4324    user: padding[2];
4324      * 4325    user: rsi;
4325      * 4326    user: padding[2];
4326      * 4327    user: rdi;
4327      * 4328    user: padding[2
```

Implementing PS & nice

```
user@neeraj-virtual-machine:~/xv6-jan-22-new-syscall/pub_xv6$ vi user.h
user@neeraj-virtual-machine:~/xv6-jan-22-new-syscall/pub_xv6$ vi proc.c
user@neeraj-virtual-machine:~/xv6-jan-22-new-syscall/pub_xv6$ vi proc.c
user@neeraj-virtual-machine:~/xv6-jan-22-new-syscall/pub_xv6$ vi sysproc.c
user@neeraj-virtual-machine:~/xv6-jan-22-new-syscall/pub_xv6$ vi sysproc.c
user@neeraj-virtual-machine:~/xv6-jan-22-new-syscall/pub_xv6$ vi usys.S
user@neeraj-virtual-machine:~/xv6-jan-22-new-syscall/pub_xv6$ vi sysproc.c
user@neeraj-virtual-machine:~/xv6-jan-22-new-syscall/pub_xv6$ vi syscall.c
user@neeraj-virtual-machine:~/xv6-jan-22-new-syscall/pub_xv6$ vi ps.c
user@neeraj-virtual-machine:~/xv6-jan-22-new-syscall/pub_xv6$ vi nice.c
user@neeraj-virtual-machine:~/xv6-jan-22-new-syscall/pub_xv6$ vi proc.c
user@neeraj-virtual-machine:~/xv6-jan-22-new-syscall/pub_xv6$ vi exec.c
user@neeraj-virtual-machine:~/xv6-jan-22-new-syscall/pub_xv6$ vi dproc.c
user@neeraj-virtual-machine:~/xv6-jan-22-new-syscall/pub_xv6$ vi Makefile
user@neeraj-virtual-machine:~/xv6-jan-22-new-syscall/pub_xv6$ vi Makefile
user@neeraj-virtual-machine:~/xv6-jan-22-new-syscall/pub_xv6$ vi proc.c
user@neeraj-virtual-machine:~/xv6-jan-22-new-syscall/pub_xv6$ █
```

[xv6 -Implementing ps, nice system calls and priority scheduling | by Harshal Shree | Medium'](#)

3 typos -

1. users.h should be read as user.h
2. Function ptr table is in syscall.c (not sysproc.c)
- 3.

https://medium.com/@flag_seeker/xv6-system-calls-how-it-works-c541408f21ff

Day 7 & 8

- Build linux kernel for gdb support & add hello syscall
- Linux kernel & rootfs
 - If PC or server - file system with RAID/encryption and full blown system manager -systemd to bring up desired file system with all demons etc
 - If embedded then perhaps application specific small rootfs with say busybox like command support
- Process management
- System calls in xv6- hello world homework?
- xv6 -Implementing ps, nice system calls and priority scheduling
| by Harshal Shree | Medium

Interesting facts & taking deeper dive

Pune professor makefile analysis bootblock (2 parts bootloader.s and bootmain.c)

Assembly code walkthrough and use the video clipping to explains GDT (switching to protected mode)

stackoverflow.com/questions/17452664/difference-between-real-mode-and-protected-mode-on-x64-architecture?rq=1

10:24

Home Add a comment

SORTED BY: Highest score (default) ▾

1 Answer

see osdev [real mode](#), [protected mode](#).

2 A CPU that is initialized by the BIOS starts in Real Mode. Enabling Protected Mode allows use of all 4GB memory that cannot be accessed by real mode. However, it will prevent you from using most of the BIOS interrupts, since these work in Real Mode (unless you have also written a V86 monitor).

Before switching to Protected Mode, you have to disable interrupts, possibly enable the A20 Line, and load the Global Descriptor Table with segment descriptors suitable for code, data, and stack.

Whether the CPU is in Real Mode or in Protected Mode is defined by the lowest bit of the CR0 or MSW register.

This example loads a descriptor table into the processor's GDTR register, and then sets the lowest bit of CR0:

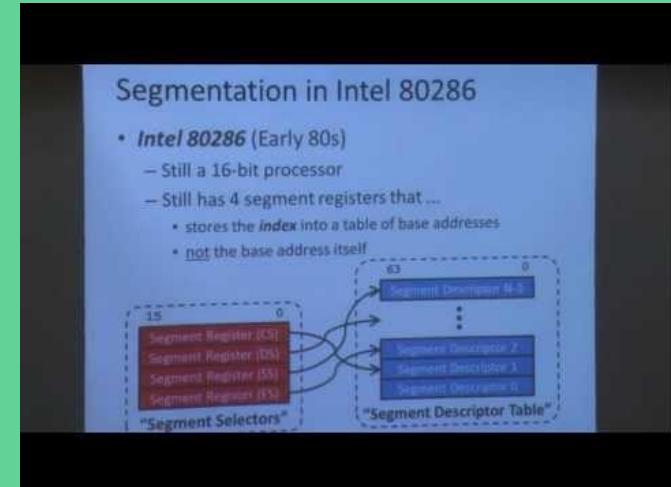
```
cli          ; disable interrupts
lgdt [gdtr]  ; load GDTR register with start address of Global Descriptor Table
mov eax, cr0
or al, 1     ; set PE (Protection Enable) bit in CR0 (Control Register 0)
mov cr0, eax

; Perform far jump to selector 08h (offset into GDT, pointing at a 32bit PM code segment descriptor)
; to load CS with proper PM32 descriptor

JMP 08h:PModeMain
; [...]
```

PModeMain:

A good description of
segment descriptor
GDTR Register/swing
to Protected mode



[difference between real mode and protected mode on x64 architecture - Stack Overflow](https://stackoverflow.com/questions/17452664/difference-between-real-mode-and-protected-mode-on-x64-architecture)

Interesting facts & taking deeper dive -UEFI vs BIOS



1. Computer is like a box with lot of h/w components- storage devices, GPU..
2. Processor needs to be told what to do -this is done by OS -paradox OS is on disk that is powered off, how it is going to be told what is done by whom
3. BIOS is like mini OS on chip available at power on- that initializes HDD and other h/w
4. BIOS obtains configuration, does Power On Self test (POST) - initialize all h/w components one by one -checking if all are working properly
5. Lot of limitations -e.g. very poor user interface, UEFI can be 32 or 64 thus can address way more memory, larger >2TB drives, n/w, nicer user interface support for mouse etc.



BIOS	vs	UEFI
Max partition size in MBR is 2TB		Max partition size in UEFI is 9TB
Max 4 Primary Partition		Max 128 Primary Partition
MBR can store only one bootloader		Separate EFI System Partition (ESP) for storing multiple bootloaders
Non-Secure at Boot time		Secure at Boot time, prevent virus

Interesting facts & taking deeper dive

Real mode vs Protected Mode- why bootloader should run in protected mode - if we assume boot chain of BIOS loads MBR, MBR loads GRUB2, GRUB2 loads kernel, given size of linux kernel is 12Mb so GRUB2 has to switch to protected mode (atleast or may be 64 bit mode?) to address 4G or higher mem to load kernel

```
neeraj@neeraj-virtual-machine: ~/xv6-nov-22/pub_xv6
neeraj@neeraj-virtual-machine:~/xv6-nov-22/pub_xv6$ ls -lh /boot/vmlinuz-5.15.0-52-generic
-rw----- 1 root root 12M Oct 13 13:19 /boot/vmlinuz-5.15.0-52-generic
neeraj@neeraj-virtual-machine:~/xv6-nov-22/pub_xv6$
neeraj@neeraj-virtual-machine:~/xv6-nov-22/pub_xv6$
neeraj@neeraj-virtual-machine:~/xv6-nov-22/pub_xv6$
neeraj@neeraj-virtual-machine:~/xv6-nov-22/pub_xv6$ ls -lh kernel*
-rwxrwxr-x 1 neeraj neeraj 197K Nov 18 05:36 kernel
```

Interesting facts & taking deeper dive

lock stackoverflow.com/questions/17452664/difference-between-real-mode-and-protected-mode-on-x64-architecture?rq=1

 stackoverflow About Products For Teams Search... Search...

Home PUBLIC Questions Tags Users Companies COLLECTIVES Explore Collectives TEAMS

Stack Overflow for Teams – Start collaborating and sharing organizational knowledge.

Create a free Team Why Teams?

What is the difference between real mode and protected mode on the x64 architecture? I'm trying to make a custom boot-loader for the Linux kernel. How can I enable protected mode in assembly?

4 linux-kernel boot bootloader

Share Improve this question Follow edited Nov 14, 2013 at 4:35 by icktoofay 124k ● 20 ● 244 ● 229 asked Jul 3, 2013 at 13:01 by Neo 51 ● 3

1 Read the [Intel® 64 and IA-32 Architectures Software Developer Manual, Volume 3: System Programming Guide](#) or the [AMD64 Architecture Programmer's Manual Volume 2: System Programming](#). – CL Jul 4, 2013 at 10:24

Add a comment

1 Answer Sorted by: Highest score (default)

see osdev [real mode](#), [protected mode](#).

2 A CPU that is initialized by the BIOS starts in Real Mode. Enabling Protected Mode allows use of all 4GB memory that cannot be accessed by real mode. However, it will prevent you from using most of the BIOS interrupts, since these work in Real Mode (unless you have also written a V86 monitor).

Before switching to Protected Mode, you have to disable interrupts, possibly enable the A20 Line, and load the Global Descriptor Table with segment descriptors suitable for code, data, and stack.

Whether the CPU is in Real Mode or in Protected Mode is defined by the lowest bit of the CR0 or MSW register.

This example loads a descriptor table into the processor's GDTR register, and then sets the lowest bit of CR0:

[difference between real mode and protected mode on x64 architecture](#)
[- Stack Overflow](#)

2023 Winter ECE 353

Advanced Scheduling