CS3210: Virtual Memory

Lecture 5

Kyle Harrigan

Administrivia

- (Feb 9) Lab 3 assigned
- (Feb 10) Lab2 due
- (Feb 16) Quiz #1. Lab1-2, Ch 0-2, Appendix A/B

Administrivia

- (Feb 9) Lab 3 assigned
- (Feb 10) Lab2 due
- (Feb 16) Quiz #1. Lab1-2, Ch 0-2, Appendix A/B

Questions?

Summary of last lectures

- Tools
  - GIT
  - QEMU
  - Booting Basics
- Booting and x86
  - BIOS
  - x86 Overview
  - Segmentation
  - Basic I/O
- Makefiles, C, gdb
- Shell and OS Organization
  - Shells
  - open/dup/fork/pipe/close
  - File descriptors
  - OS designs: monolithic (xv6) vs. micro kernels (jos)
  - Isolation mechanisms: CPL (aka ring), address space (aka process)

Lecture Motivation: Lab 2

In lab2, the focus is on writing memory management code:

Physical memory allocator:

- Which physical pages are free and which are in use?

Virtual Memory:

- How to map virtual addresses to physical memory
- Permissions…

Virtual Memory Motivation

Why Virtual Memory?

Virtual Memory Motivation

Why Virtual Memory?

- Primary purpose: isolation
  - Each process has its own address space
  - Also convenient programming model (large continuous address space)
  - Alternatives (segmentation, etc.)
- Benefits:
  - Memory utilization, fragmentation, sharing, etc.
- Level-of-indirection
  - Provides kernel with opportunity to do cool stuff (example?)

Basic questions 1

- Can we use ring0 for userspace, ring3 for kernel (yes/no)? (ref Hardware isolation in x86)
- Kernel can not read userspace memory (yes/no)?
- Userspace can not read kernel's memory (yes/no)?

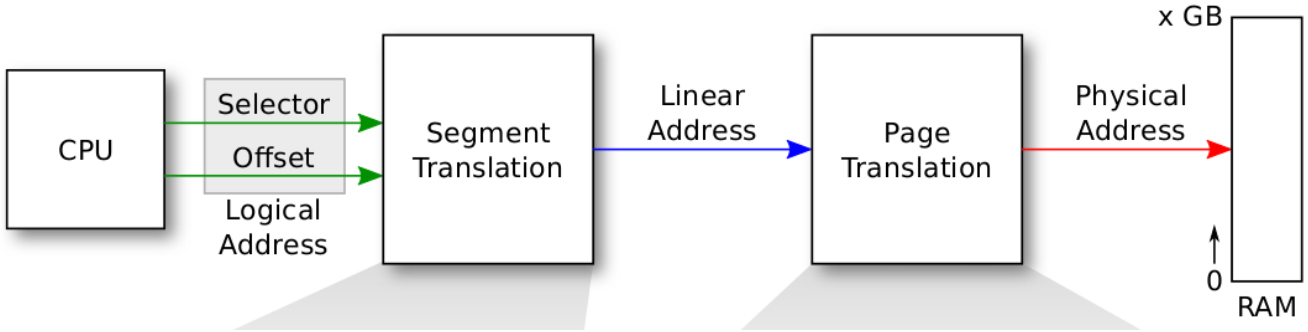
Basic questions 1

- Can we use ring0 for userspace, ring3 for kernel (yes/no)? (ref Hardware isolation in x86)
- Kernel can not read userspace memory (yes/no)?
- Userspace can not read kernel's memory (yes/no)?

Basic questions 2

- Pointers in kernel are using physical address (yes/no)?
- Processes requiring 3GB memory cannot run on a machine with 2GB RAM (yes/no)?

Big picture: address translation



- Segmentation vs. paging?
- Virtual address?

DEMO: Find physical pages (32-bit Linux)

```
/proc/pid/pagemap.  This file lets a userspace process find out which
physical frame each virtual page is mapped to.  It contains one 64-bit
value for each virtual page, containing the following data (from
fs/proc/task_mmu.c, above pagemap_read)
```

DEMO: Find physical pages (32-bit Linux)

```
/proc/pid/pagemap.  This file lets a userspace process find out which
physical frame each virtual page is mapped to.  It contains one 64-bit
value for each virtual page, containing the following data (from
fs/proc/task_mmu.c, above pagemap_read)
```

```
$ ./test
/proc/30118/pagemap, buf=0xbff4bfdc, main=0x80484cd

$ sudo ./read_virt /proc/30118/pagemap 0xbff4bfdc
0x8600000000070ba, pfn=28858

$ sudo gdb -c /proc/kcore
(gdb) x/30b 0xc0000000 + 28858*4096 + 4060
0xc70bafdc: 0   1   2   3   4   5   6   7
0xc70bafe4: 8   9   10  11  12  13  14  15
0xc70bafec: 16  17  18  19  20  21  22  23
0xc70baff4: 24  25  26  27  28  29
```

DEMO: Find physical pages (32-bit Linux)

```
/proc/pid/pagemap.  This file lets a userspace process find out which
physical frame each virtual page is mapped to.  It contains one 64-bit
value for each virtual page, containing the following data (from
fs/proc/task_mmu.c, above pagemap_read)
```

```
$ ./test
/proc/30118/pagemap, buf=0xbff4bfdc, main=0x80484cd

$ sudo ./read_virt /proc/30118/pagemap 0xbff4bfdc
0x8600000000070ba, pfn=28858

$ sudo gdb -c /proc/kcore
(gdb) x/30b 0xc0000000 + 28858*4096 + 4060
0xc70bafdc: 0   1   2   3   4   5   6   7
0xc70bafe4: 8   9   10  11  12  13  14  15
0xc70bafec: 16  17  18  19  20  21  22  23
0xc70baff4: 24  25  26  27  28  29
```

What did we just do?

- Test program initializes a buffer with values, gives us a pointer address to the buffer, and a pid
- We read the local process page map, which gives us a physical page number, referenced from KERNBASE
- We accessed this directly using /proc/core

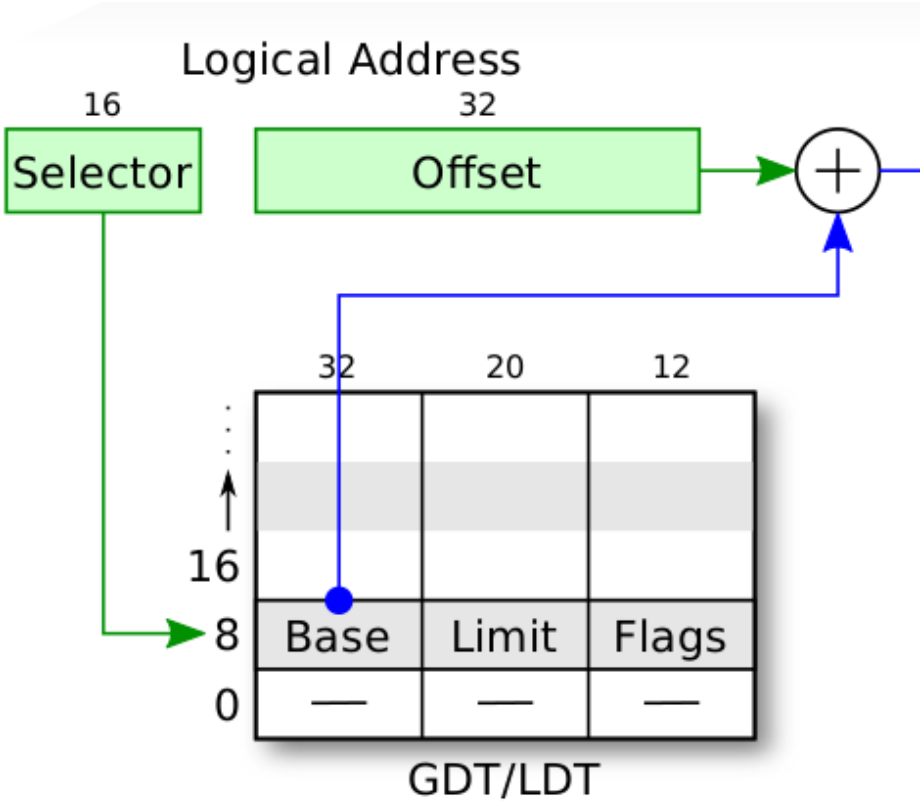
DEMO: Find physical pages (64-bit Linux)

```
$ ./test
/proc/24823/pagemap, buf=0x7ffd224c4e90

$ sudo ./readvirt /proc/24823/pagemap 0x7ffd224c4e90
0x8600000000000ace9, pfn=44265

$ gdb -c /proc/kcore
(gdb) x/30x 0xFFFF880000000000 + 44265 * 4096 + 3728
0xffff88000ace9e90: 0x00 0x01 0x02 0x03 0x04 0x05 0x06 0x07
0xffff88000ace9e98: 0x08 0x09 0x0a 0x0b 0x0c 0x0d 0x0e 0x0f
0xffff88000ace9ea0: 0x10 0x11 0x12 0x13 0x14 0x15 0x16 0x17
0xffff88000ace9ea8: 0x18 0x19 0x1a 0x1b 0x1c 0x1d
```

Segmentation



- Flags?, Selector?
- What's output?

GDT / LDT

- Global Descriptor Table
  - Memory segments which apply to all processes
- Local Descriptor Table
  - Private to each process, used very little in modern OS
- Registers point to GDT or LDT as appropriate (CS, DS, etc.)
- In this case, our GDT entries are a segment descriptor
  - Base, Limit, Permissions, DPL
  - GDT/LDT can hold things other than segment descriptors…
  - TSS (later!)

Segmentation in xv6/bootloader

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

How do we specify to use code/data segments?

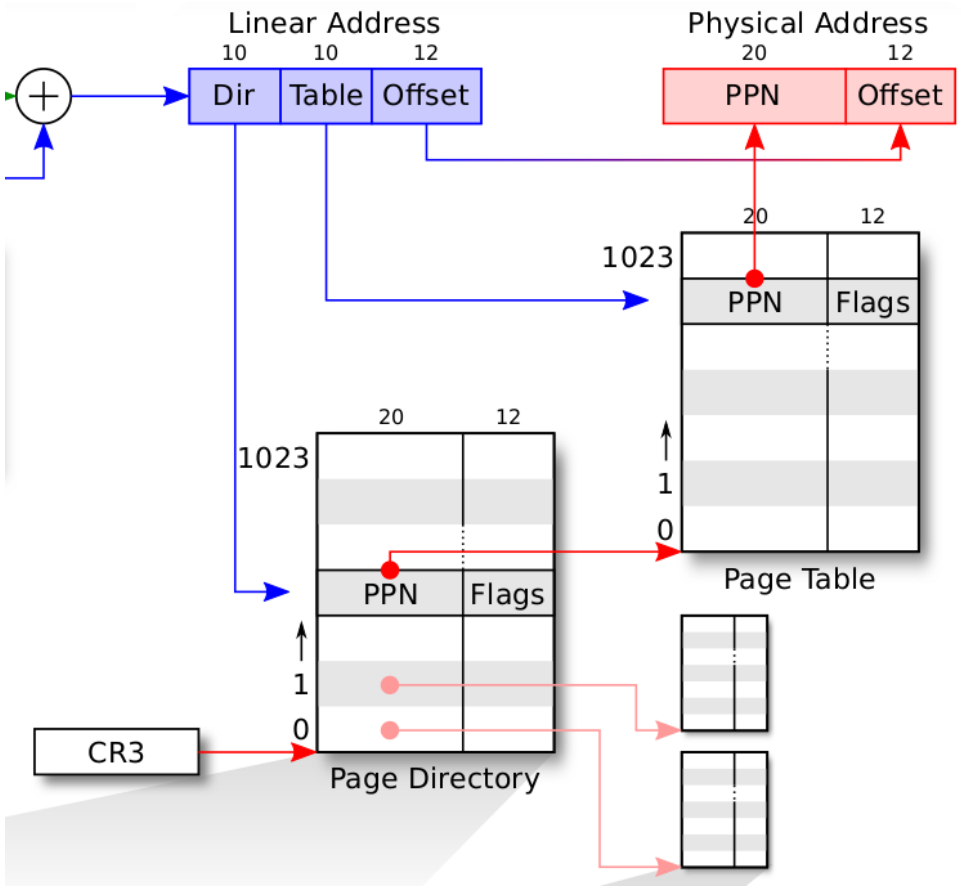
Segmentation in xv6

```
void seginit(void) {
  struct cpu *c = &cpus[cpunum()];
  c=>gdt[SEG_KCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, DPL_KERN);
  c=>gdt[SEG_KDATA] = SEG(STA_W      , 0, 0xffffffff, DPL_KERN);
  c=>gdt[SEG_UCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, DPL_USER);
  c=>gdt[SEG_UDATA] = SEG(STA_W      , 0, 0xffffffff, DPL_USER);

  lgdt(c=>gdt, sizeof(c=>gdt));
  ...
}
```

What happens if userspace ptr points to 0xff000000?

Page Translation



Page Translation

- CR3 should be virtual address? (yes/no)
- We divide a 32 bit address into [dir=10|tbl=10|off=12]
  - [dir=05|tbl=15|off=12]?
  - [dir=15|tbl=05|off=12]?
  - [dir=10|tbl=00|off=22]?
  - [dir=00|tbl=20|off=12]?

Single-level paging

[dir=10|tbl=00|off=22]

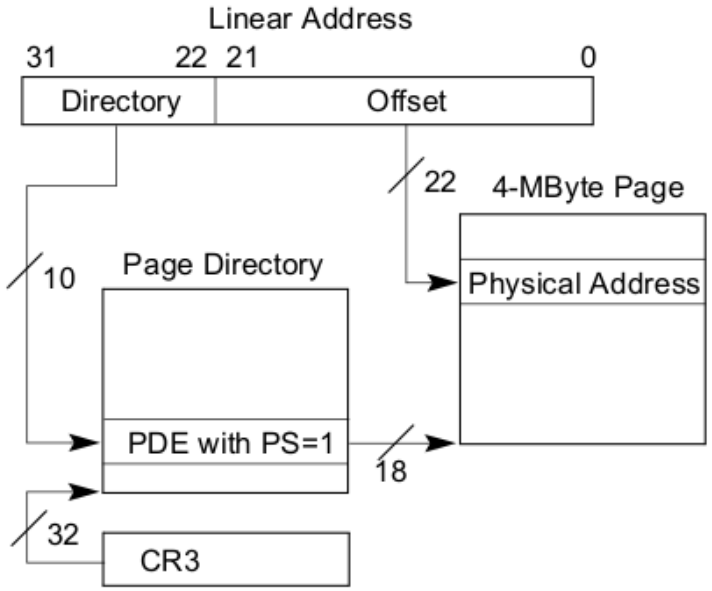


Figure 4-3. Linear-Address Translation to a 4-MByte Page using 32-Bit Paging

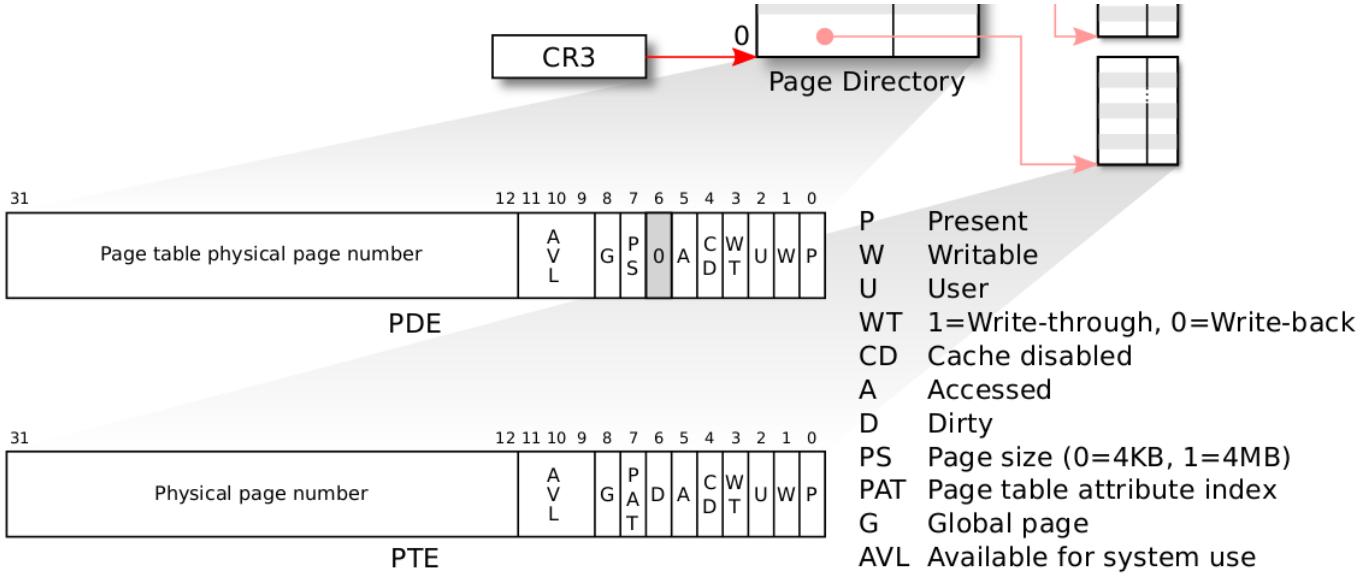
- Q: page size?
- Q: trade-off (better/worse)?

32bit: 4KB, 4MB **64bit**: 4KB, 2MB, 1GB

Single-level paging in xv6

```
// main.c
__attribute__((__aligned__(PGSIZE)))
pde_t entrypgdir[NPDENTRIES] = {
  // Q?
  [0] = (0) | PTE_P | PTE_W | PTE_PS,
  // Q?
  [KERNBASE>>PDXSHIFT] = (0) | PTE_P | PTE_W | PTE_PS,
};
```

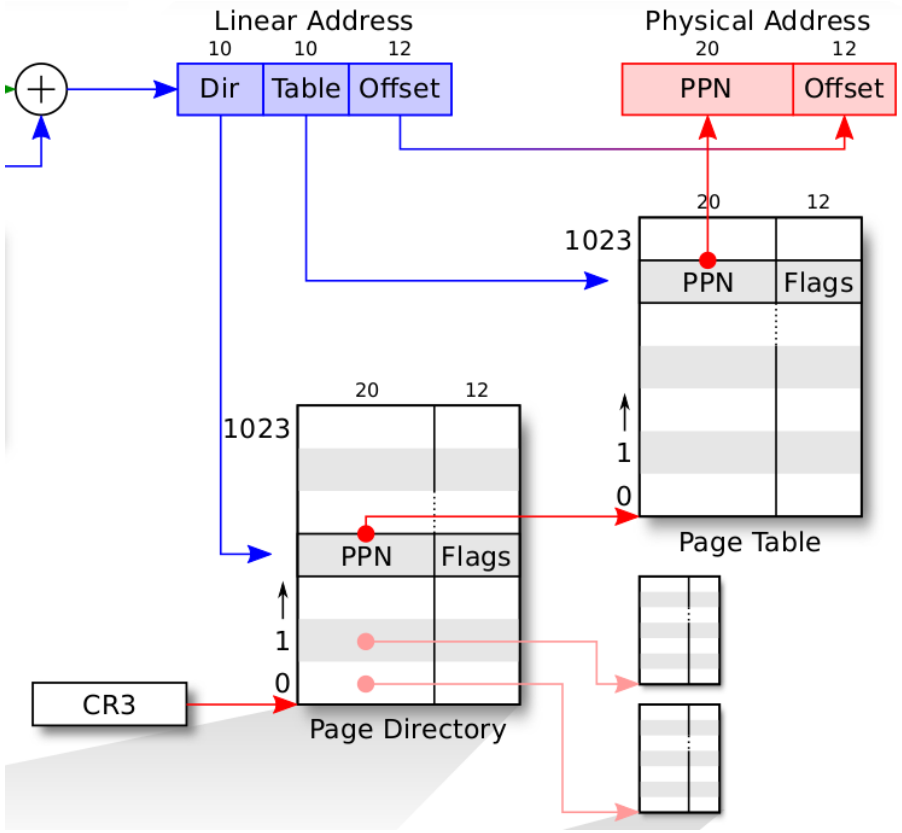
Page Table Entry / Page Directory Entry (PTE/PDE)



PTE/PDE: how to interpret?

```
// main.c
__attribute__((__aligned__(PGSIZE)))
pde_t entrypgdir[NPDENTRIES] = {
  // Q?
  [0] = (0) | PTE_P | PTE_W | PTE_PS,
  // Q?
  [KERNBASE>>PDXSHIFT] = (0) | PTE_P | PTE_W | PTE_PS,
};
```

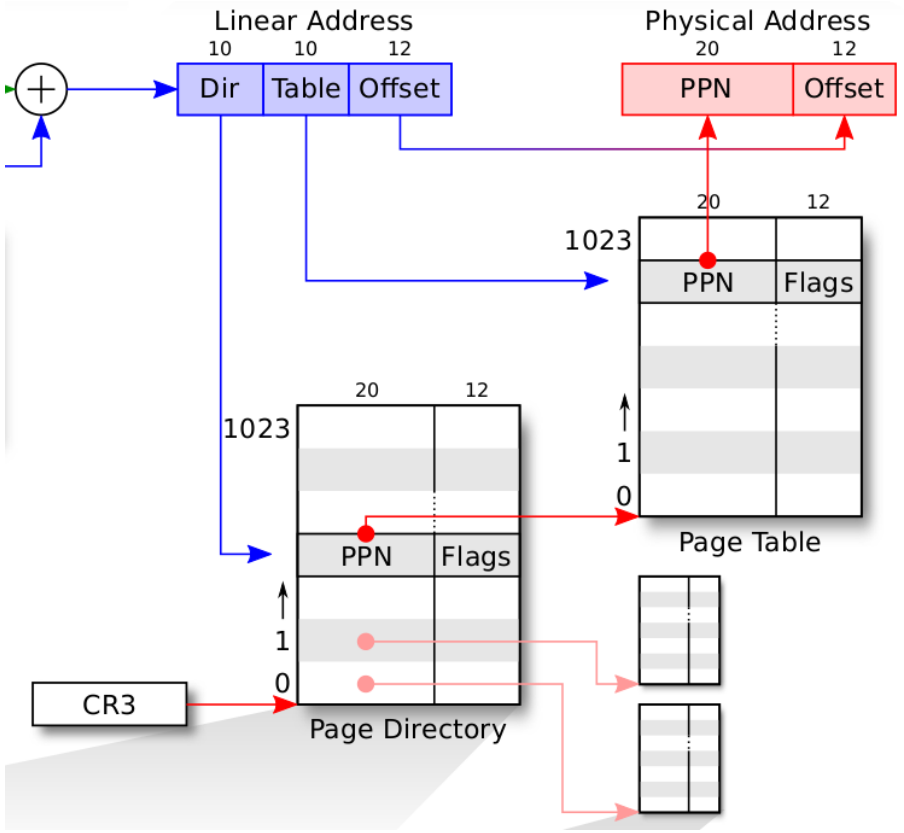
Multi-level paging



- Why not just 4K single-level paging?
  - i.e., 2^20 x size(pte) = ?? MB
- Why problematic?
  - size
  - performance
  - fragmentation

Multi-level paging

Page Translation (two-level)



Given a virtual address, how many memory lookups to translate it to a physical address (aka page walk)?

Multi-level paging

Page Translation (two-level)

Four-level paging in x86-64

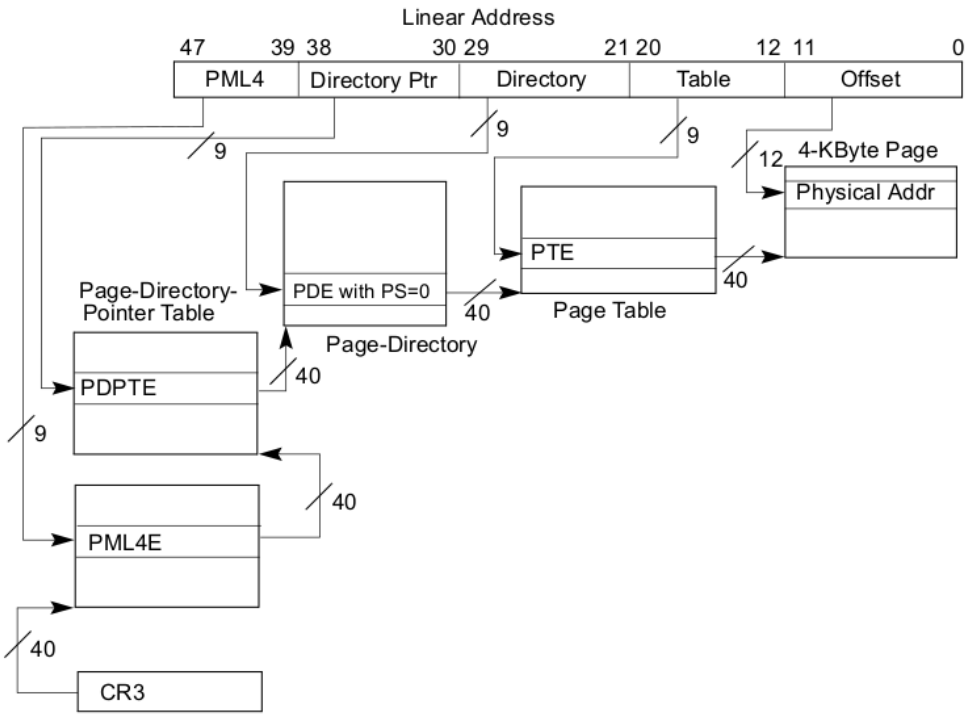


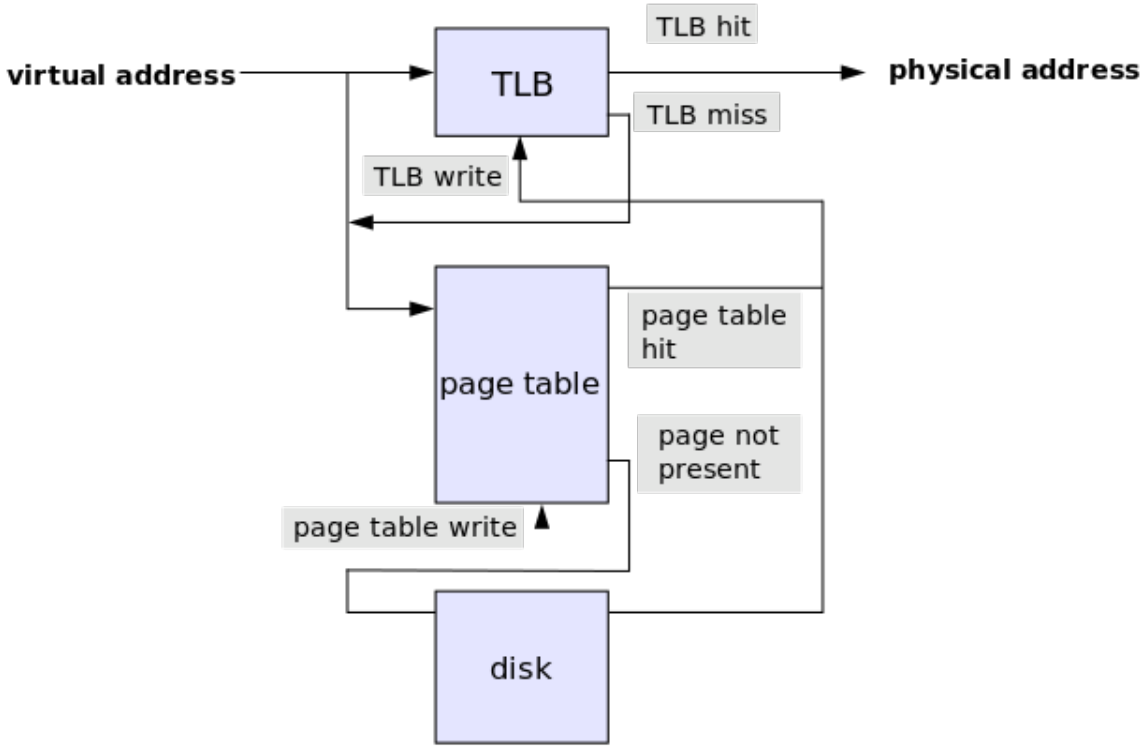
Figure 4-8. Linear-Address Translation to a 4-KByte Page using IA-32e Paging

Same idea, more levels of indirection.

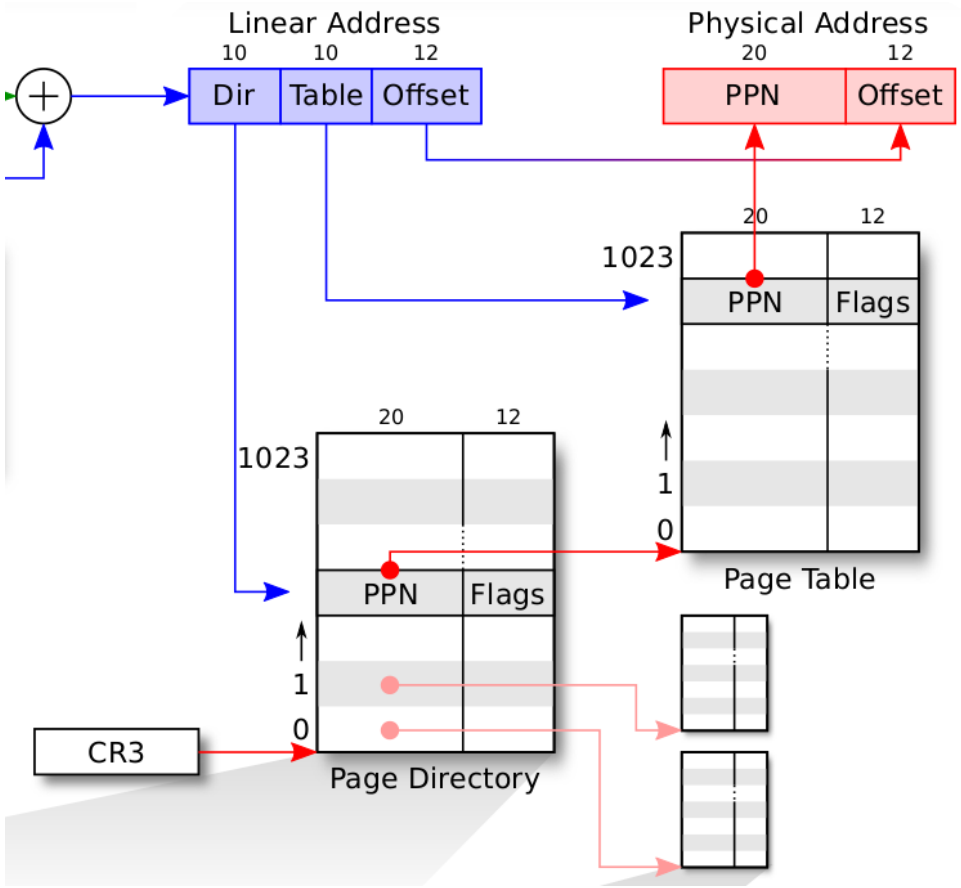
Memory Management Unit (MMU)

- Hardware support: VA -> PA translation
- Cache: TLB (translation lookaside buffer)
  - Caching mappings: virtual -> physical addresses
  - Optimization: iTLB, dTLB

Paging with TLB



What if PTE is not present (P)?



What if we fail to translate? (next week)

So, why is paging good?

So, why is paging good?

Potential applications:

- Kernel tricks (e.g., one zero-filled page)
- Faster system calls (e.g., copy-on-write fork)
- New features (e.g., memory-mapped files)

NOTE: project ideas?

Code: Paging in xv6

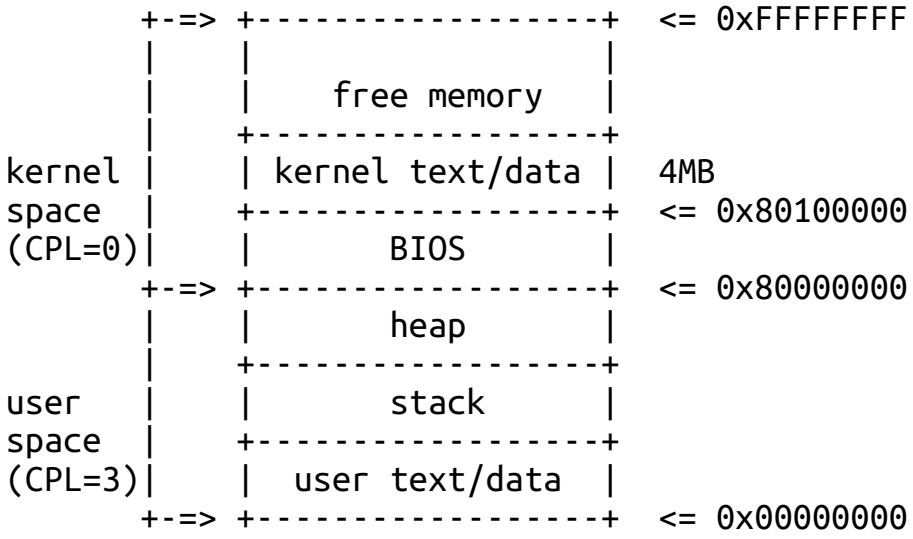
Recall:

bootasm.S (real to protected) -> bootmain (protected, load MBR) -> ...

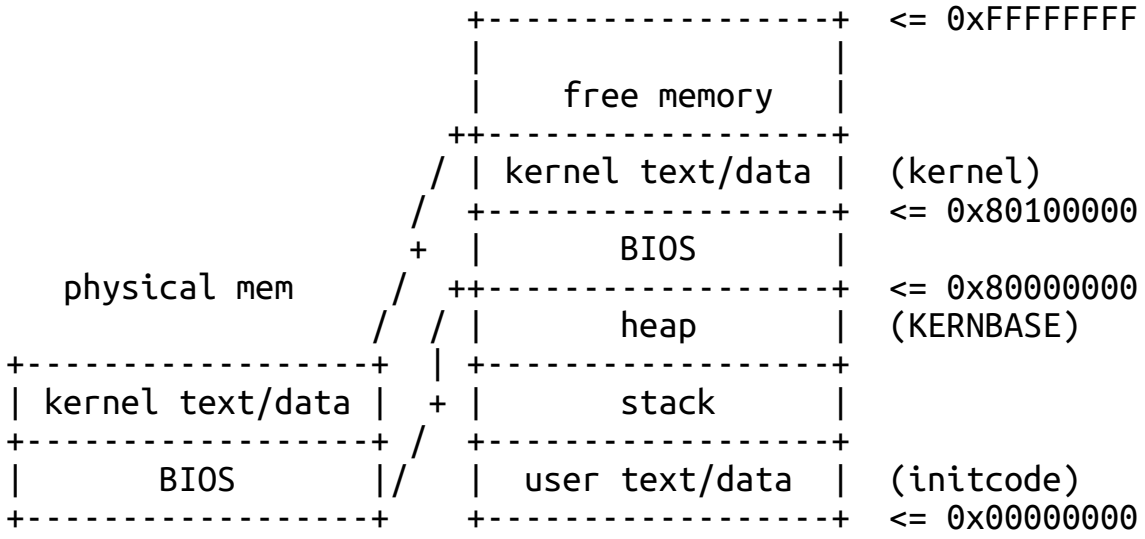
entry -> ... main

- `kinit1()` in `kalloc.c` (physical page allocator)
- `kvmalloc()` in `main.c` (kernel page table)

Virtual address space in xv6



The first address space in xv6



References

- Intel Manual
- UW CSE 451
- OSPP
- MIT 6.828
- Wikipedia
- The Internet