Getting started with Linux Internals

Boot Sequence
OS Initialization
Building OS

xv6 to linux

xv6 → PC Linux & ARM 64 Linux

xv6 - Intro, building & adding new commands

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

Familiarity with xv6 codebase & systemcall 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

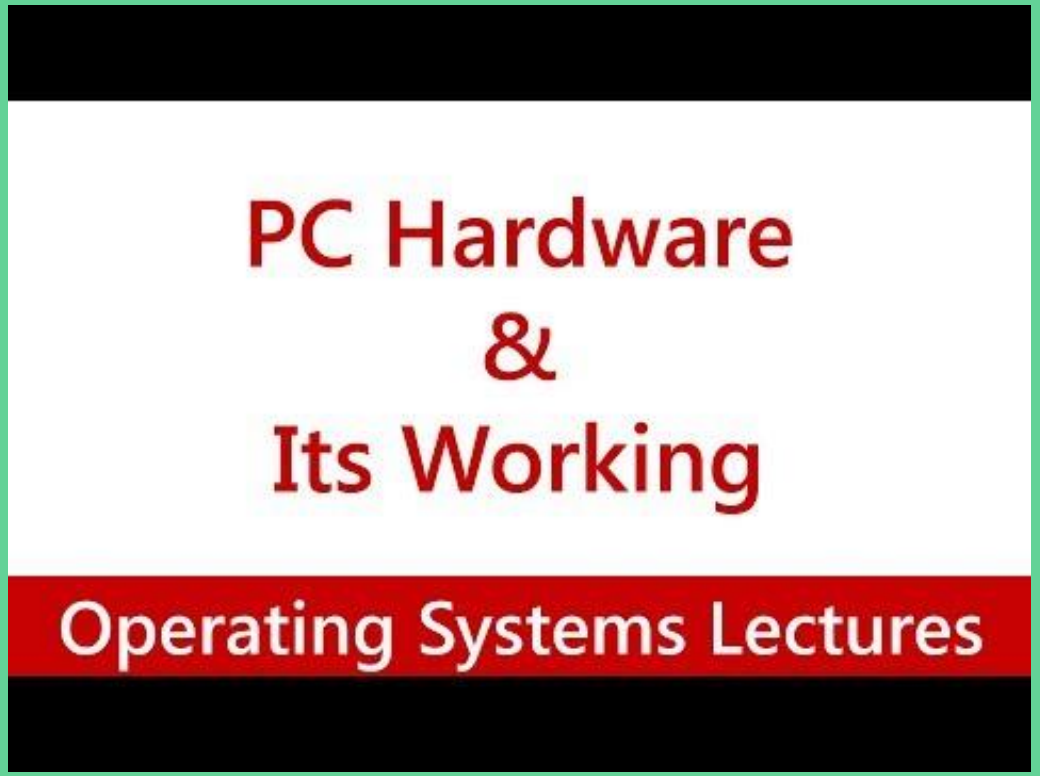
Building minimal linux system -kernel and busybox

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

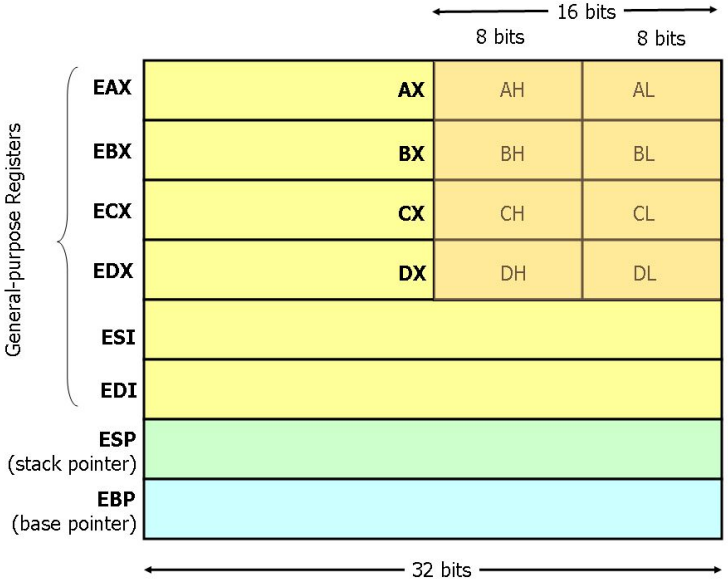
x86 Evolution (@21 mins)

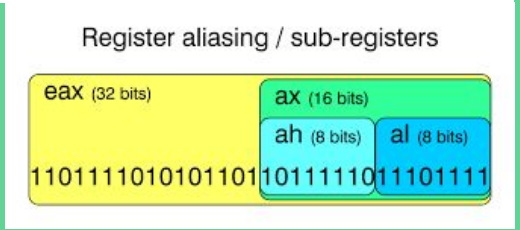


How intel processors have evolved from 16 bit 8088/8086 to 64 bit processors

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 /* 0xaddr 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]

https://web.stanford.edu/class/cs107/resources/x86-64-reference.pdf

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

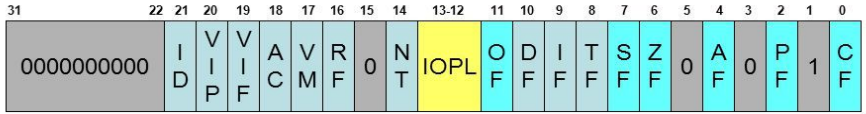
  `and $0x0f, %eax — clear all but the last 4 bits of EAX`

- 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

= 'reserved' bit

= 'status' flag (for application programming)

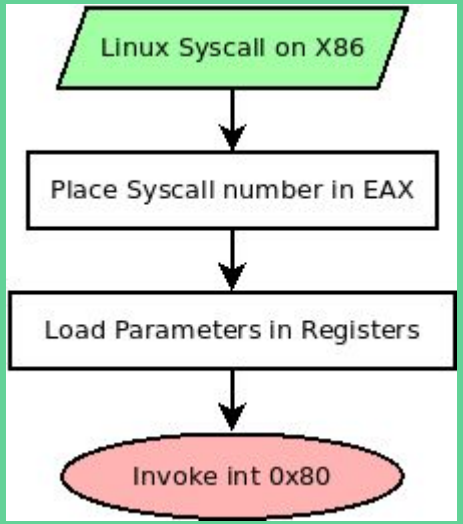
Jump/conditional jump

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**
jge label	**jump >= (signed) SF=OF**
jl label	**jump < (signed) SF!=OF**
jle label	**jump <= (signed) ZF=1 or SF!=OF**
ja label	**jump > (unsigned) CF=0 and ZF=0**
jae label	**jump >= (unsigned) CF=0**
jb label	**jump < (unsigned) CF=1**
jbe label	**jump <= (unsigned) CF=1 or ZF=1**
jz label	**if result of last instruction is 0**

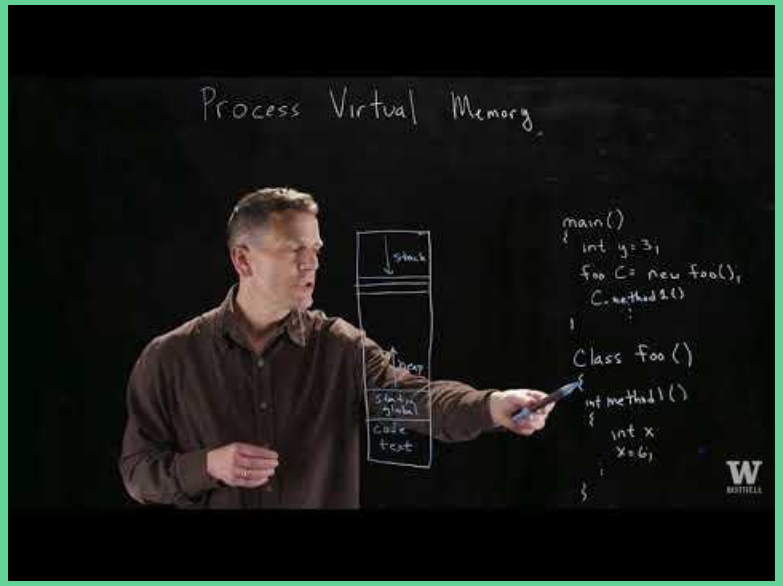
X86 Control Flow & System instructions

Control Flow - jmp, je, jl...
test - no change in operands - just dpes 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

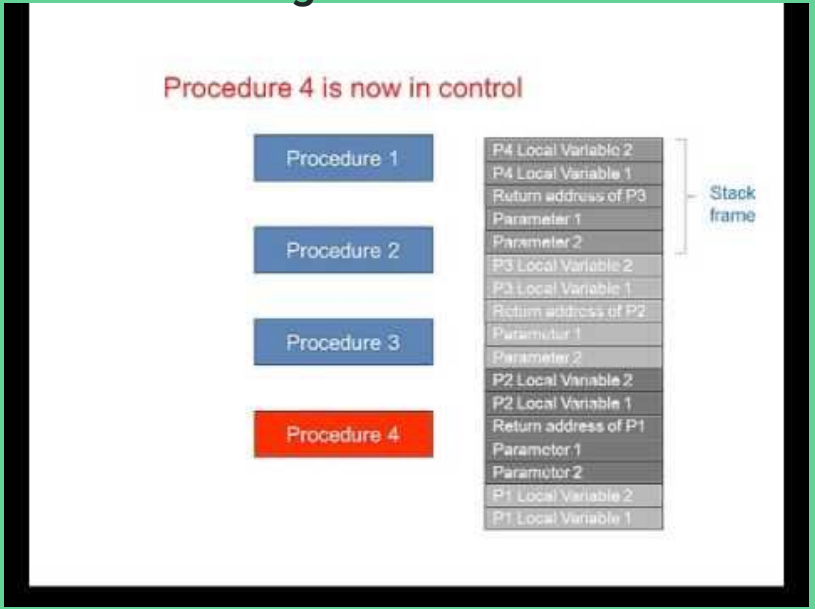


Process VM layout



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

[2. x86 Assembly and Call Stack - Computer Security]

https://gabrieletolomei.wordpress.com/miscellanea/operating-systems/in-memory-layout/

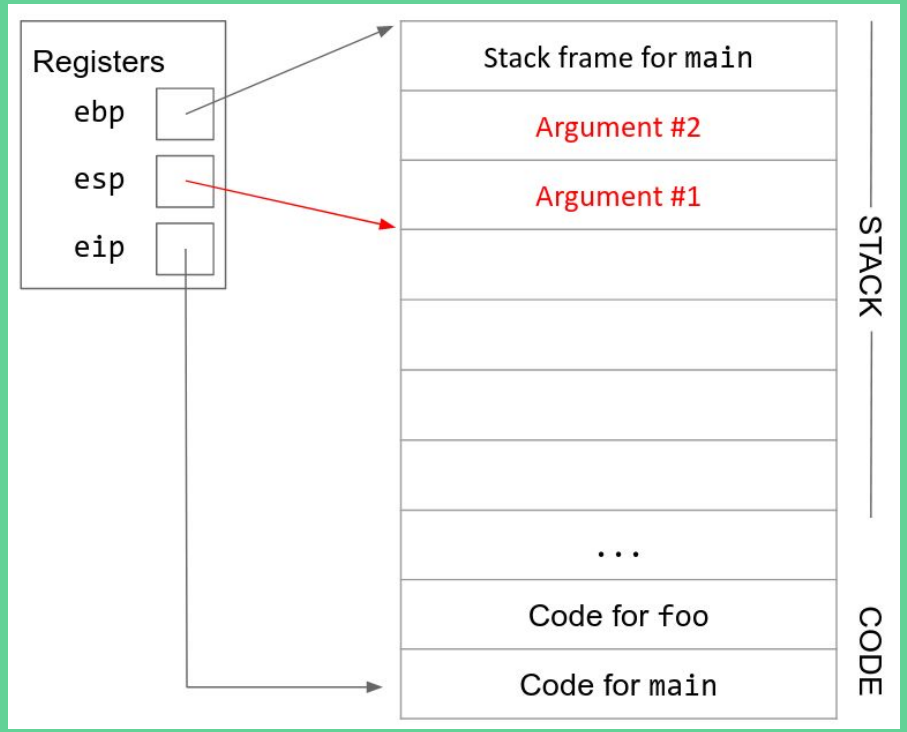
Pushing to stack

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

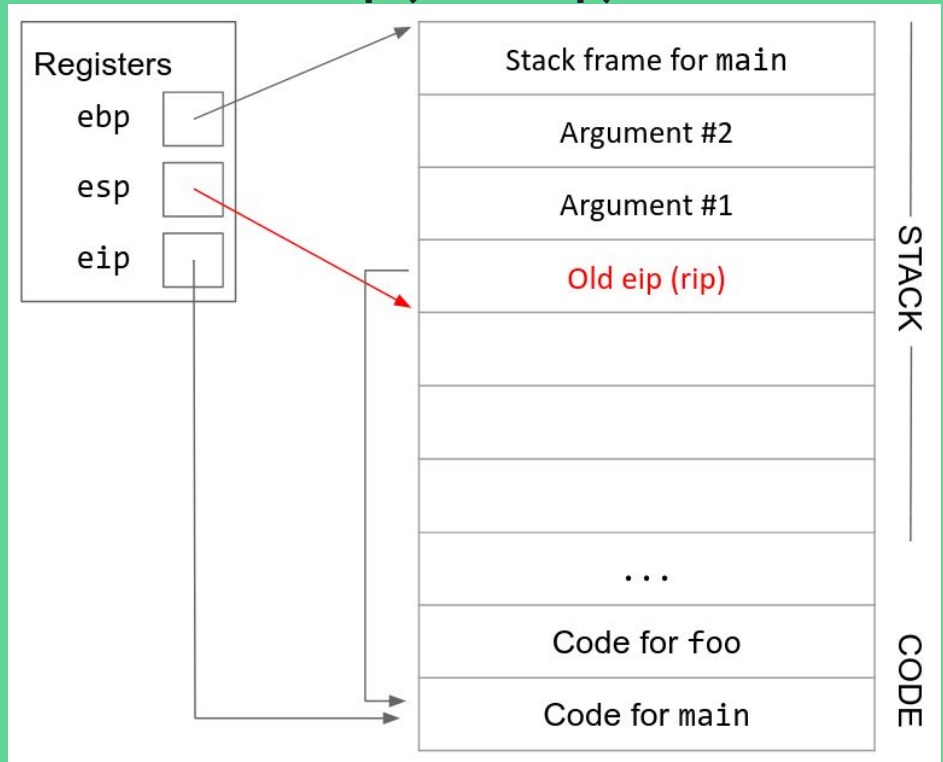
"push" does both of these steps

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

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

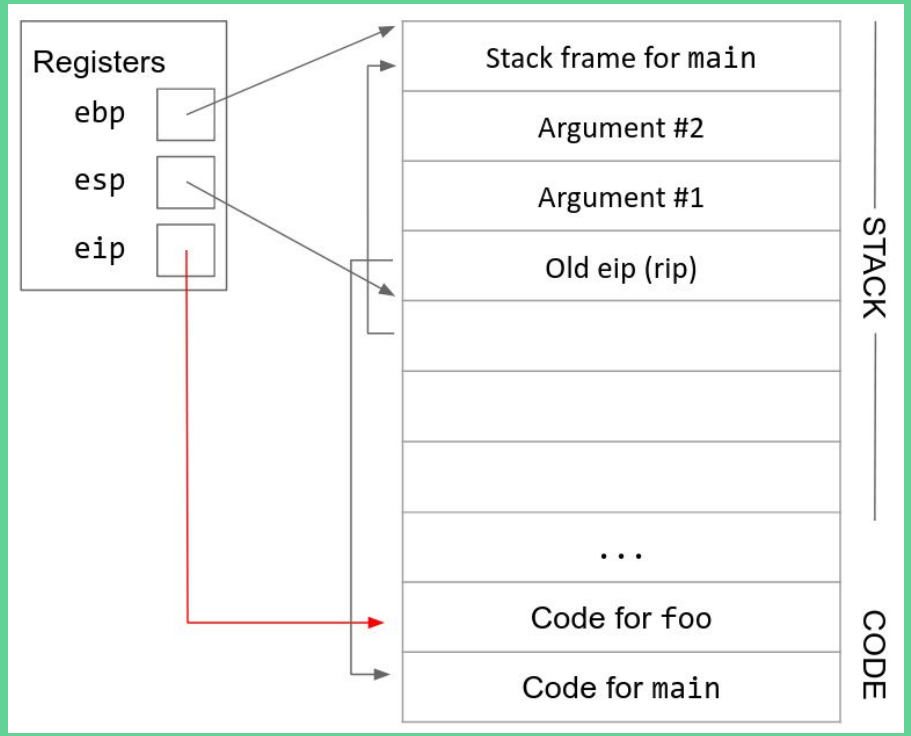


2. Push the old eip (return ip) on the stack

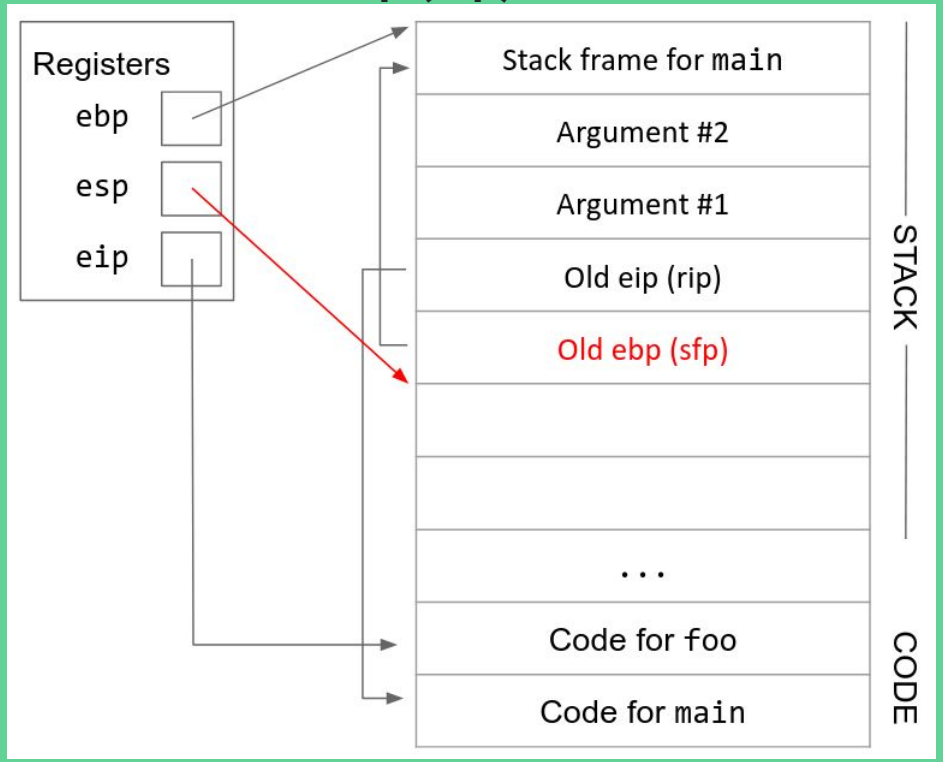


11 steps to calling an x86 function and returning- main calls the foo function-

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

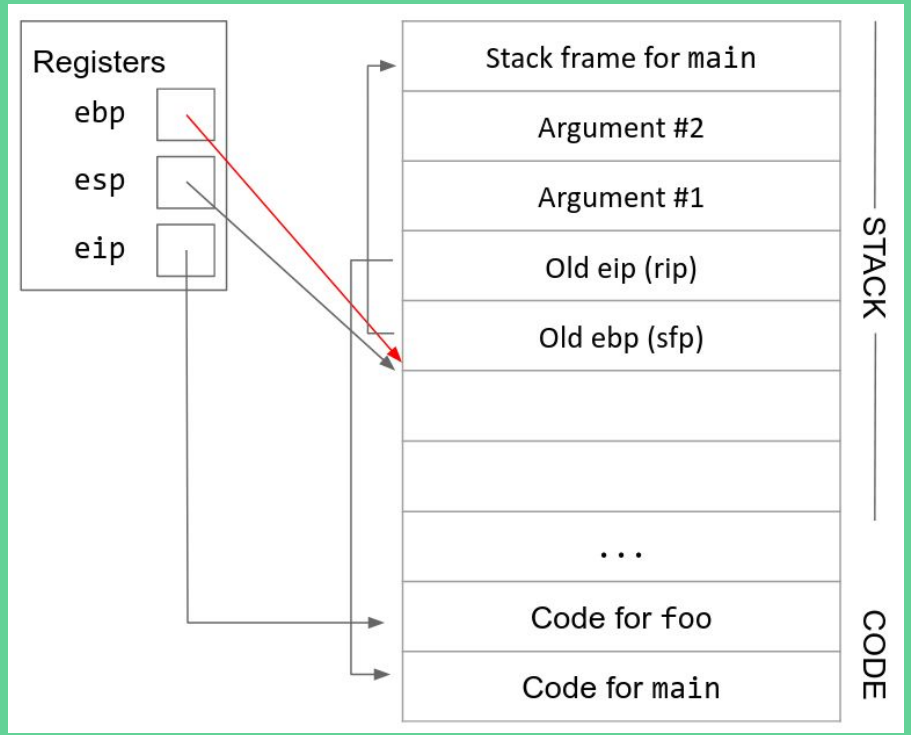


4. Push the old ebp (sfp) on the stack

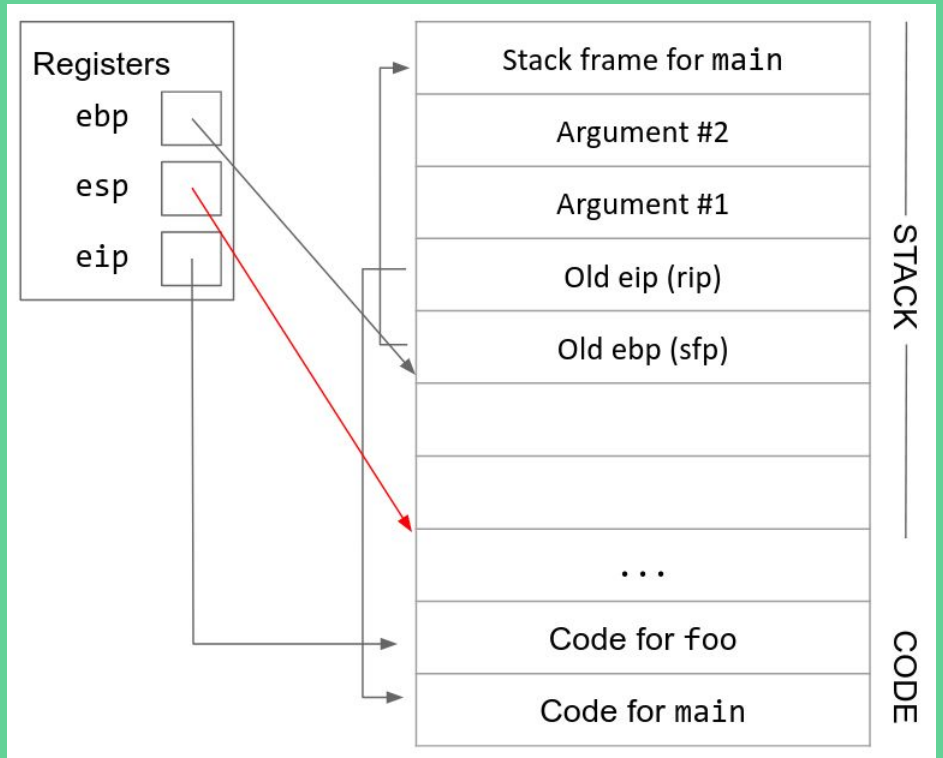


11 steps to calling an x86 function and returning- main calls the foo function-

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

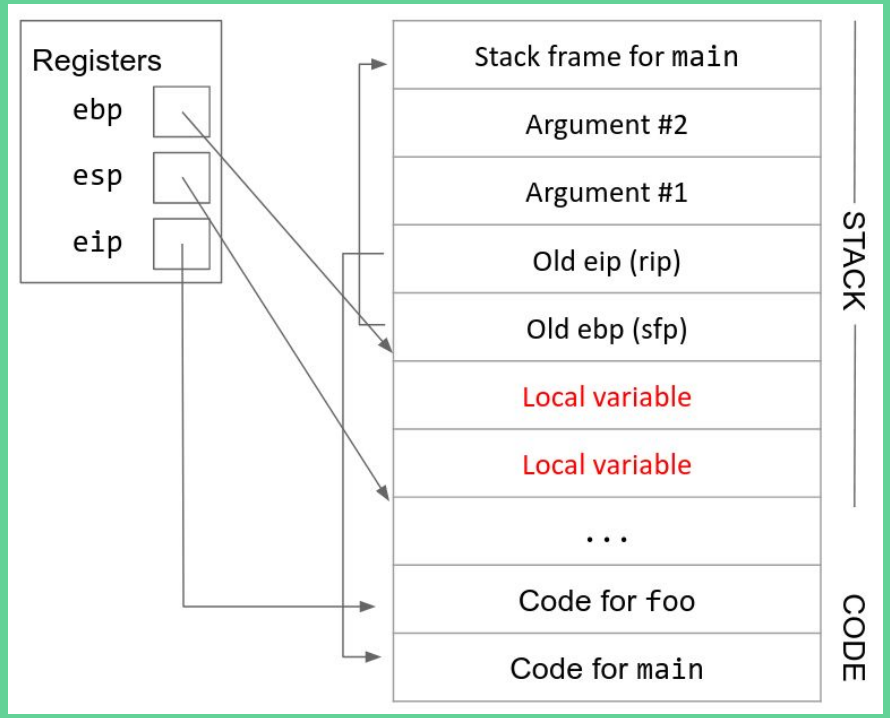


6 dec. ESP to alloc space for new stck fm

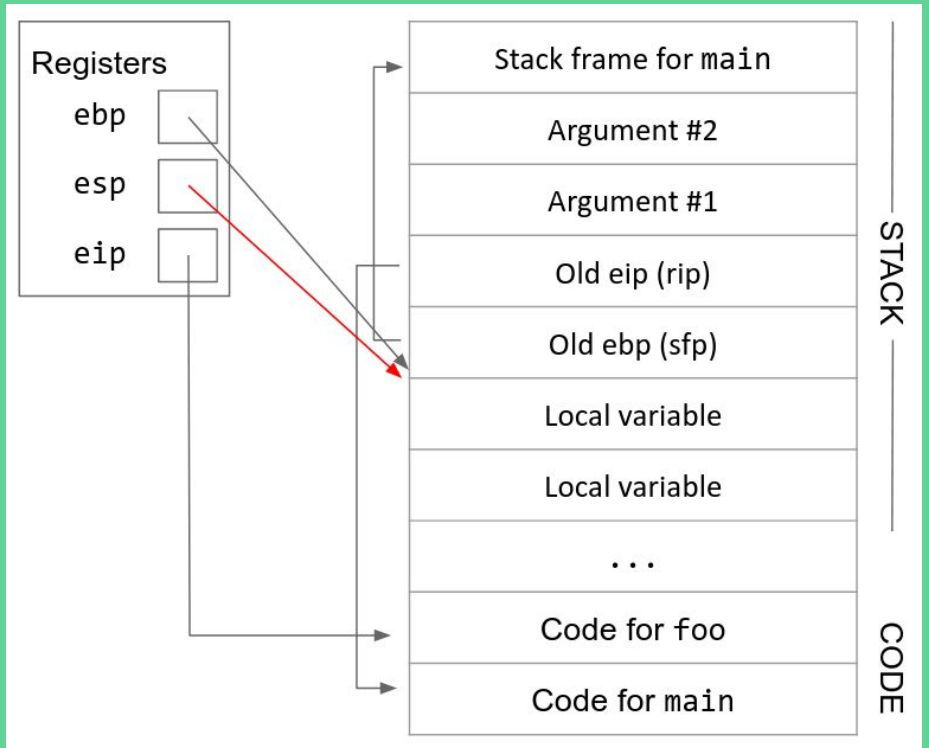


11 steps to calling an x86 function and returning- main calls the foo function-

7. Execute the function

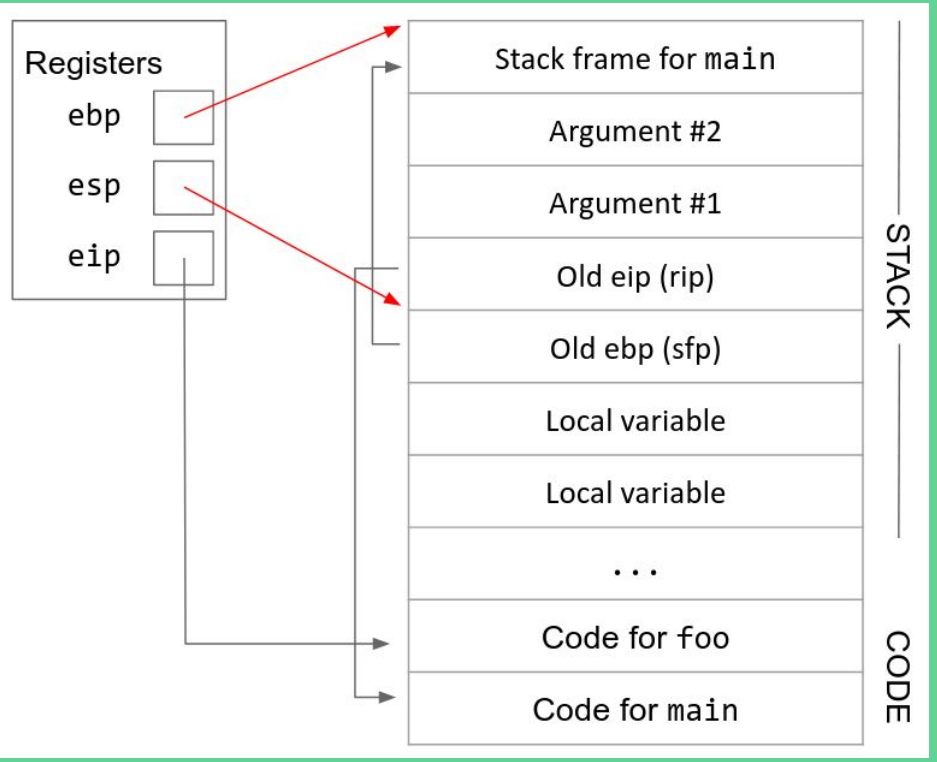


8. To return inc. esp to pt to top of stack

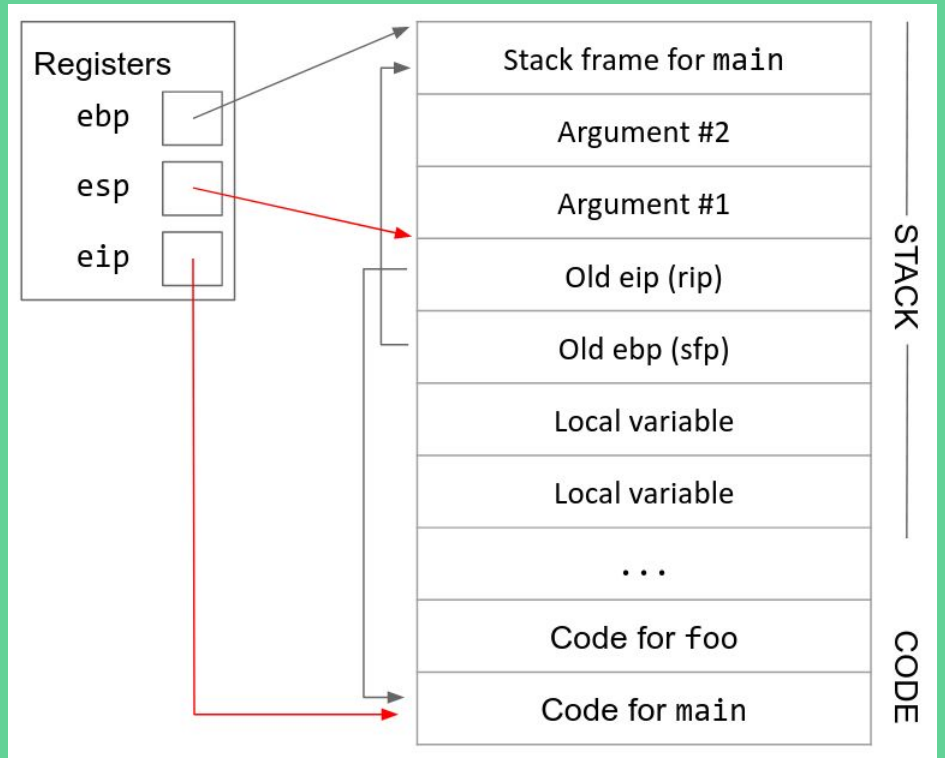


11 steps to calling an x86 function and returning- main calls the foo function-

9. Restore the old ebp (sfp)

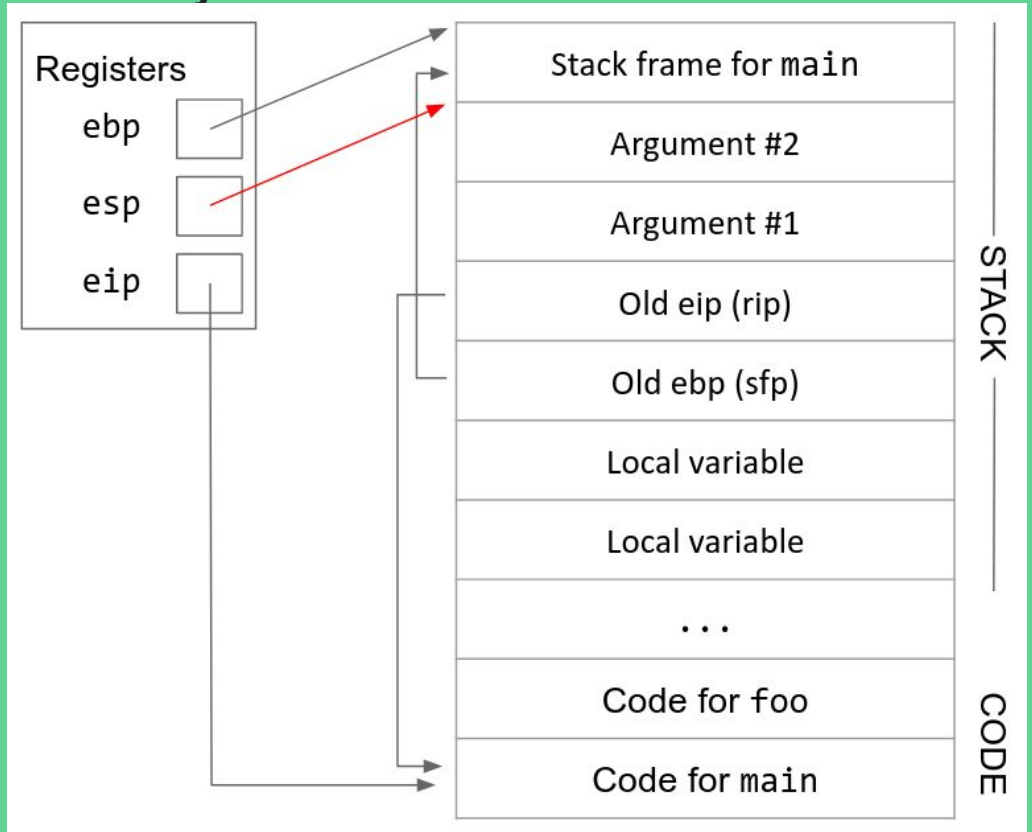


10. Restore the old eip (rip)



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

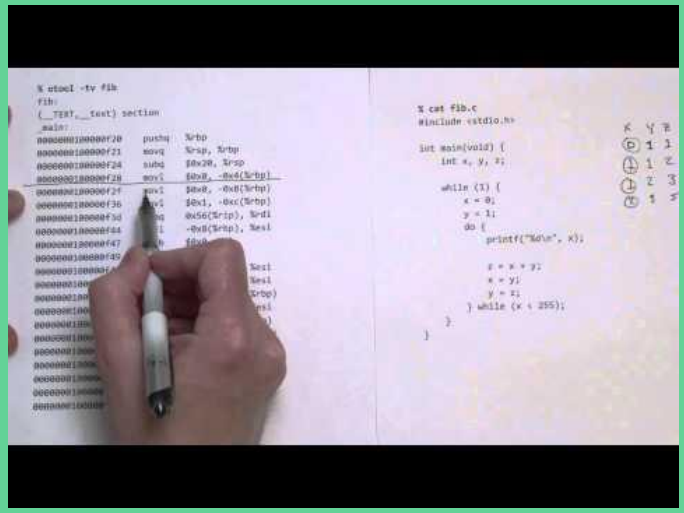


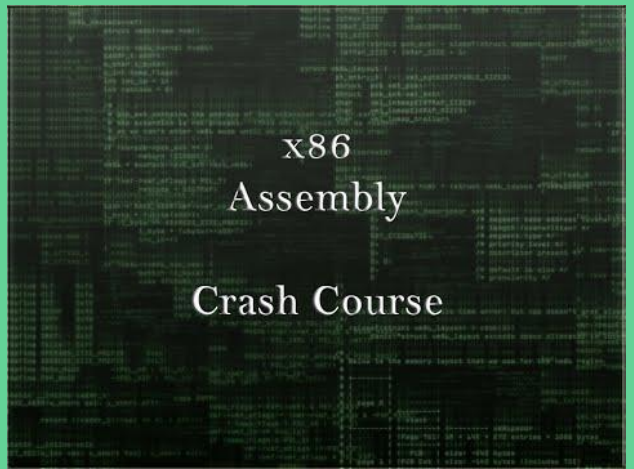
```c
int main(void) {
    foo(1, 2);
}

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





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

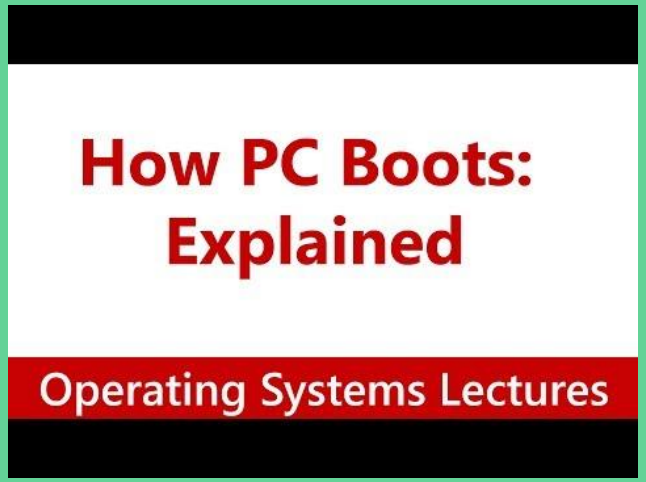
Day 2 agenda

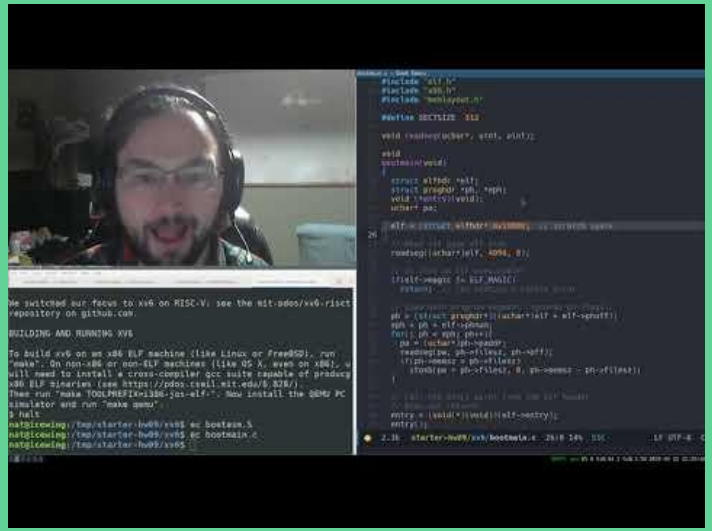
1. How PC boots - understand significance of 0x7c00, MBR signature 0x55AA
   BIOS->MBR-->grub/bootloader->kernel

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= 0xffff0
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
```

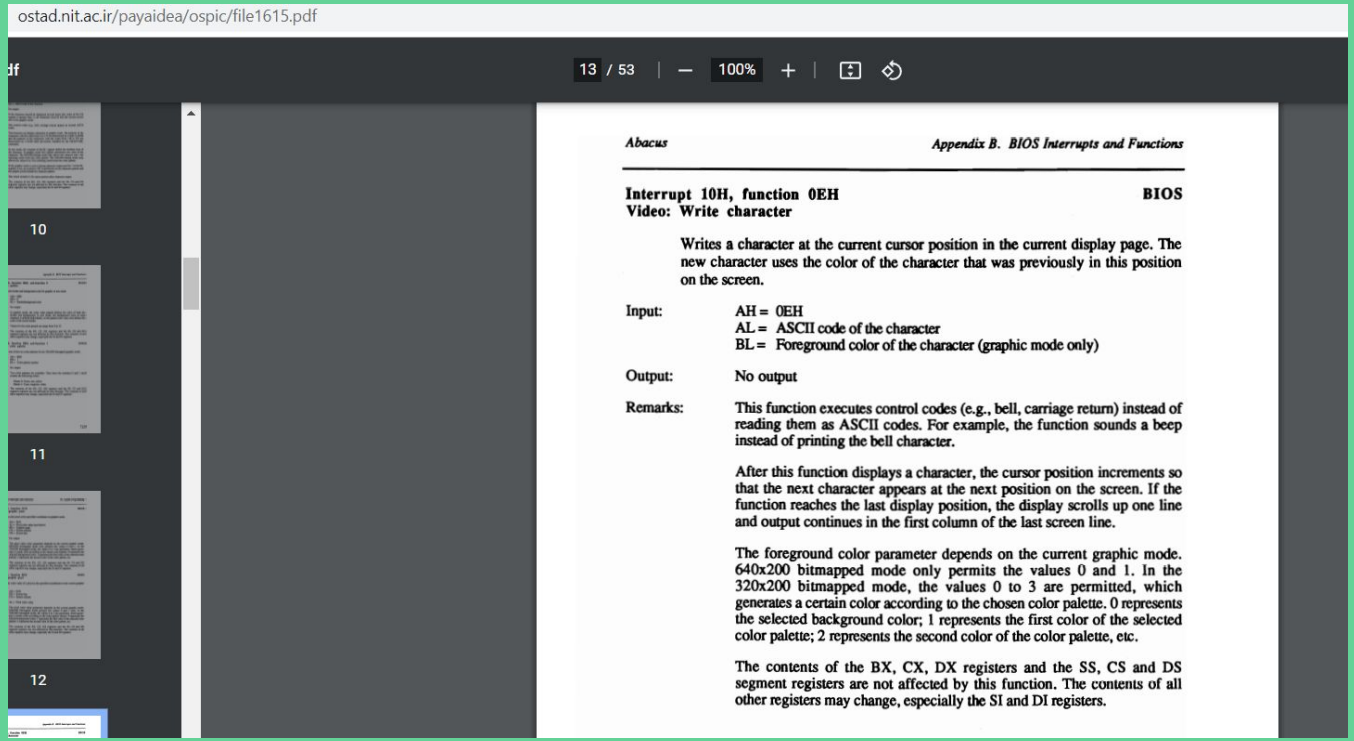
Bootloader printing helloworld ..

```
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://50linesofco.de/post/2018-02-28-writing-an-x86-hello-world-bootloader-with-assembly

BIOS Service 10H- video write char

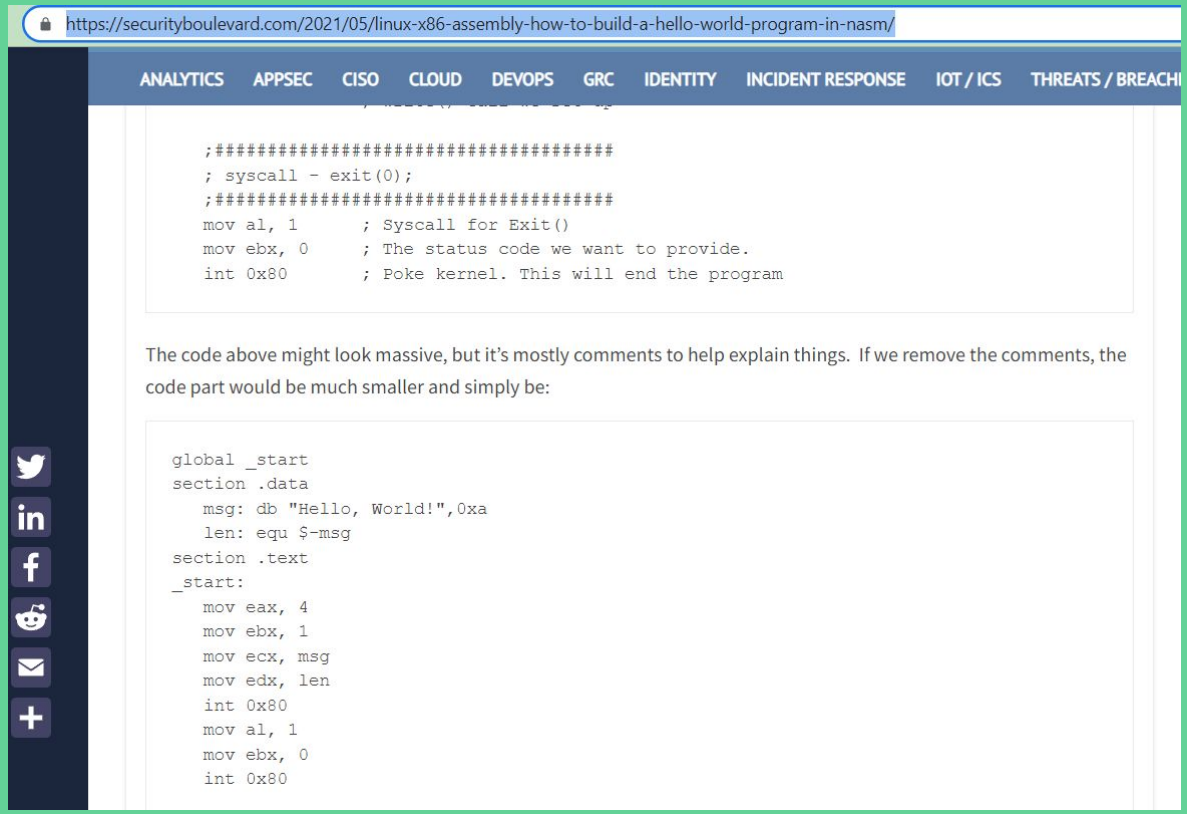


https://ostad.nit.ac.ir/payaidea/ospic/file1615.pdf

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]



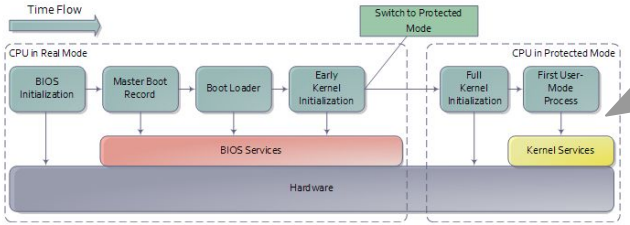
Bootloader “hello world”-bios interrupt, bios services lifespan, boot order, boot sector...

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

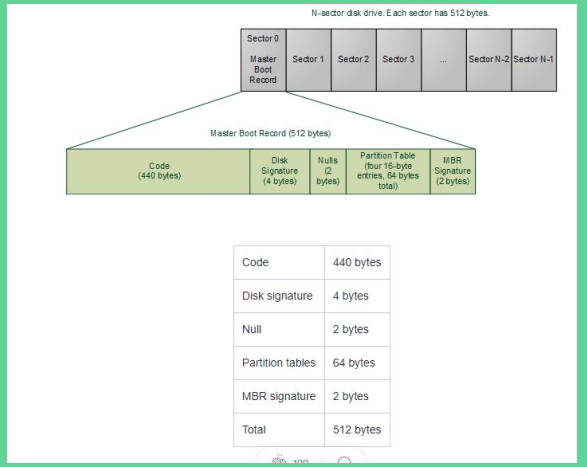
Maps | Java exercises... | Dysphagia - Di... | (33) Spring Tu... | (33) Important... | Insta

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)



Code	440 bytes
Disk signature	4 bytes
Null	2 bytes
Partition tables	64 bytes
MBR signature	2 bytes
Total	512 bytes

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

Makefile - Notepad

File Edit Format View Help

```
output:main.o f1.o
        gcc main.o f1.o -o output
main.o: main.c
        gcc -c main.c
f1.o:f1.c
        gcc -c f1.c
```

target

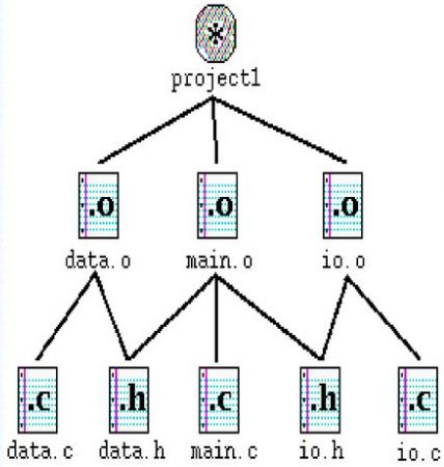
file(s) the target depends on (prerequisites)

cmd(s) to be executed if target is outdate w.r.t dependent files

```
target: depencencies
        command
```

Rule

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
```

make and Makefiles

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

Open | Makefile ~/neeraj/pub_xv6

```
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 bootasm.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

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

https://yohei.codes/post/xv6-bootblock/

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 :

```
ULIB = ulib.o usys.o printf.o umalloc.o

_%: %.o $(ULIB)
    $(LD) $(LDFLAGS) -N -e main -Ttext 0 -o $@ $^
    $(OBJDUMP) -S $@ > $*.asm
    $(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

https://stackoverflow.com/questions/29448967/understanding-code-block-from-xv6-makefile

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)
```

foo.c:

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

testmain.c:

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

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 $(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
```

https://stackoverflow.com/questions/53473410/how-to-fix-linker-error-when-compiling-xv6

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\
```

xv6 makefile parses | Renaissance of silicon-carbon double bonds (iluvrachel.github.io)

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
```

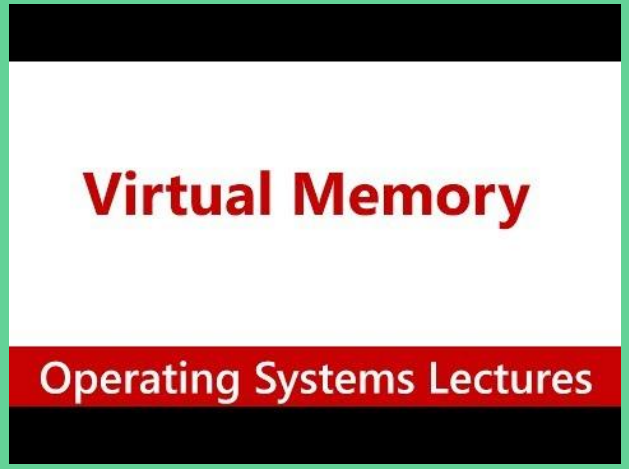
bootasm.S

```
  # 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 %eip.  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 0x8a00
  outw    %ax, %dx
spin:
  jmp     spin
```

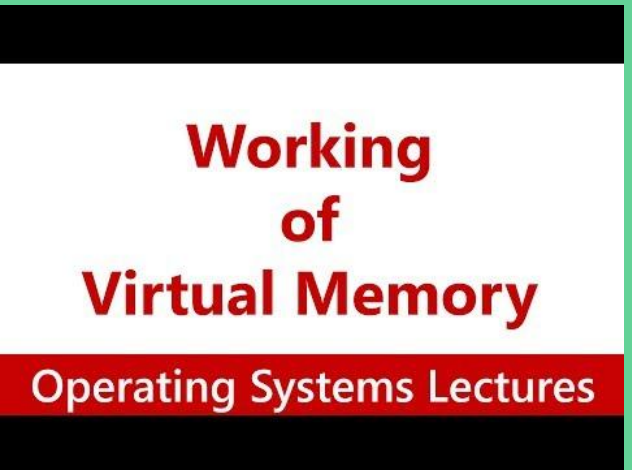
This part of the code is meant for entering into protected mode

To understand this we need to understand x86 memory mgt arch

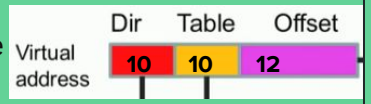
Memory Management - virtual memory



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

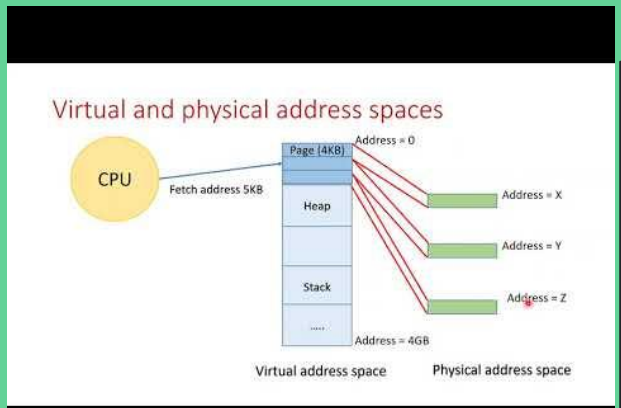


1 **Virtual address space of Process - contiguous address space in ELF (objdump -ph)**
2 MMU translates -VA to PA before raising read request on memory bus-using page table of process
3.4K page so 32 bit address is split into 12 bit offset & remaining 20 bit as page index
4. 2^20 => 1MB entries of 32 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

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
2.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 explicitely mentioned in instructions
      - Mov FS:$200, 30
  - 32 bit registers
    - can address upto 2^{32} memory

Div2 - xv6 bootloader code (2022-01-27)

X86 segmentation

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

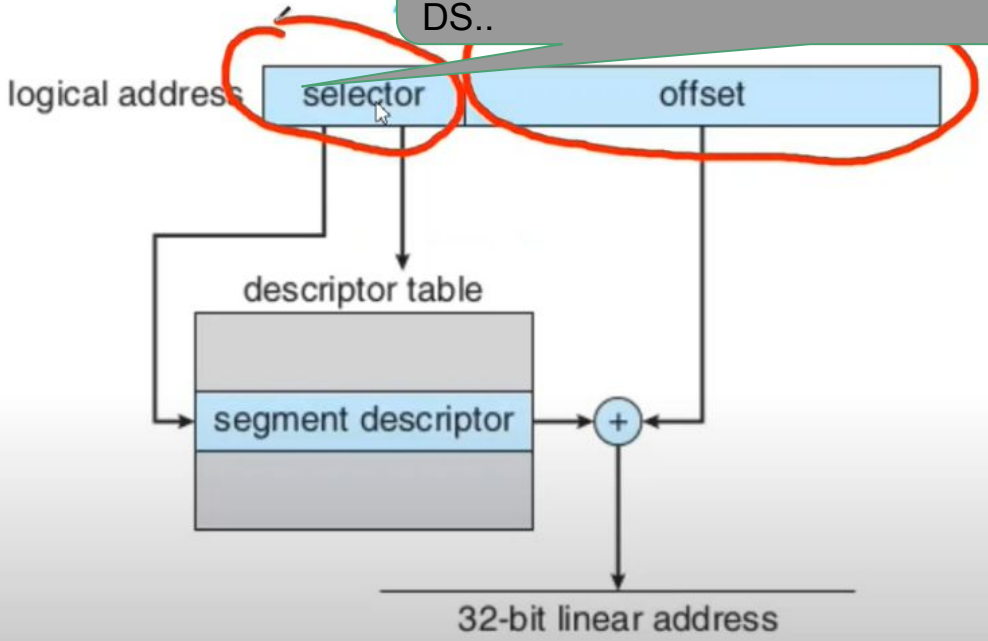
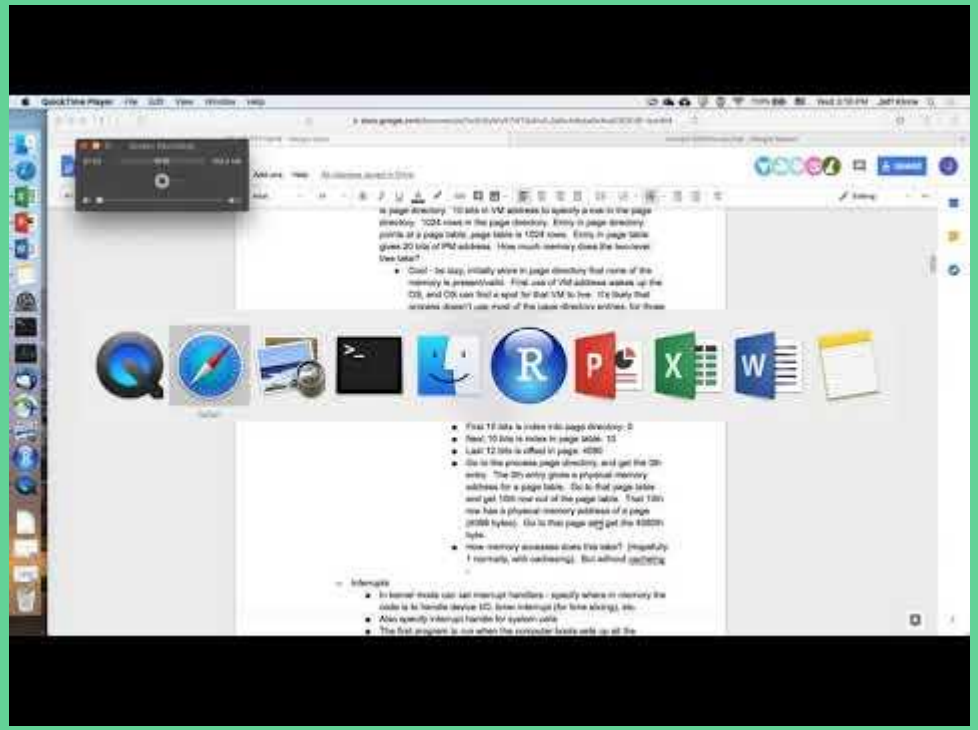


Figure 8.22 IA-32 segmentation.

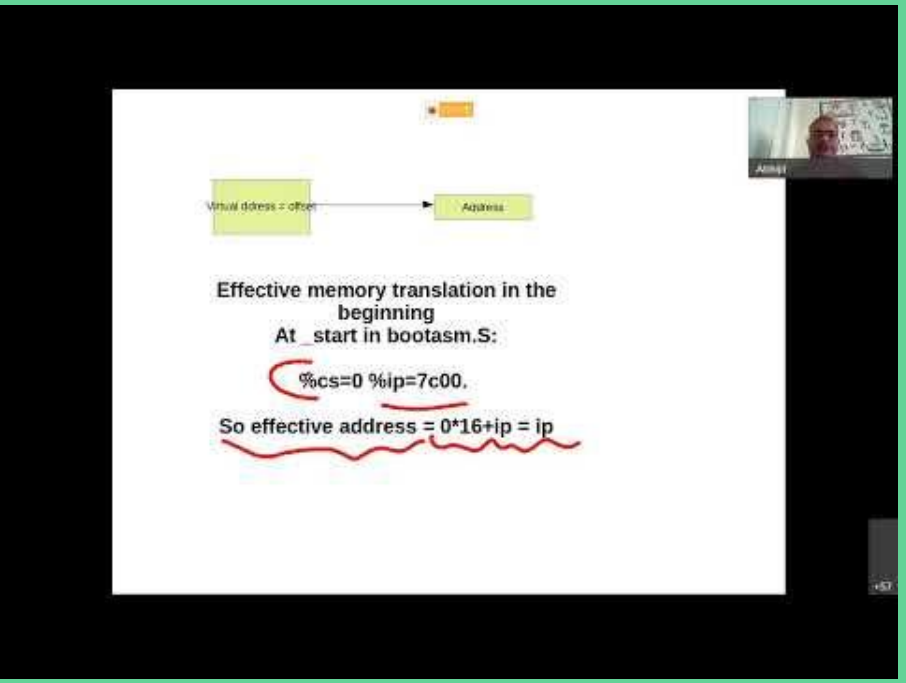
Assignment -Memory Management to walk a pointer VA (virtual address) to PA (physical address) mapping - first 12 mins



[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

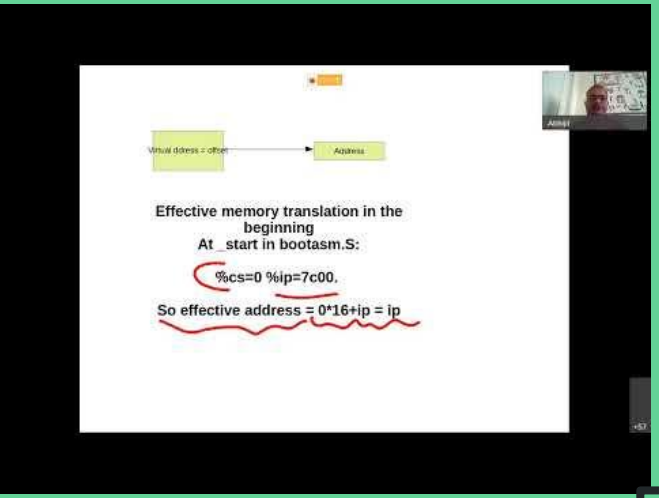
Bootloader part 1 & 2 - what does it do 1/2

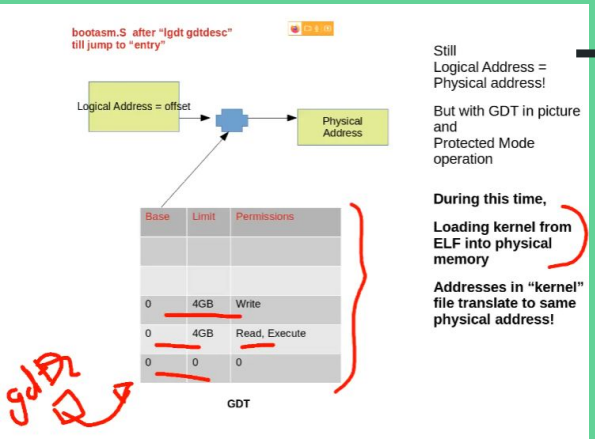


1. xv6 Makefile bootblock made up of bootasm.S & bootmain.c
2. bootasm.S is first binary and is meant to be located @0x7c00
3. cli -clear all interrupts - no ISRs are installed/expected as well.
4. Some code to enable above 20 (21) bit addressing -can be skipped
5. Line 42 -51 - are meant to switch from 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
9. (array has 3 entries- null entry, 0-4G r x, 0-4G w)-**how seg is effectively disabled?**
10. entire 4G memory as single segment
11. Enable protected mode by setting LSB/bit of CR0
12. 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

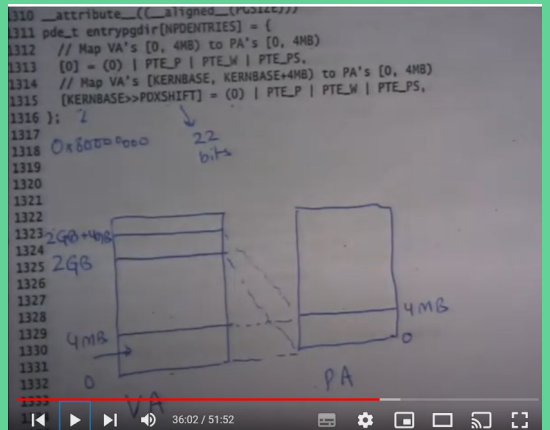
Day 4 - 12/16
Understanding xv6 bootloader - C part
Kernel bringup

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

0. Big picture - xv6 startup
    BIOS-> MBR(bootloader/bootblock.o)
    MBR/bootblock -> 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

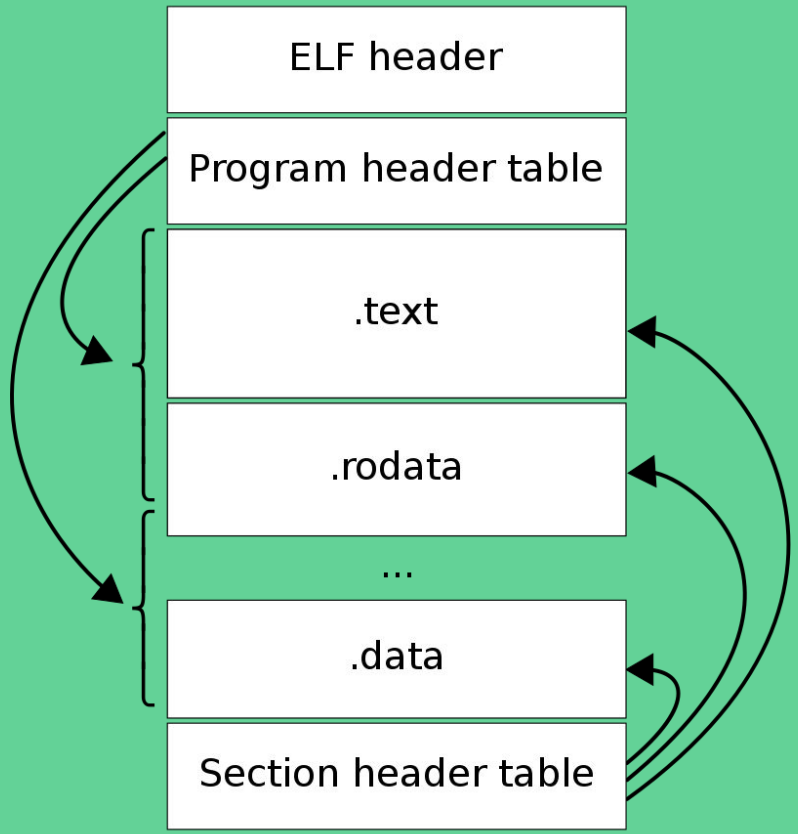
8. Writing first system call

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

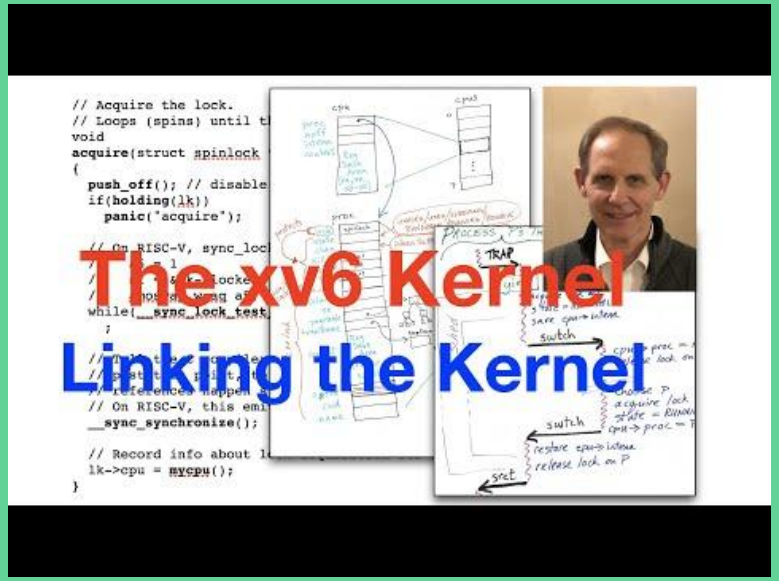


1. Recap of bootblock (bootasm.S) assembly part
   a. ldgdtr -glb desc tbl is array of 3 entries & ldgtr inst. make gdtr pt to that array **Q how segmentation is effectively disabled?**
2. Bootmain loader is doing one run of for loop for each segment of kernel program
3. objdump -p @21:05, above 0x100,000 (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 comercial 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  lapic.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

On power/REST
CS=0xf000 IP=0xfff0

Sector 0/MBR

bootblock.o

Bootasm.S

Bootmain.c

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()

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 (`lgdt 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
```

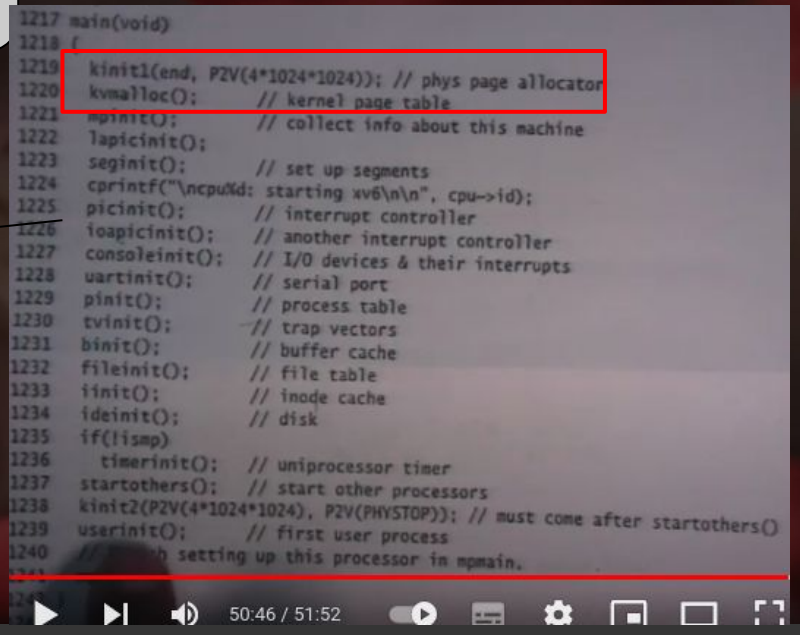
kernel

entry.S

main() init and start entire world of user processes

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)



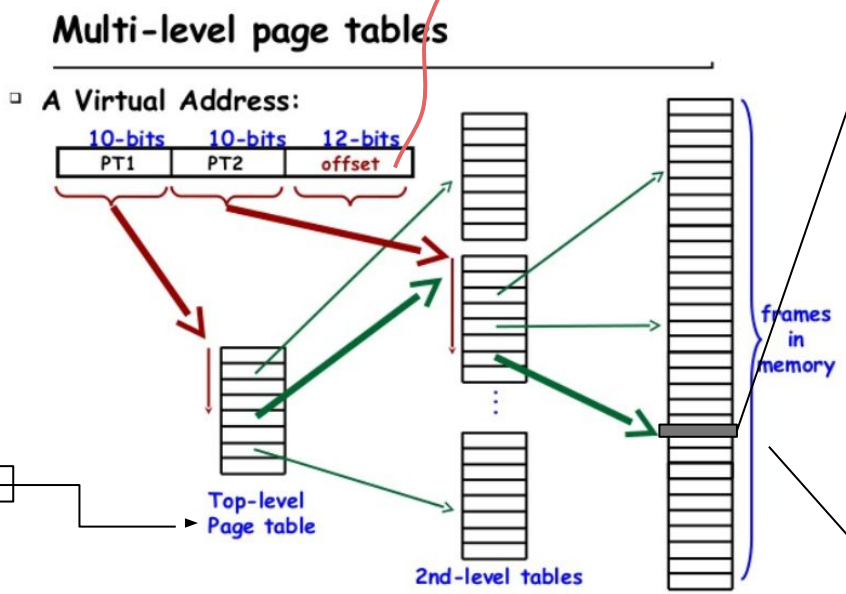
Multi level Page table

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

stackoverflow About Products For Teams Search...

2. They may map to secondary (or tertiary, etc, etc.) page tables

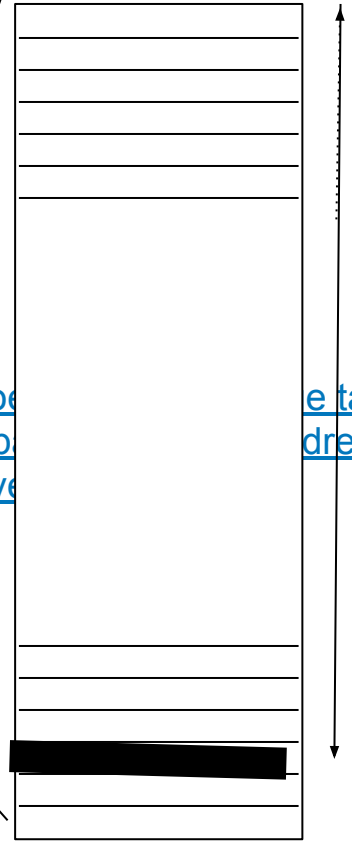
Here's an example of a multi-level page table:



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

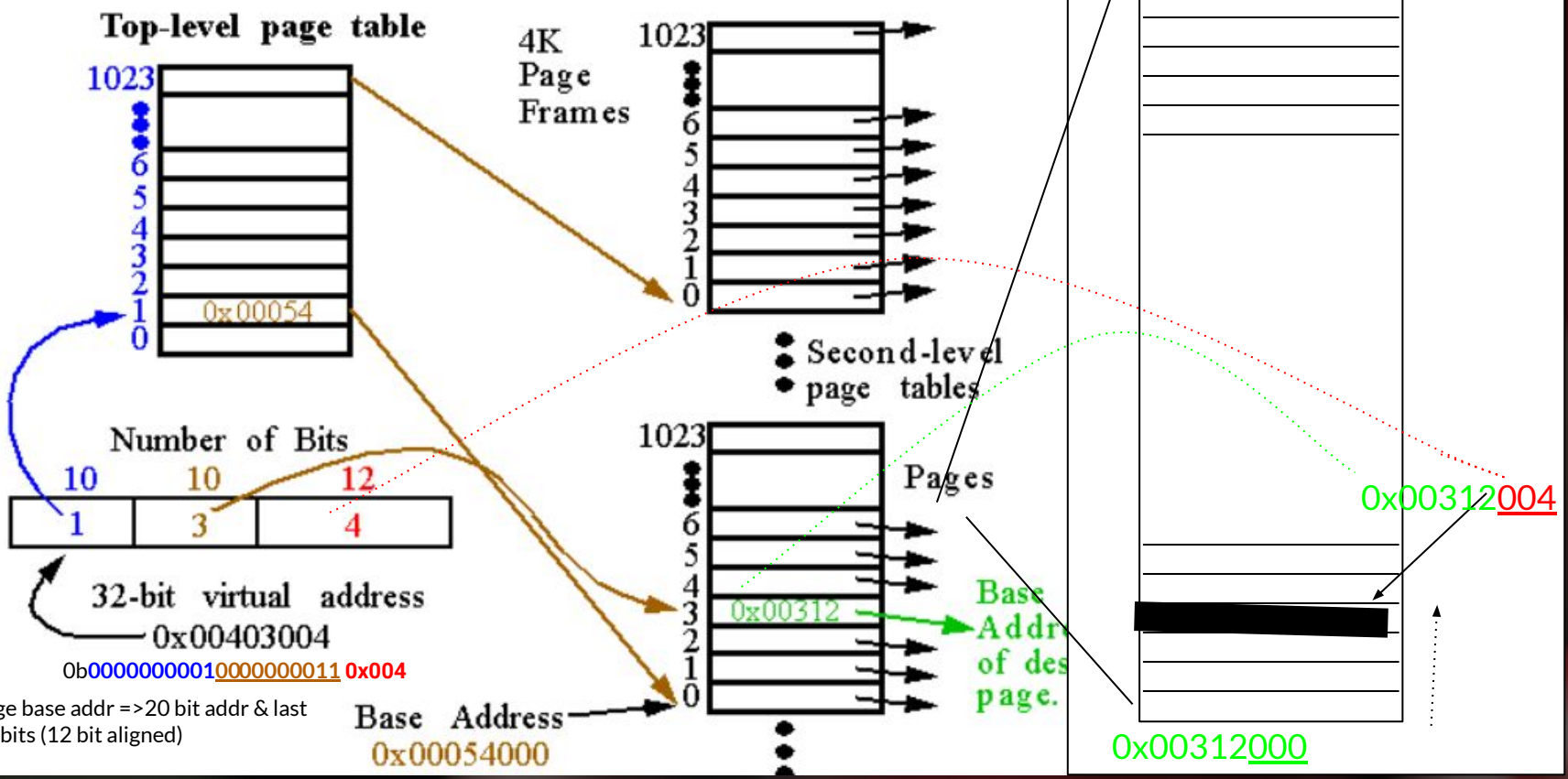


Doe e table represents a p dress? - Stack Ove

Multilevel Page Tables

http://ece-research.unm.edu/jimp/310/slides/os_essentials.html

• *Instead of using only one level of indirection, use t*



Which location in page

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

```
{
  kpgdir = setupkvm();
  switchkvm();
}

// Switch h/w page tab
// for when no process
void
switchkvm(void)
{
  lcr3(v2p(kpgdir));
}
```

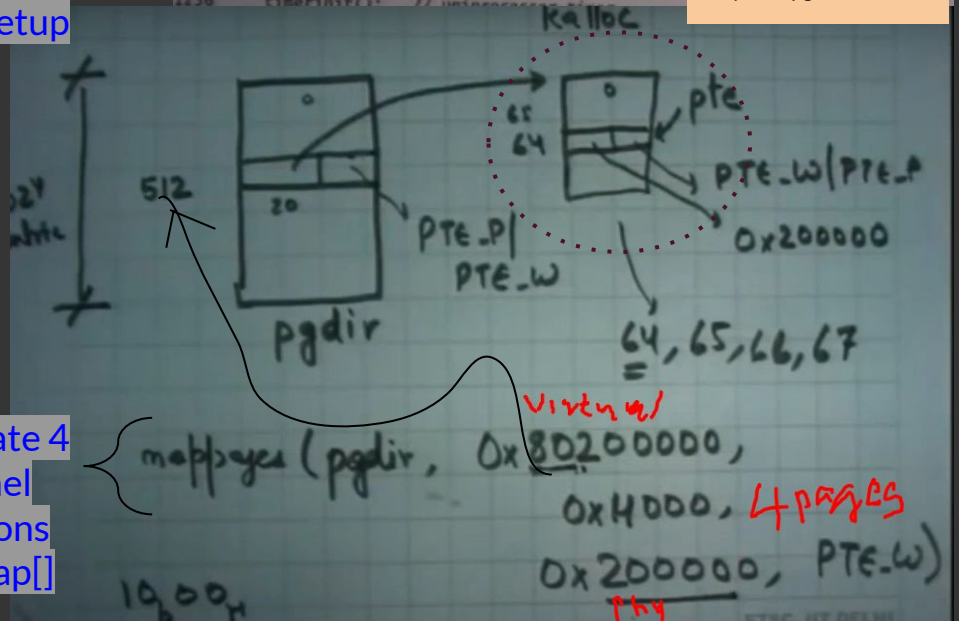
```
1217 main(void)
1218 {
1219   kinit1(end, P2V(4*1024*1024)); // phys page allocator
1220   kvmalloc();      // kernel page table
1221   mpinit();        // collect info about this machine
1222   lapicinit();
1223   seginit();       // set up segments
1224   cprintf("\ncpu%d: starting xv6\n\n", cpu->id);
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(!ismp)
1236     timerinit();   // uniprocessor
```

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

mappages()

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

kpgdir setup



Iterate 4 kernel regions -kmap[]

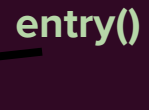
- Page table entries added by "mappages"
  - Arguments: page directory, range of virtual addresses, physical addresses to map to, permissions of the pages
  - For each page, walks page table, get pointer to PTE via function "walkpgdir", fills it with physical address and permissions
- Function "walkpgdir"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

Add entries in outer page tbl & call walkpgdir to build (or alloc/build inner tbl)

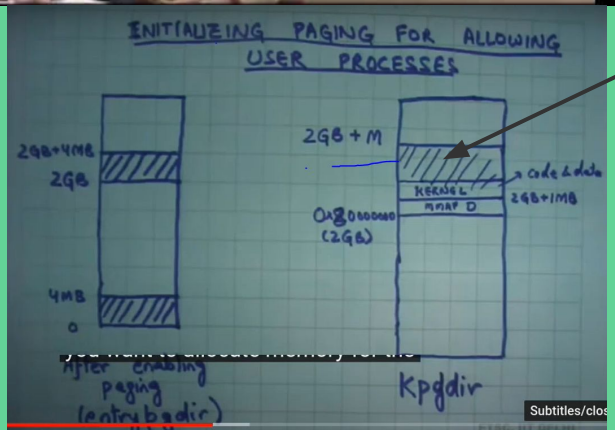
```
1756 // Create PTEs for virtual addresses starting at va that refer to
1757 // physical addresses starting at pa. va and size might not
1758 // be page-aligned.
1759 static int
1760 mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
1761 {
1762   char *a, *last;
1763   pte_t *pte;
1764
1765   a = (char*)PGROUNDDOWN((uint)va);
1766   last = (char*)PGROUNDDOWN(((uint)va) + size - 1);
1767   for(;;){
1768     if((pte = walkpgdir(pgdir, a, 1)) == 0)
1769       return -1;
1770     if(*pte & PTE_P)
1771       panic("remap");
1772     *pte = pa | perm | PTE_P;
1773     if(a == last)
1774       break;
```

```
1731 // Return the address of the PTE in page table pgdir
1732 // that corresponds to virtual address va.  If alloc!=0,
1733 // create any required page table pages.
1734 static pte_t *
1735 walkpgdir(pde_t *pgdir, const void *va, int alloc)
1736 {
1737   pde_t *pde;
1738   pte_t *pgtab;
1739
1740   pde = &pgdir[PDX(va)];
1741   if(*pde & PTE_P){
1742     pgtab = (pte_t*)P2V(PTE_ADDR(*pde));
1743   } else {
1744     if(!alloc || (pgtab = (pte_t*)kalloc()) == 0)
1745       return 0;
1746     // Make sure all those PTE_P bits are zero.
1747     memset(pgtab, 0, PGSIZE);
1748     // The permissions here are overly generous, but they can
1749     // be further restricted by the permissions in the page table
1750     // entries, if necessary.
1751     *pde = V2P(pgtab) | PTE_P | PTE_W | PTE_U;
```

```
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

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
```



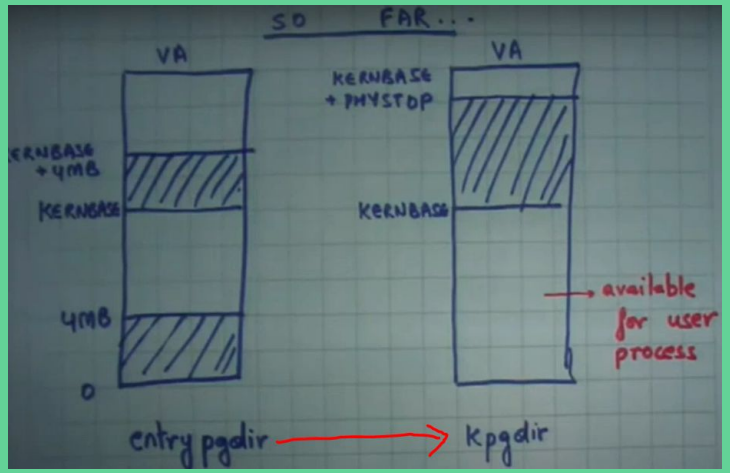
Setting up page table in for kernel - changing from entrydir to kernelpgdir



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
        iii. 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      switchuvm(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

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

stackoverflow About Products For Teams Search...

Home

PUBLIC

Questions

Tags

Users

Companies

COLLECTIVES

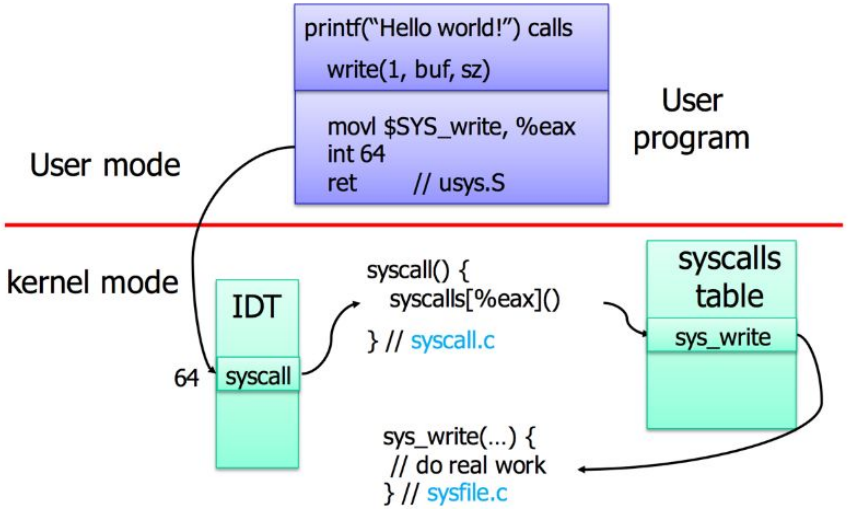
Explore Collectives

TEAMS

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

Free

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]

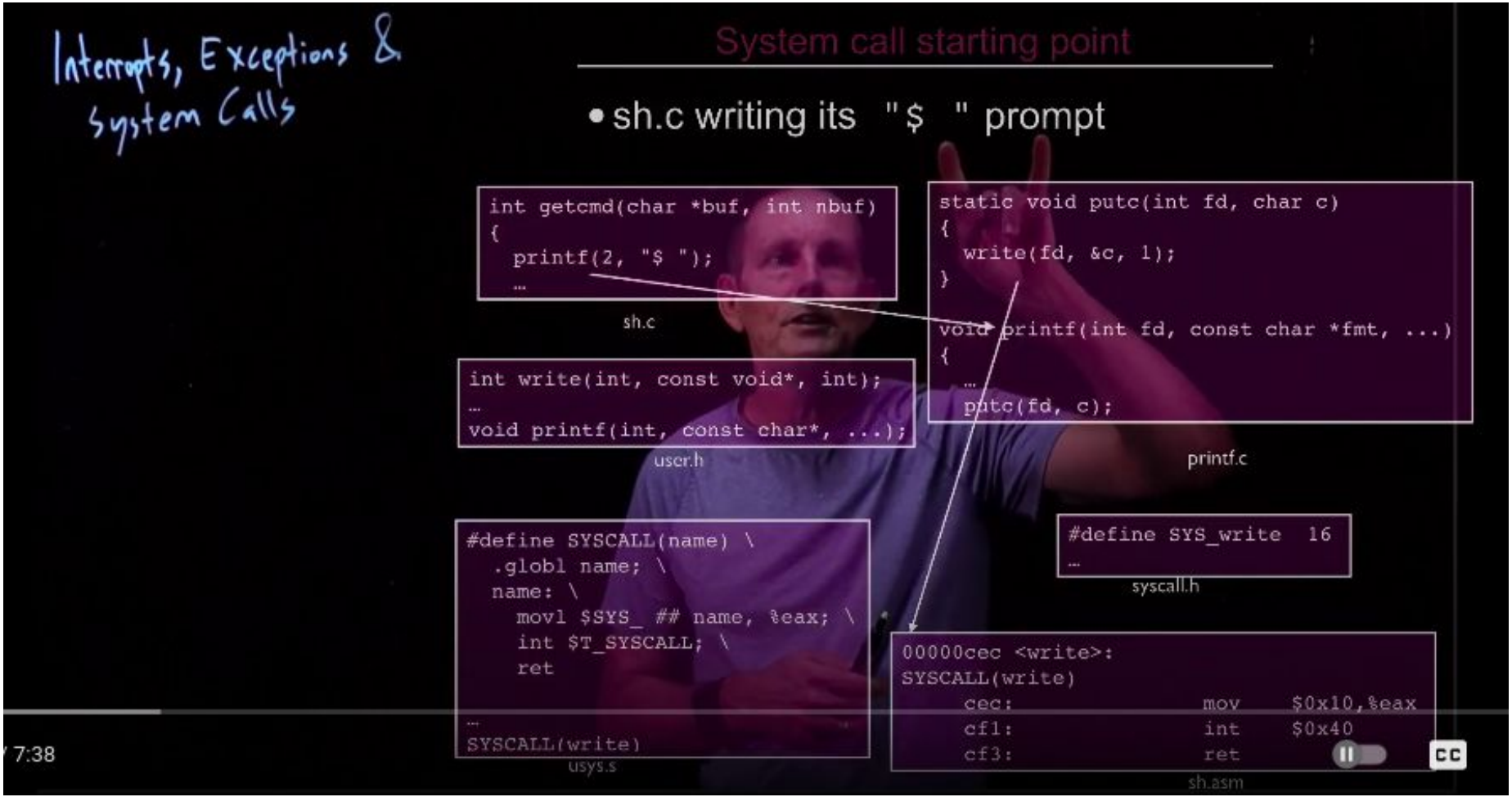
Typos -

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

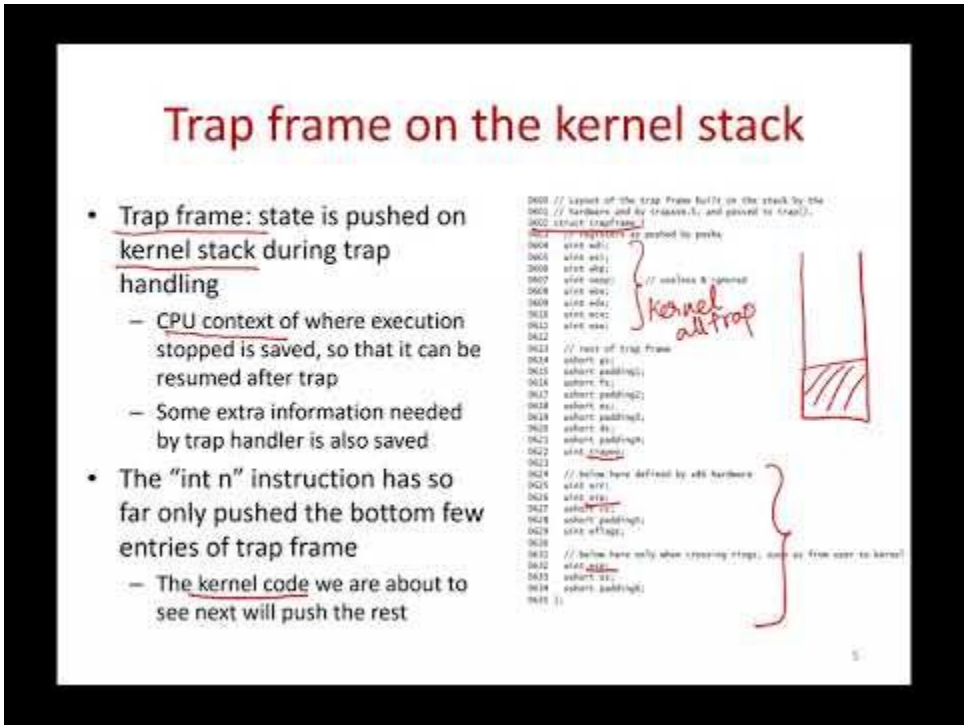
```
we need to modify following
files :
  1) syscall.c
  2) syscall.h
  3) sysproc.c
  4) usys.S
  5) user.h
  6) user prog
     to invoke and
  7) change Makefile
```

kernel

User space


Interrupts, Exceptions & System Calls
System call starting point
• sh.c writing its "$ " prompt
int getcmd(char *buf, int nbuf)
{
printf(2, "$ ");
…
sh.c
static void putc(int fd, char c)
{
write(fd, &c, 1);
}
void printf(int fd, const char *fmt, ...)
{
…
putc(fd, c);
printf.c
int write(int, const void*, int);
…
void printf(int, const char*, ...);
user.h
#define SYS_write 16
…
syscall.h
#define SYSCALL(name) \
.globl name; \
name: \
movl $SYS_ ## name, %eax; \
int $T_SYSCALL; \
ret
…
SYSCALL(write)
usys.s
00000cec <write>:
SYSCALL(write)
cec: mov $0x10,%eax
cf1: int $0x40
cf3: ret
sh.asm
7:38
CC

Code walk through



How does a system call work [duplicate]

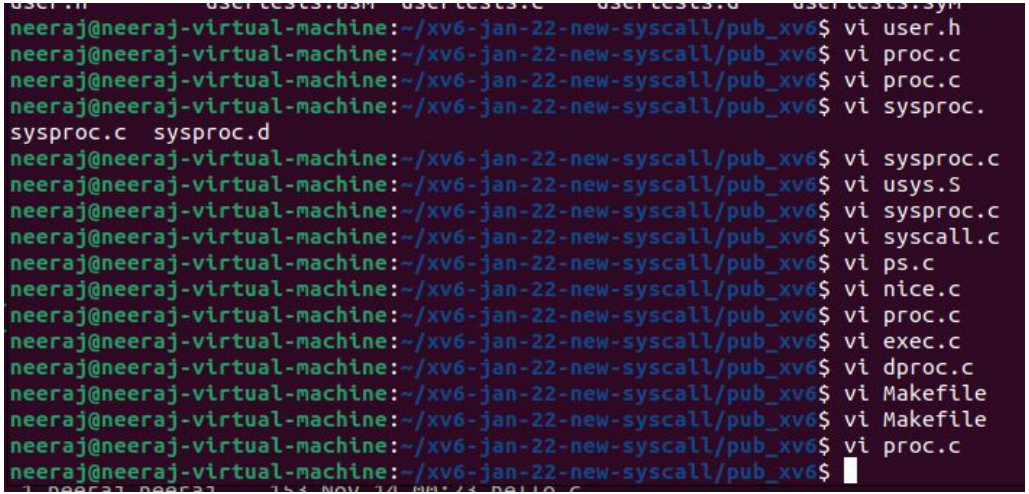
https://medium.com/@mahi12/adding-system-call-in-xv6-a5468ce1b463

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

Implementing PS & nice



[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

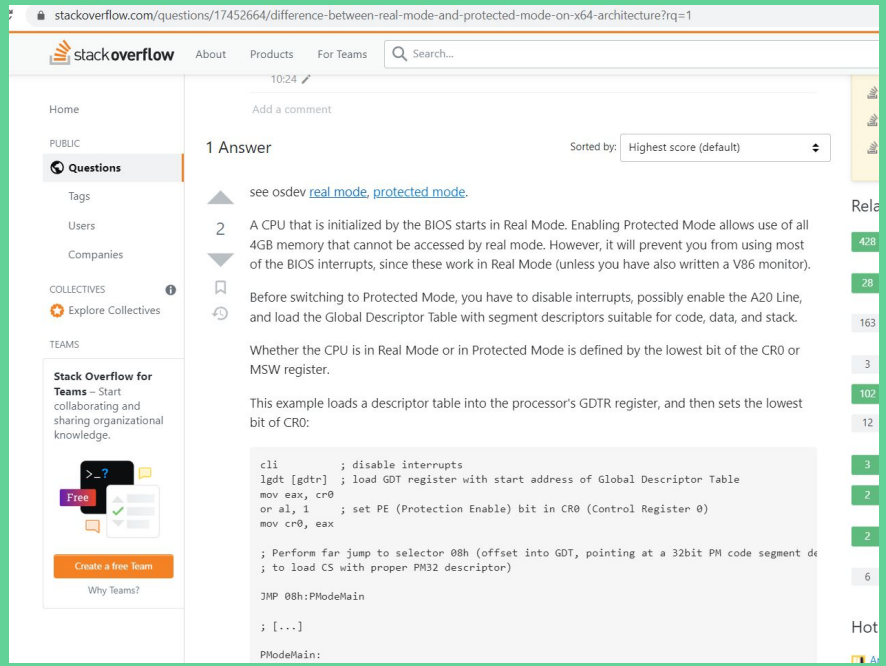
Day 7 & 8

- 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

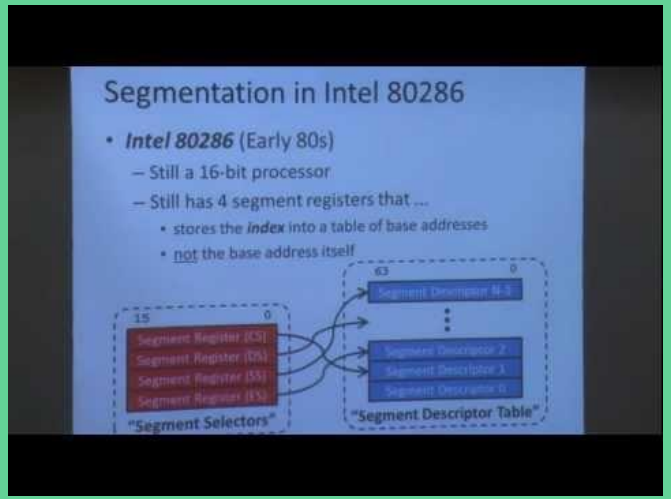
Pune professor makefile analysis bootblock (2 parts bootloader.s and bootmain.c)

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



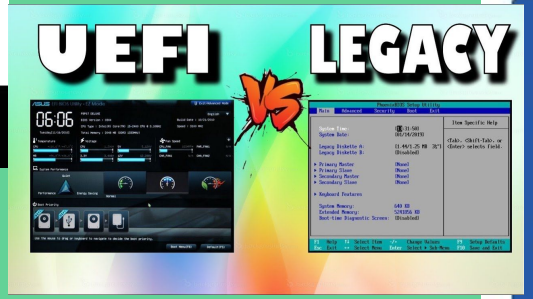
[difference between real mode and protected mode on x64 architecture - Stack Overflow]

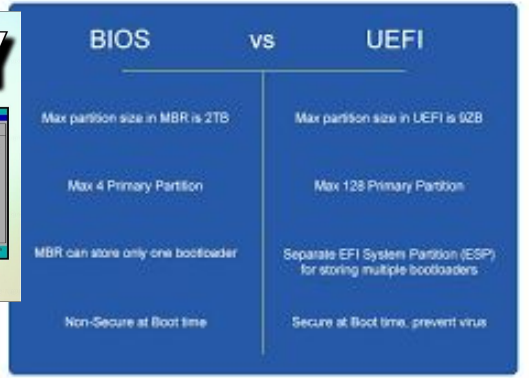
A good description of segment descriptor GDTR Register/swing to Protected mode



Interesting facts & taking deeper dive -UEFI vs BIOS

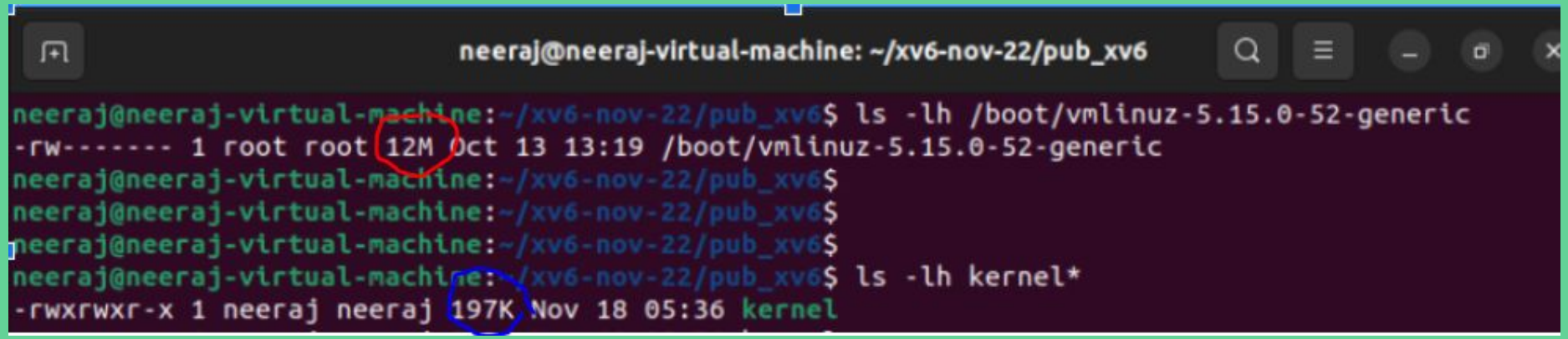
1. Computer is like a box with lot of h/w components- storage devices,GPU..
2. Processor needs to be told what to do -this is done by OS -paradox OS is on disk that is powered off, how it is going to be told what is done by whom
3. BIOS is like mini OS on chip available at power on- that initializes 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.



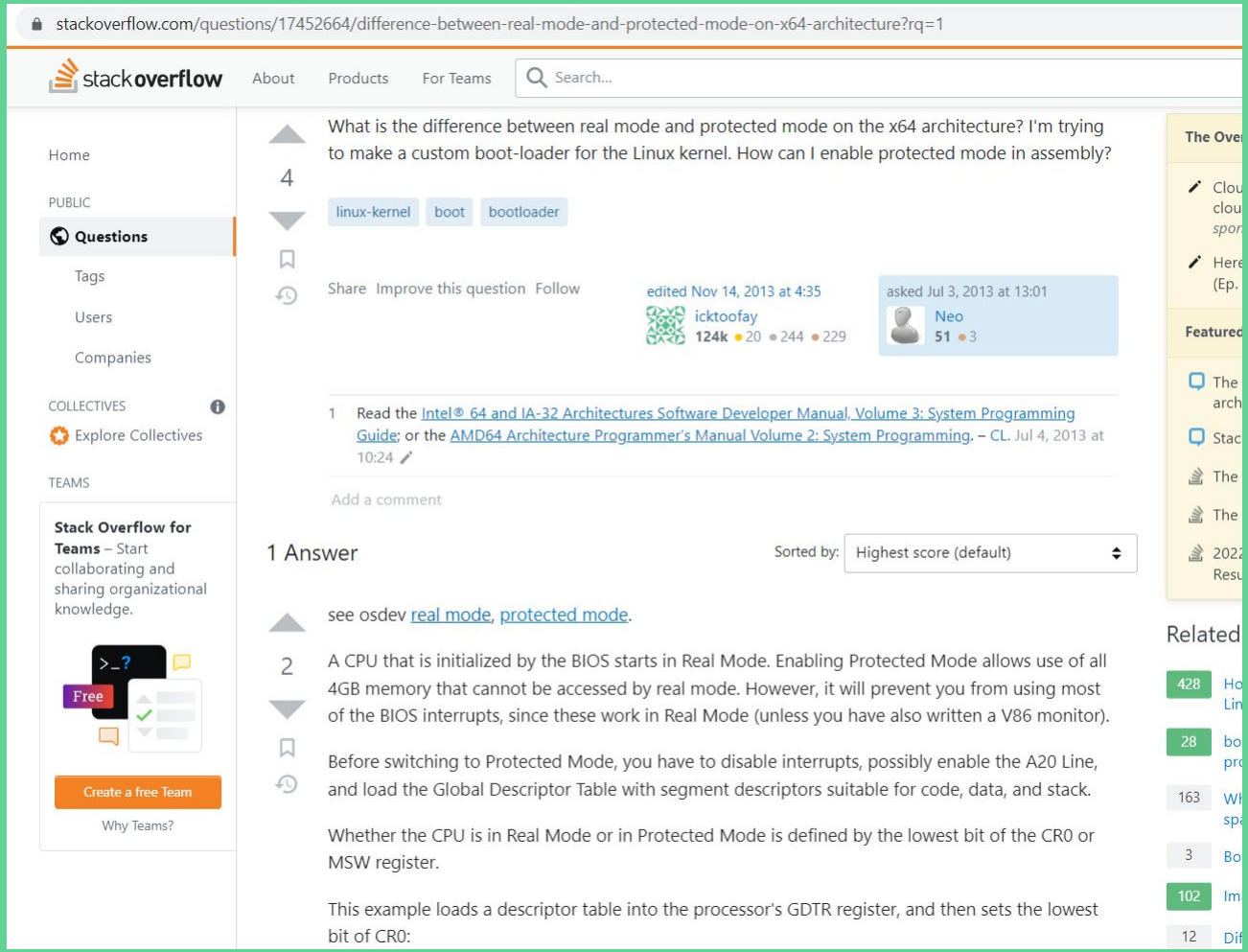


Interesting facts & taking deeper dive

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



Interesting facts & taking deeper dive



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

CFS