CS3210: Virtual Memory Applications

Kyle Harrigan

Administrivia

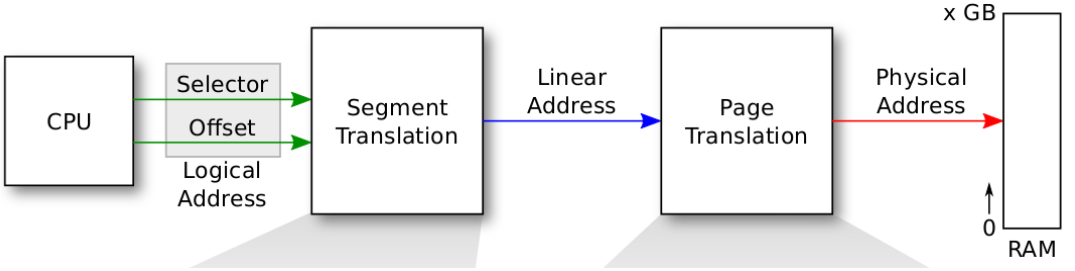
- (Feb 10) Lab 2 Due (This Friday!)
- (Feb 16) Quiz #1. Lab1-2, Ch 0-2, Appendix A/B
  - Open book/laptop
  - No Internet
- (Feb 20) Project Pre-Proposal Due (1 Page)
  - We will go over requirement next week
- (Feb 24) Lab 3 (Part A) Due
- We are investigating the inconsistent readvirt behavior. Hang tight, we will do something reasonable for grading.

Administrivia

- (Feb 10) Lab 2 Due (This Friday!)
- (Feb 16) Quiz #1. Lab1-2, Ch 0-2, Appendix A/B
  - Open book/laptop
  - No Internet
- (Feb 20) Project Pre-Proposal Due (1 Page)
  - We will go over requirement next week
- (Feb 24) Lab 3 (Part A) Due
- We are investigating the inconsistent readvirt behavior. Hang tight, we will do something reasonable for grading.

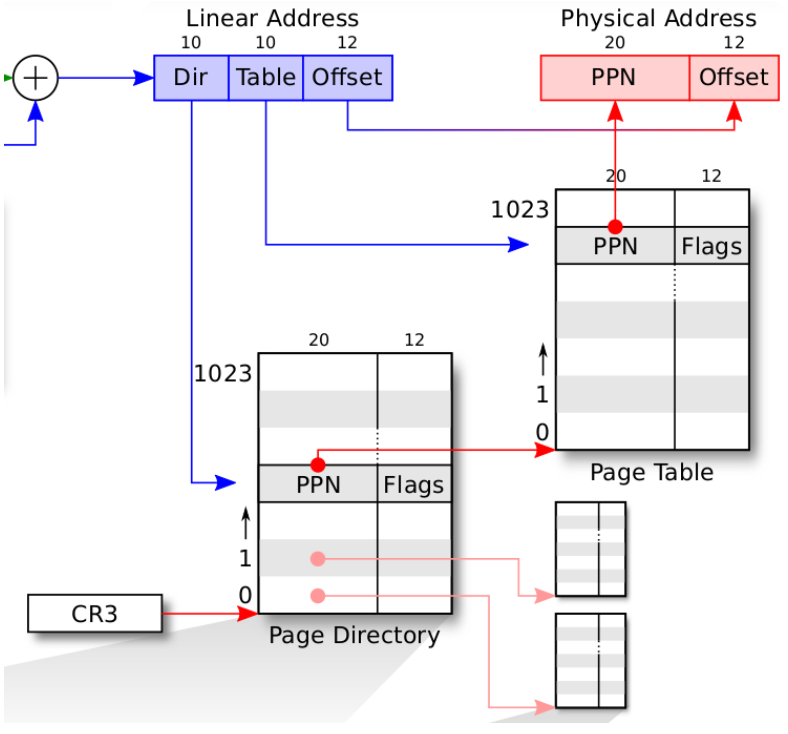
Questions?

Recap: address translation



- What are the **advantanges** of the address translation?
- What are the **disadvantanges** of the address translation?

Recap: page translation



Recap: design trade-off

- We divide 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]?
- What's "super page"? good or bad?

Why is paging useful?

- 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

Lab2 Selected Topics

- Function overview - Go through each function and explain a little about what it is supposed to do
- `memlayout.h` - Discuss virtual memory layout. We will reference this from various slides in this section.
- Discuss some pitfalls and misconceptions

Function Overview (Part 1)

```
boot_alloc()
mem_init() (only up to the call to check_page_free_list(1))
page_init()
page_alloc()
page_free()
```

Function Overview (Part 1)

```
boot_alloc()
mem_init() (only up to the call to check_page_free_list(1))
page_init()
page_alloc()
page_free()
```

boot_alloc: *Initial physical memory allocator. Allocate continguous blocks of physical memory, store in nextfree*

Function Overview (Part 1)

```
boot_alloc()
mem_init() (only up to the call to check_page_free_list(1))
page_init()
page_alloc()
page_free()
```

boot_alloc: *Initial physical memory allocator. Allocate continguous blocks of physical memory, store in nextfree*

mem_init: *Setup kernel address space (above UTOP)*

Function Overview (Part 1)

```
boot_alloc()
mem_init() (only up to the call to check_page_free_list(1))
page_init()
page_alloc()
page_free()
```

boot_alloc: *Initial physical memory allocator. Allocate continguous blocks of physical memory, store in nextfree*

mem_init: *Setup kernel address space (above UTOP)*

page_init: *Initialize pages structure and memory free list.*

Function Overview (Part 1)

```
boot_alloc()
mem_init() (only up to the call to check_page_free_list(1))
page_init()
page_alloc()
page_free()
```

boot_alloc: *Initial physical memory allocator. Allocate continguous blocks of physical memory, store in nextfree*

mem_init: *Setup kernel address space (above UTOP)*

page_init: *Initialize pages structure and memory free list.*

page_alloc: *Allocate a physical page, zero fill.*

Function Overview (Part 1)

```
boot_alloc()
mem_init() (only up to the call to check_page_free_list(1))
page_init()
page_alloc()
page_free()
```

boot_alloc: *Initial physical memory allocator. Allocate continguous blocks of physical memory, store in nextfree*

mem_init: *Setup kernel address space (above UTOP)*

page_init: *Initialize pages structure and memory free list.*

page_alloc: *Allocate a physical page, zero fill.*

page_free: *Free a page*

Function overview (Part 2)

```
pgdir_walk()
boot_map_region()
page_lookup()
page_remove()
page_insert()
```

Function overview (Part 2)

```
pgdir_walk()
boot_map_region()
page_lookup()
page_remove()
page_insert()
```

pgdir_walk(): *Walks the page directory. Given a linear address, return a page table entry*

Function overview (Part 2)

```
pgdir_walk()
boot_map_region()
page_lookup()
page_remove()
page_insert()
```

pgdir_walk(): *Walks the page directory. Given a linear address, return a page table entry*

boot_map_region(): *Map [va, va+size] to [pa, pa+size] by creating a page table entry, with particular permissions*.

Function overview (Part 2)

```
pgdir_walk()
boot_map_region()
page_lookup()
page_remove()
page_insert()
```

pgdir_walk(): *Walks the page directory. Given a linear address, return a page table entry*

boot_map_region(): *Map [va, va+size] to [pa, pa+size] by creating a page table entry, with particular permissions*.

page_lookup(): *Return page mapped at va, or NULL if no page mapped.*

Function overview (Part 2)

```
pgdir_walk()
boot_map_region()
page_lookup()
page_remove()
page_insert()
```

pgdir_walk(): *Walks the page directory. Given a linear address, return a page table entry*

boot_map_region(): *Map [va, va+size] to [pa, pa+size] by creating a page table entry, with particular permissions*.

page_lookup(): *Return page mapped at va, or NULL if no page mapped.*

page_remove(): *Unmap page at virtual address va*

Function overview (Part 2)

```
pgdir_walk()
boot_map_region()
page_lookup()
page_remove()
page_insert()
```

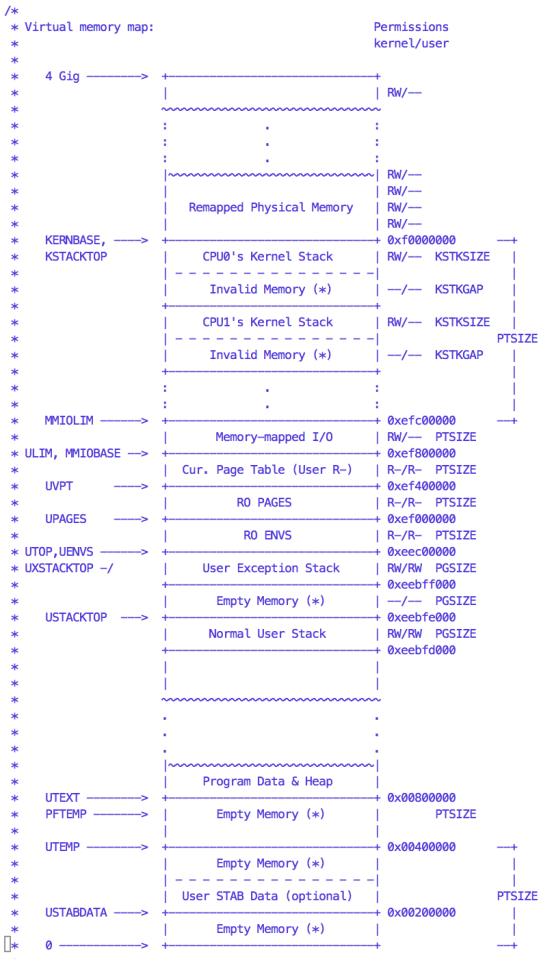
pgdir_walk(): *Walks the page directory. Given a linear address, return a page table entry*

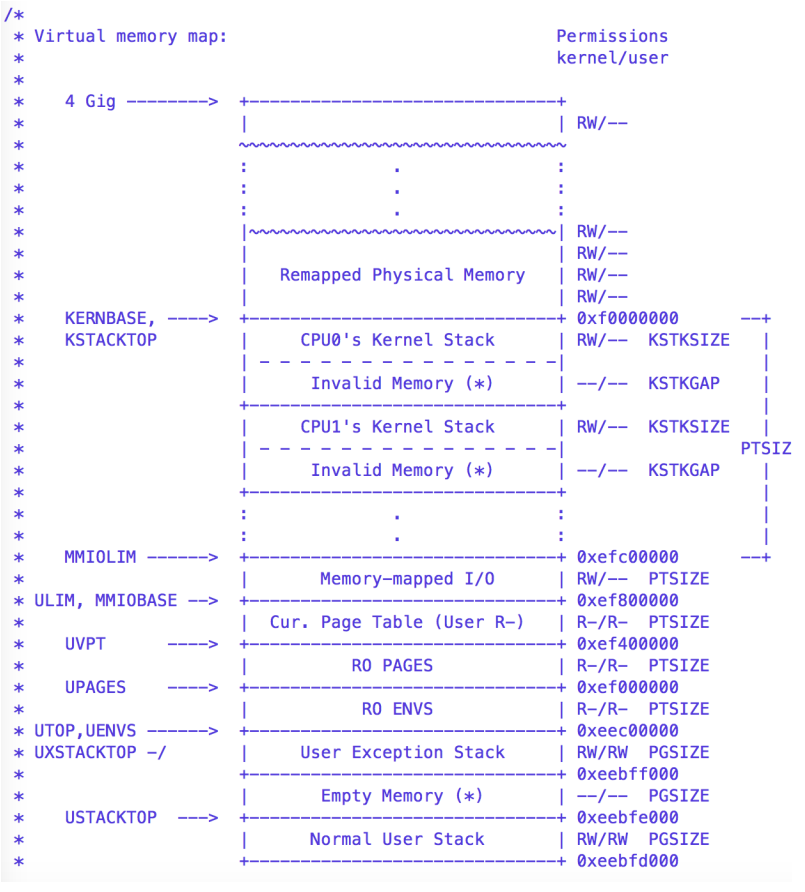
boot_map_region(): *Map [va, va+size] to [pa, pa+size] by creating a page table entry, with particular permissions*.

page_lookup(): *Return page mapped at va, or NULL if no page mapped.*

page_remove(): *Unmap page at virtual address va*

page_insert(): *Map the physical page 'pp' at virtual address 'va'*


/*
* Virtual memory map:
Permissions
kernel/user
4 Gig
RW/--
Remapped Physical Memory
KERNBASE,
KSTACKTOP
CPU0's Kernel Stack
RW/-- KSTKSIZE
Invalid Memory (*)
--/-- KSTKGAP
CPU1's Kernel Stack
RW/-- KSTKSIZE
Invalid Memory (*)
--/-- KSTKGAP
PTSIZE
MMIOLIM
0xefc00000
Memory-mapped I/O
RW/-- PTSIZE
ULIM, MMIOBASE
0xef800000
Cur. Page Table (User R-)
R-/R- PTSIZE
UVPT
0xef400000
RO PAGES
UPAGES
0xef000000
RO ENVS
UTOP,UENVS
0xeec00000
UXSTACKTOP -/
User Exception Stack
RW/RW PGSIZE
0xeebff000
Empty Memory (*)
--/-- PGSIZE
USTACKTOP
0xeebfe000
Normal User Stack
RW/RW PGSIZE
0xeebfd000
Program Data & Heap
UTEXT
0x00800000
PFTEMP
Empty Memory (*)
PTSIZE
UTEMP
0x00400000
Empty Memory (*)
User STAB Data (optional)
USTABDATA
0x00200000
Empty Memory (*)
0


/*
* Virtual memory map:
Permissions
kernel/user
4 Gig
RW/--
Remapped Physical Memory
KERNBASE,
KSTACKTOP
CPU0's Kernel Stack
RW/-- KSTKSIZE
Invalid Memory (*)
--/-- KSTKGAP
CPU1's Kernel Stack
RW/-- KSTKSIZE
Invalid Memory (*)
--/-- KSTKGAP
PTSIZ
MMIOLIM
0xefc00000
Memory-mapped I/O
RW/-- PTSIZE
ULIM, MMIOBASE
0xef800000
Cur. Page Table (User R-)
R-/R- PTSIZE
UVPT
0xef400000
RO PAGES
UPAGES
0xef000000
RO ENVS
UTOP,UENVS
0xeec00000
UXSTACKTOP -/
User Exception Stack
RW/RW PGSIZE
0xeebff000
Empty Memory (*)
--/-- PGSIZE
USTACKTOP
0xeebfe000
Normal User Stack
RW/RW PGSIZE
0xeebfd000

Other pitfalls and misconceptions

- The MMU is being "bypassed" or "tricked" in certain situations.

Other pitfalls and misconceptions

- The MMU is being "bypassed" or "tricked" in certain situations.
  - Nope! Once we turned paging on, we have to have a page directory which determines the mapping.

Other pitfalls and misconceptions

- The MMU is being "bypassed" or "tricked" in certain situations.
    - Nope! Once we turned paging on, we have to have a page directory which determines the mapping.

Remember, very early on we map 4MB of memory from [KERNBASE, KERNBASE+4MB) to [0, 4MB]. This means there is a trivial mapping we can use to help us figure out what the physical address is for certain variables.

Other pitfalls and misconceptions

- The MMU is being "bypassed" or "tricked" in certain situations.
    - Nope! Once we turned paging on, we have to have a page directory which determines the mapping.

Remember, very early on we map 4MB of memory from [KERNBASE, KERNBASE+4MB) to [0, 4MB]. This means there is a trivial mapping we can use to help us figure out what the physical address is for certain variables.

- The kernel can access memory below KERNBASE using addresses less than 0xf0000000?

Other pitfalls and misconceptions

- The MMU is being "bypassed" or "tricked" in certain situations.
    - Nope! Once we turned paging on, we have to have a page directory which determines the mapping.

Remember, very early on we map 4MB of memory from [KERNBASE, KERNBASE+4MB) to [0, 4MB]. This means there is a trivial mapping we can use to help us figure out what the physical address is for certain variables.

- The kernel can access memory below KERNBASE using addresses less than 0xf0000000?
    - Nope! The kernel can only use kernel virtual addresses (0xf0000000). To the extent it needs to access memory elsewhere, there must be page table entries to make this happen.

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)
- Project ideas?

Virtual memory recap

- CPU asks OS to set up a data structure for VA → PA
  - per-process page table; flags (P/W/U/…)
  - switch page table with process
  - JOS: `inc/memlayout.h`
  - xv6
    - `struct proc` in `proc.h`
    - `scheduler()` → `switchuvm(p)` → `lcr3(v2p(p→pgdir))`

Virtual memory recap

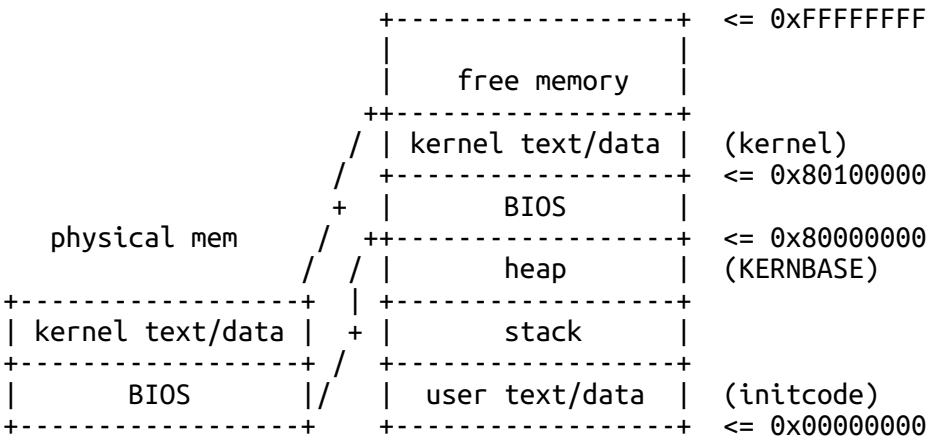
- Linux
  - `cat /proc/iomem`
  - `cat /proc/self/map` (or replace `self` with a PID)
  - are these physical or virtual addresses
- “All problems in computer science can be solved by another level of indirection”

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

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

Protection: preventing stack overflow

- 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

- Can we run an app. requiring > 2GB in xv6?
- 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

- 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
- When to page-in/page-out content?

Feature: single zero page

- `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
- What if one process writes to the page?

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

- 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

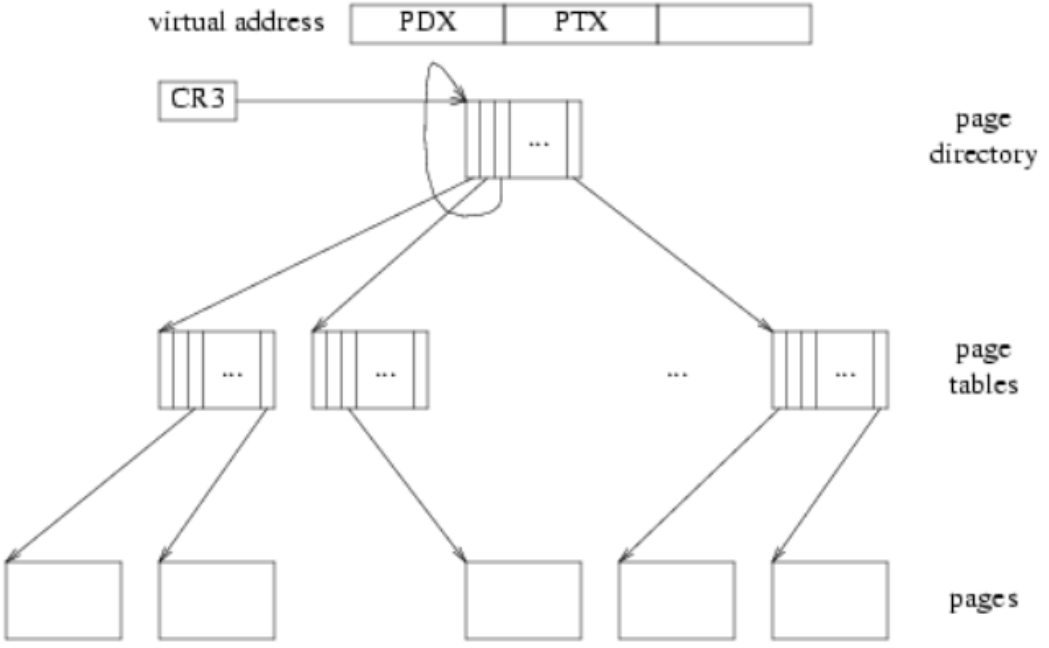
Feature: virtual *linear* page tables

- How big is the page table if we have a single level (4KB pages)?
- 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