CS3210: Virtual memory applications

Taesoo Kim

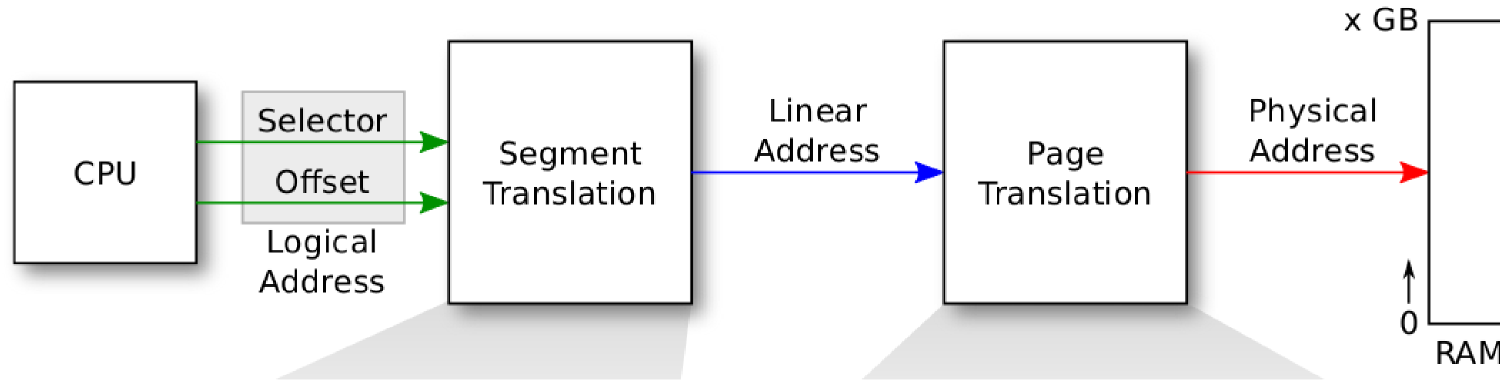
Administrivia

- Lab schedule
  - No Lab 6 (sad, but bonus pt!)
  - One extra week for Lab 4 (part A)
- (Feb 23) Quiz #1. Lab1-3, Ch 0-2, Appendix A/B
  - Open book/laptop
  - No Internet
- (Feb 25) Time to brainstorm project ideas!!
  - Prep question: submit 1-page pre-proposal (by Feb 24, 10pm)

Summary of last lectures

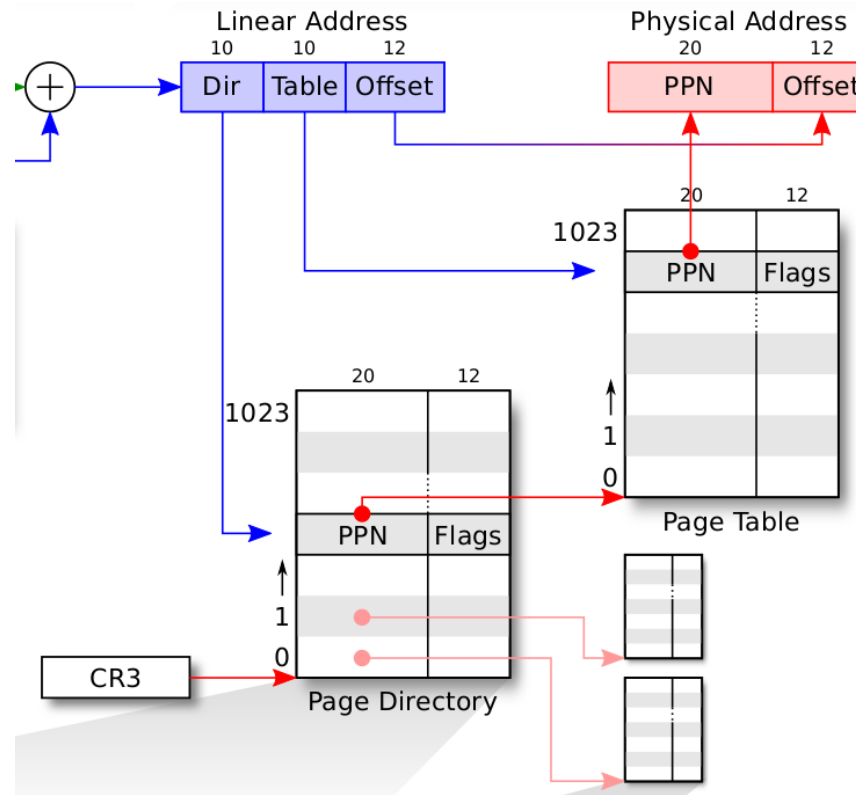
- Power-on → BIOS → bootloader → kernel → init (+ user bins)
- OS: abstraction, multiplexing, isolation, sharing
- OS design: monolithic (xv6) vs. micro kernels (jos)
- Isolation mechanisms
  - CPL (aka ring), address space (aka process)
  - Virtual memory, paging

Recap: address translation



- Q: what are the **advantanges** of the address translation?
- Q: what are the **disadvantanges** of the address translation?

Recap: page translation



Recap: design trade-off

- We devide a 32 bit address into [dir=10|tbl=10|off=12]
  - [dir=00|tbl=20|off=12]?
  - [dir=10|tbl=00|off=22]?
  - [dir=05|tbl=15|off=12]?
  - [dir=15|tbl=05|off=12]?
- Q: what's "super page"? good or bad?

So, why paging is good?

- Primary purpose: isolation
  - each process has its own address space
- Benefits:
  - memory utilization, fragmentation, sharing, etc.
- Level-of-indirection
  - provides kernel with opportunity to do cool stuff

Today: 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 idea?

Key idea: interposition

```
#define PTE_P           0x001   // Present
#define PTE_W           0x002   // Writeable
#define PTE_U           0x004   // User
#define PTE_PWT         0x008   // Write-Through
#define PTE_PCD         0x010   // Cache-Disable
#define PTE_A           0x020   // Accessed (Q?)
#define PTE_D           0x040   // Dirty (Q?)
#define PTE_PS          0x080   // Page Size
```

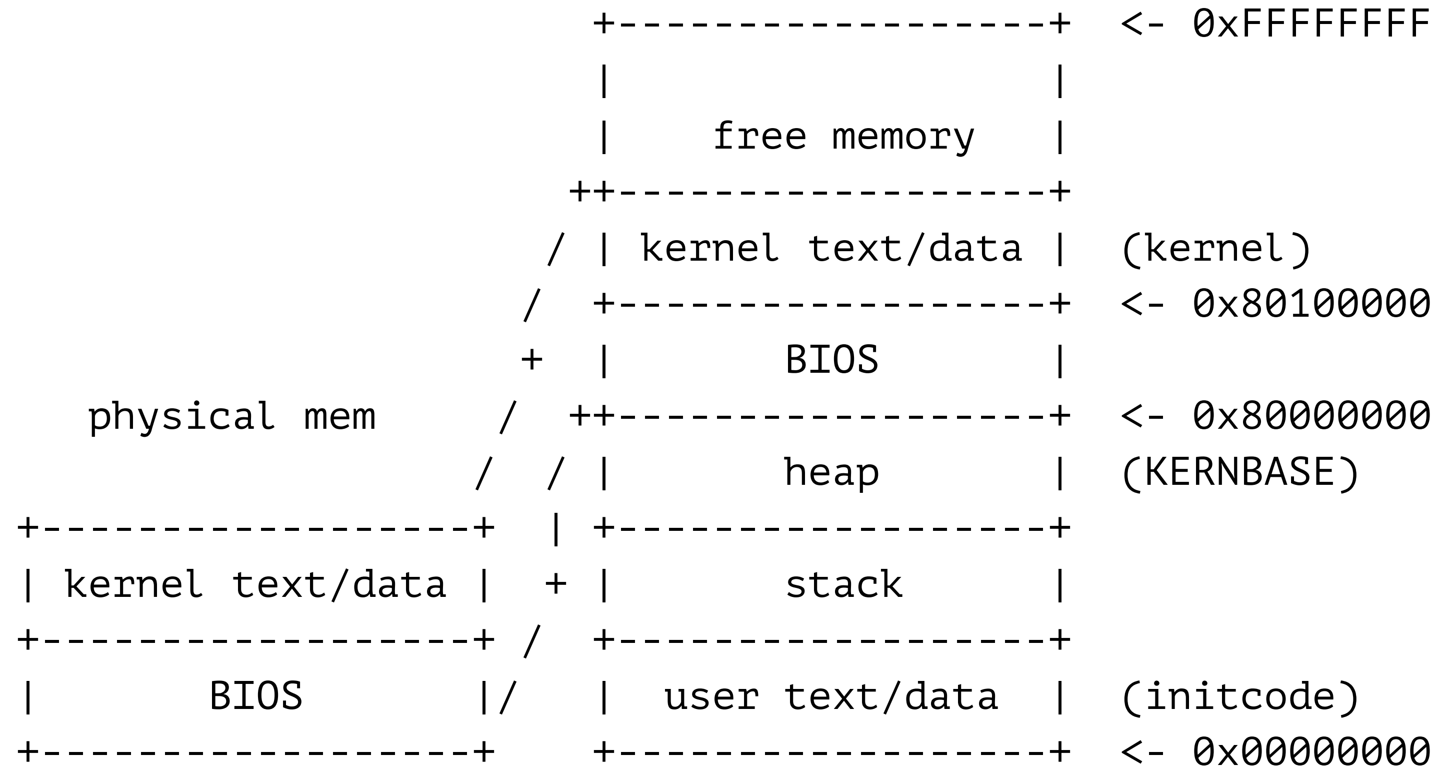
- Q: what if PTE is not present (P)?
- Q: what if a process attempts to write to non-writable memory?
- Q: what are these options for?

Code: paging in xv6 (once more)

- `entry()` in `entry.S`
- `kinit1()` in `main.c`
- `kvmalloc()` in `main.c`

```
$ cat /proc/iomem
00000000-00000fff : reserved
00001000-0009cfff : System RAM
0009d000-0009ffff : reserved
...
```

The first address space in xv6



Protection: preventing NULL dereference

- Q: what's NULL dereference? how serious? in xv6? (Linux exploit)
- NULL pointer dereference exception
  - Q: how would you implement this for Java, say `obj->field`
  - Trick: put a non-mapped page at VA zero
    - Useful for catching program bugs
    - Q: limitations?

Protection: preventing stack overflow

- Q: what's stack overflow? how serious? in xv6? (check cs6265!)
- "Toyota's major stack mistakes" (see Michael Barr's Bookout v. Toyota)
  - Trick: put a non-mapped page right below user stack
  - JOS: `inc/memlayout.h`

```
UTOP,UENVS ------>  +------------------------------+ 0xeec00000
UXSTACKTOP -/       |     User Exception Stack     | RW/RW  PGSIZE
                    +------------------------------+ 0xeebff000
                    |       Empty Memory (*)       | --/--  PGSIZE
   USTACKTOP  --->  +------------------------------+ 0xeebfe000
                    |      Normal User Stack       | RW/RW  PGSIZE
                    +------------------------------+ 0xeebfd000
```

Feature: "virtual" memory

- Q: can we run an app. requiring > 2GB in xv6?
- Q: what about an app. requiring > 1GB on a machine with 512MB?

Feature: "virtual" memory

- Applications often need more memory than physical memory
  - early days: two floppy drives
  - strawman: applications store part of state to disk and load back later
  - hard to write applications
- Virtual memory: offer the illusion of a large, continuous memory
  - swap space: OS pages out some pages to disk transparently
  - distributed shared memory: access other machines' memory across network

Feature: "virtual" memory

```
$ free
       total  used  free  shared  buff/cache   available
Mem:     19G  5.1G  424M    1.4G         13G         12G
Swap:     0B    0B    0B
```

Feature: memory-mapped files

- Q: what's benefit of having `open()`, `read()`, `write()`?
- `mmap()`: map files, read/write files like memory
- Simple programming interface, memory read/write
- Avoid data copying: e.g., send an mmaped file to network
  - compare to using `read`/`write`
  - no data transfer from kernel to user
- Q: when to page-in/page-out content?

Feature: single zero page

- Q: `calloc()`? `memset(buf, 0, buflen)`?
- Often need to allocate a page with zeros to start with
- Trick: keep *one* zero page for all such pages
- Q: what if one process writes to the page?

Feature: copy-on-write (CoW) fork (Lab 4)

- Q: what's `fork()`? and what happens when forking?
- Observation: child and parent share most of the data
  - mark pages as copy-on-write
  - make a copy on page fault
- Other sharing
  - multiple guest OSes running inside the same hypervisor
  - shared objects: `.so`/`.dll` files

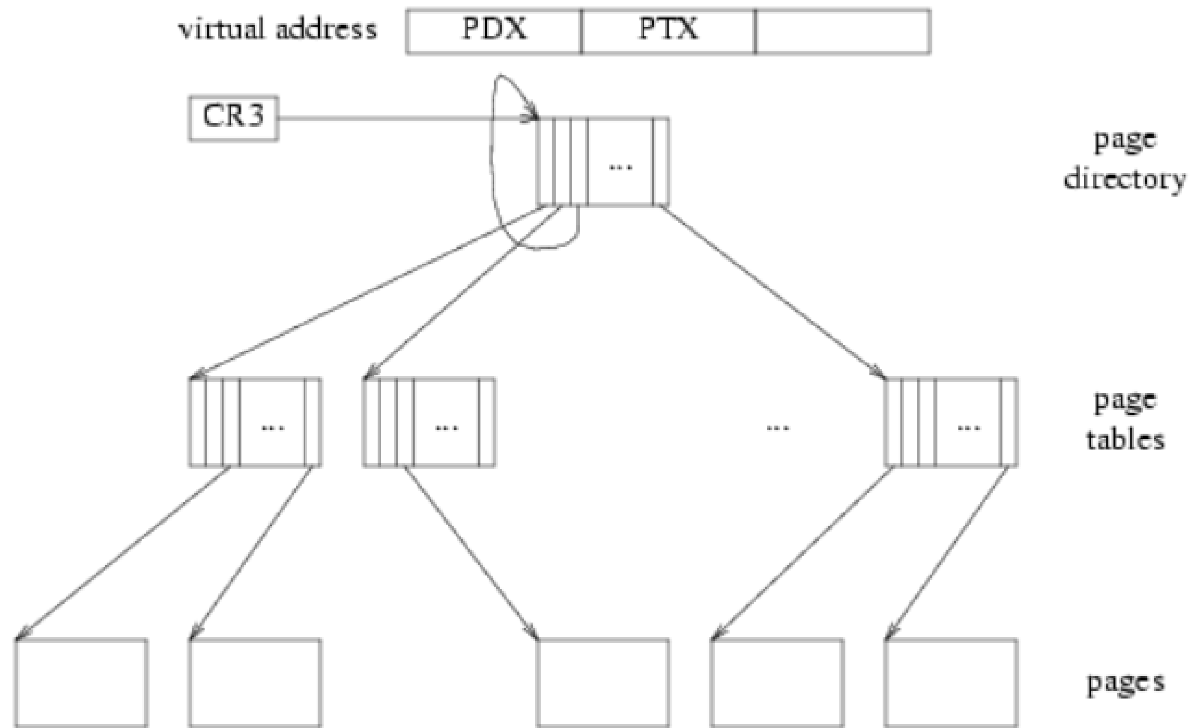
Feature: virtual *linear* page tables

- Q: how big is the page table if we have a single level (4KB pages)?
- Q: how to make all page tables show up on our address space?

Feature: virtual *linear* page tables

- `uvpt[n]` gives the PTE of page `n`
    - Self mapping: set one PDE to point to the page directory
    - CPU walks the tree as usual, but ends up in one level up

Feature: virtual *linear* page tables



Next tutorial

- Lazy allocation
- Grow stack on demand

References

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