CS3210: Virtual Memory

Lecture 5

Dr. Tim Andersen

Administrivia

- (Sep 21) Lab 3 assigned
- (Sep 29) Lab 2 due
- (Oct 3) Quiz #1. Lab1-2, Ch 0-2, Appendix A/B

Lecture Motivation: Lab 2

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

Physical memory allocator:

- Which physical pages (4096 byte chunks of memory) are free and which are in use?

Virtual Memory:

- How to map virtual addresses to physical memory
- Permissions, x86 Memory Management Unit (MMU), page tables.

Virtual Memory Motivation

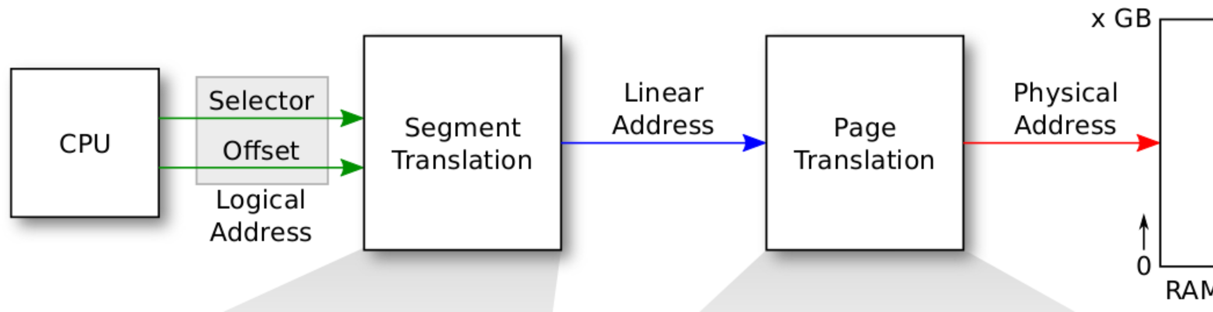
Why Virtual Memory?

Virtual Memory Motivation

Why Virtual Memory?

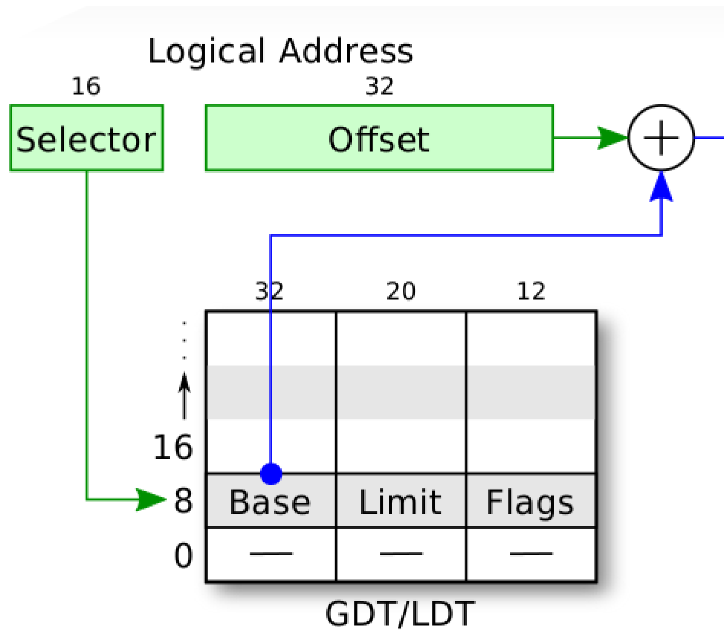
- Primary purpose: isolation
  - Each process has its own address space
  - Also convenient programming model (large continuous address space)
  - Alternatives (segmentation, etc.)
- Benefits:
  - Memory utilization, fragmentation, sharing, etc.
- Level-of-indirection
  - Provides kernel with opportunity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

Big picture: address translation



- Segmentation vs. paging?
- Virtual address?

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?

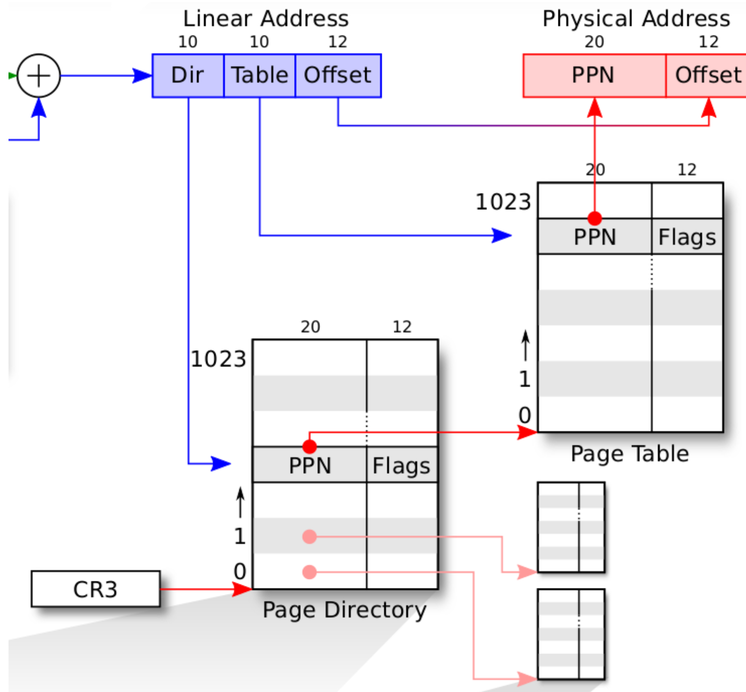
Segmentation in xv6

```
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]

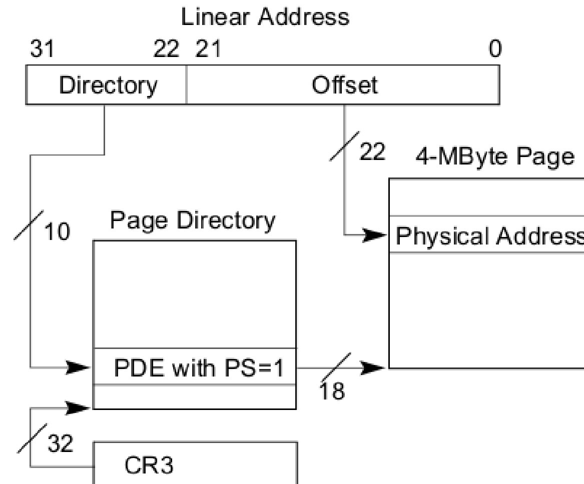


Figure 4-3. Linear-Address Translation to a 4-MByte Page using 32-Bit Paging

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

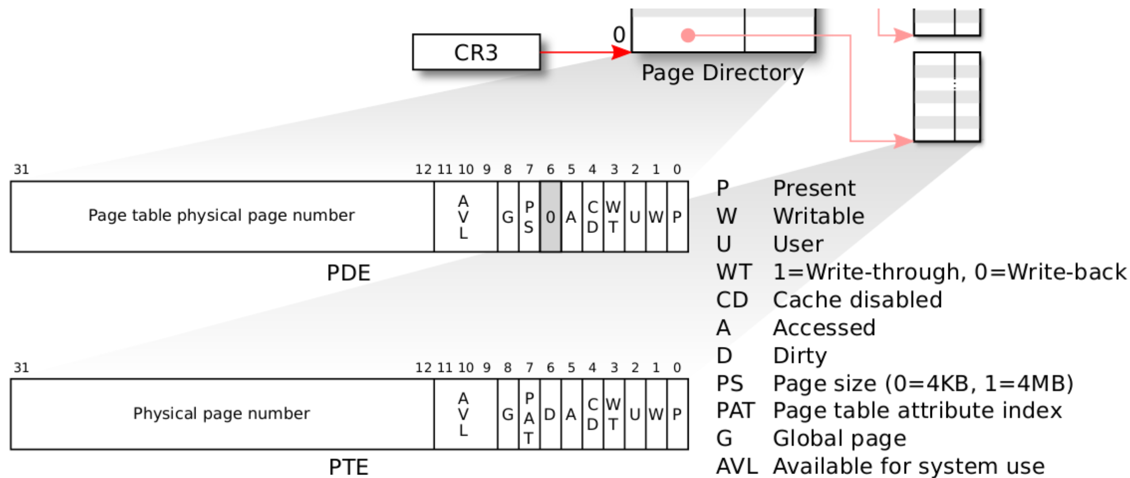
32bit: 4KB, 4MB **64bit**: 4KB, 2MB, 1GB

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

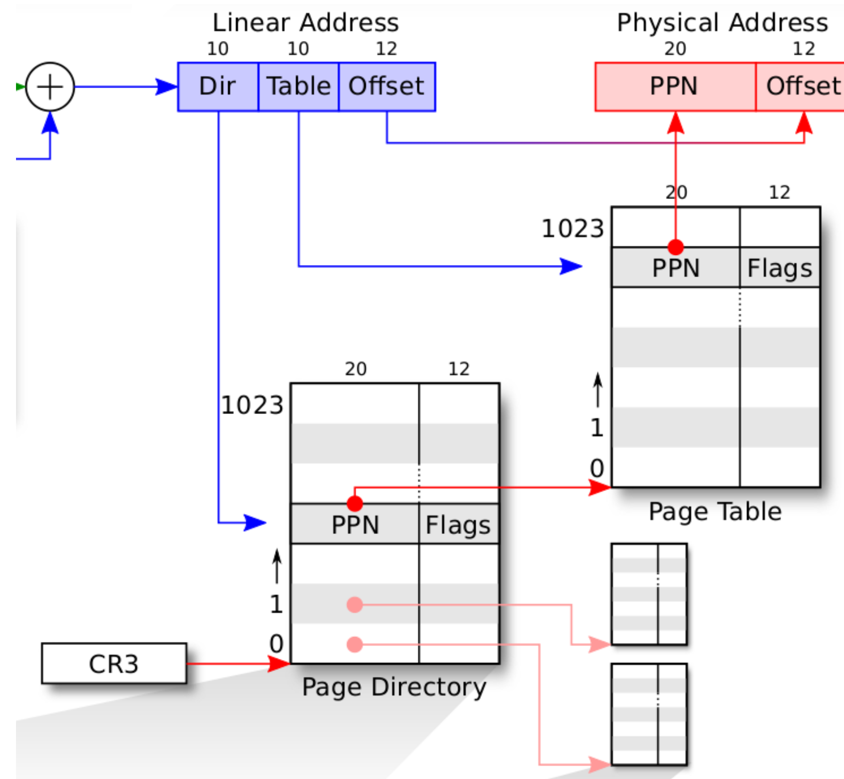
(PTE/PDE)



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,
};
```

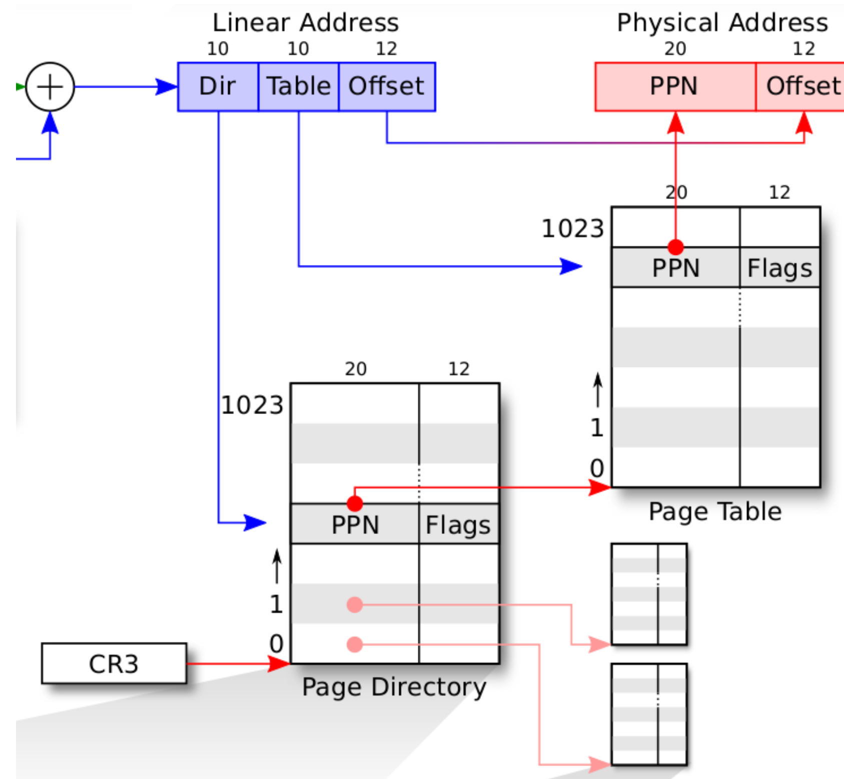
Multi-level paging



- 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

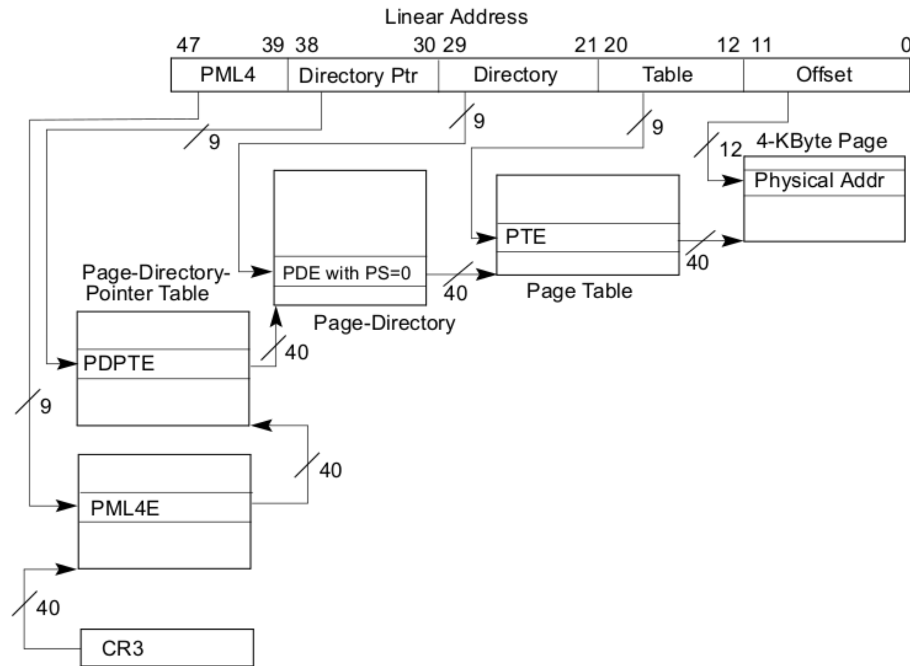


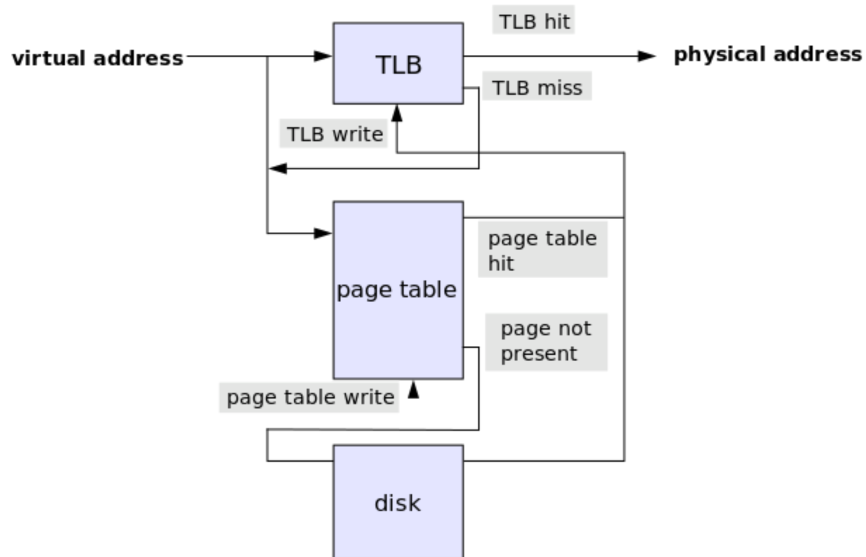
Figure 4-8. Linear-Address Translation to a 4-KByte Page using IA-32e Paging

Same idea, more levels of indirection.

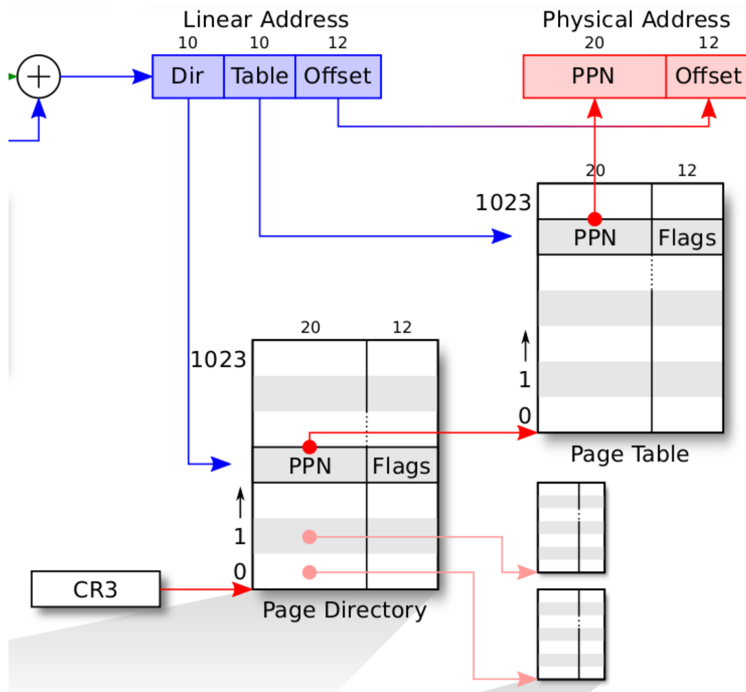
Memory Management Unit (MMU)

- 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

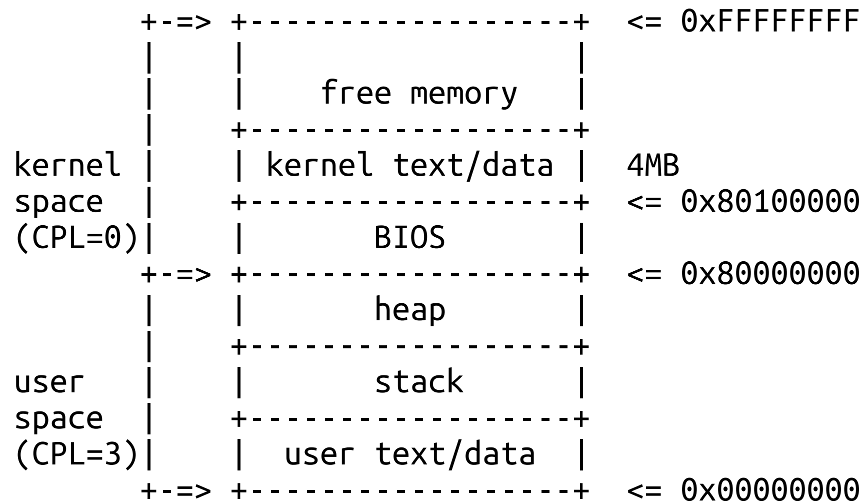
Recall:

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

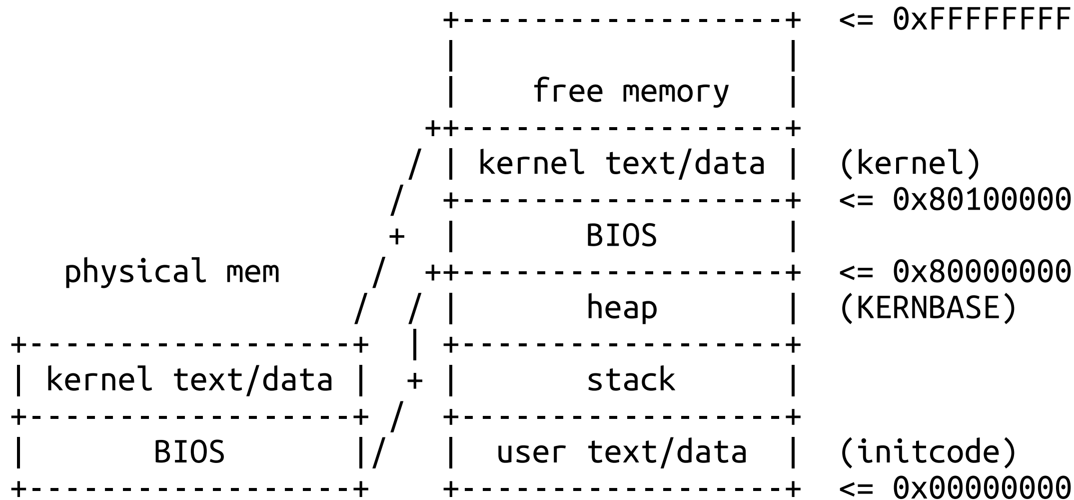
entry -> … main

- `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



Physical Page Management in xv6

Boot Allocation in entry.S

```
# Entering xv6 on boot processor, with paging off.
 .globl entry
 entry:
   # Turn on page size extension for 4Mbyte pages
   movl    %cr4, %eax
   orl     $(CR4_PSE), %eax
   movl    %eax, %cr4
   # Set page directory
   movl    $(V2P_WO(entrypgdir)), %eax
   movl    %eax, %cr3
   # Turn on paging.
   movl    %cr0, %eax
   orl     $(CR0_PG|CR0_WP), %eax
   movl    %eax, %cr0
```

- xv6 uses a single 4M superpage with entrypgdir as the directory to bootstrap the kernel
- JOS uses a function called boot_alloc that allocates enought 4K pages to hold `n` bytes.

Physical Page Management in xv6

```
// Initialization happens in two phases.
// 1. main() calls kinit1() while still using entrypgdir to place just
// the pages mapped by entrypgdir on free list.
// 2. main() calls kinit2() with the rest of the physical pages
// after installing a full page table that maps them on all cores.
void
kinit1(void *vstart, void *vend)
{
  initlock(&kmem.lock, "kmem");
  kmem.use_lock = 0;
  freerange(vstart, vend);
}
```

```
void
freerange(void *vstart, void *vend)
{
  char *p;
  p = (char*)PGROUNDUP((uint)vstart);

  for(; p + PGSIZE <= (char*)vend; p += PGSIZE)
    kfree(p);
}
```

Physical Page Management in xv6

- JOS tracks free pages with a linked list of `struct PageInfo` objects.
- xv6 embeds the page info in the free pages (no page sharing)
- kfree in kalloc.c adds to the linked list of free pages.

```c
// Free the page of physical memory pointed at by v, which normally should
// have been returned by a call to kalloc().  (The exception is when
// initializing the allocator; see kinit above.)
void
kfree(char *v)
{
  struct run *r;

  if((uint)v % PGSIZE || v < end || V2P(v) >= PHYSTOP)
    panic("kfree");

  // Fill with junk to catch dangling refs.
  memset(v, 1, PGSIZE);


  if(kmem.use_lock)
    acquire(&kmem.lock);
  r = (struct run*)v;
  r->next = kmem.freelist;
  kmem.freelist = r;
  if(kmem.use_lock)
    release(&kmem.lock);
}
```

Page Table Management in xv6

From vm.c:

```
// This table defines the kernel's mappings, which are present in
 // every process's page table.
 static struct kmap {
   void *virt;
   uint phys_start;
   uint phys_end;
   int perm;
 } kmap[] = {
  { (void*)KERNBASE, 0,             EXTMEM,    PTE_W}, // I/O space
  { (void*)KERNLINK, V2P(KERNLINK), V2P(data), 0},     // kern text+rodata
  { (void*)data,     V2P(data),     PHYSTOP,   PTE_W}, // kern data+memory
  { (void*)DEVSPACE, DEVSPACE,      0,         PTE_W}, // more devices
 };
```

Page Table Management in xv6

- The kernel part of the page table is set up

```
// Set up kernel part of a page table.
 pde_t*
 setupkvm(void)
 {
   pde_t *pgdir;
   struct kmap *k;

   if((pgdir = (pde_t*)kalloc()) == 0)
     return 0;
   memset(pgdir, 0, PGSIZE);
   if (P2V(PHYSTOP) > (void*)DEVSPACE)
     panic("PHYSTOP too high");
   for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
     if(mappages(pgdir, k->virt, k->phys_end - k->phys_start,
                 (uint)k->phys_start, k->perm) < 0) {
       freevm(pgdir);
       return 0;
     }
   return pgdir;
 }
```

Page Table Management in xv6

- xv6 has a function that maps pages (sets up Page Table Entries)
- walkpgdir finds PTE from a virtual address (what the x86 hardware does)
- Does similar task to `boot_map_region` in JOS.

```
static int
mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
{
  char *a, *last;
  pte_t *pte;

  a = (char*)PGROUNDDOWN((uint)va);
  last = (char*)PGROUNDDOWN(((uint)va) + size - 1);
  for(;;){
    if((pte = walkpgdir(pgdir, a, 1)) == 0)
      return -1;
    if(*pte & PTE_P)
      panic("remap");
    *pte = pa | perm | PTE_P;

    if(a == last)
      break;
    a += PGSIZE;
    pa += PGSIZE;
  }
  return 0;
}
```

Page Walking in xv6

- Simpler than JOS because no pp_ref!

```
static pte_t *
walkpgdir(pde_t *pgdir, const void *va, int alloc)
{
  pde_t *pde;
  pte_t *pgtab;

  pde = &pgdir[PDX(va)];
  if(*pde & PTE_P){
    pgtab = (pte_t*)P2V(PTE_ADDR(*pde));
  } else {
    if(!alloc || (pgtab = (pte_t*)kalloc()) == 0)
      return 0;
    // Make sure all those PTE_P bits are zero.
    memset(pgtab, 0, PGSIZE);
    // The permissions here are overly generous, but they can
    // be further restricted by the permissions in the page table
    // entries, if necessary.
    *pde = V2P(pgtab) | PTE_P | PTE_W | PTE_U;
  }
  return &pgtab[PTX(va)];
}
```

JOS vs. xv6

- xv6 has no ability to have multiple virtual pages map to a single page
    - Makes Copy-on-Write almost impossible
    - Shared memory mapping difficult
- xv6 expects the kernal to map all of physical memory (up to 2GB) including all user programs
    - Any user program can map virtual pages to free physical pages
    - Once the kernel and a user program (at most) have a mapping to a physical page, the physical page can have no more mappings.
    - Page mapping is simple and binary (free or used)
- JOS can have many user programs map virtual pages to the same physical page
    - Enables copy-on-write support
    - Shared libraries need only be in one place in physical memory and mapped over and over
    - Means that page references must be tracked

References

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