EECS 482
Introduction to Operating Systems
Fall 2017
Manos Kaprİtsos
Slides by: Harsha V. Madhyastha
## Page table contents

<table>
<thead>
<tr>
<th>Physical page #</th>
<th>Resident</th>
<th>Read/Write enabled</th>
<th>Dirty</th>
<th>Referenced</th>
</tr>
</thead>
</table>

Written by OS, Read by MMU

Written by OS/MMU
Read by OS
Page table contents

Written by OS, Read by MMU

| Physical page # | read_enabled | write_enabled |
Address Space Management

- How to manage a process’s accesses to its address space?
  - Kernel sets up page table per process and manages which pages are resident
  - MMU looks up page table to translate any virtual address to a physical memory address

- What about kernel’s address space?
- How does MMU handle kernel’s loads and stores?
Storing Page Tables

- Two options:
  1. In physical memory
  2. In kernel’s virtual address space

- Difference: Is PTBR a physical or virtual addr?

- Pros and cons of option 2?
  - Can page out user page tables
  - Kernel page table must be kept in physical memory

- Project 3 uses option 2
  - Kernel’s address space managed by infrastructure
Kernel vs. user address spaces

- Can you evict the kernel’s virtual pages?
  - Yes, except code for handling paging in/out

- How can kernel access specific physical memory addresses (e.g., to refer to translation data)?
  - Kernel can **issue untranslated address** (bypass MMU)
  - Kernel can **map physical memory into a portion of its address space** (e.g., `vm_physmem` in Project 3)
How does kernel access user’s address space?

- Kernel can manually translate a user virtual address to a physical address, then access the physical address.

- Can map kernel address space into every process’s address space.

- Trap to kernel doesn’t change address spaces; it just enables access both OS and user parts of that address space.

```
fffff
  
  80000

7ffff
  
  00000
```

```
operating system

user process
```
Kernel vs. user mode

- How are we protecting a process’s address space from other processes?
- Must ensure that only kernel can modify translation data
- How does CPU know kernel is running?
  - Hardware support: Mode bit

- Recap of protection:
  - Address space $\rightarrow$ Translation data $\rightarrow$ Mode bit
Kernel vs. user mode

- How are we protecting a process’s address space from other processes?
- Must ensure that only kernel can modify translation data

In what mode does a root user’s process run?

How can a root user reboot the machine?

- Recap of protection:
  - Address space $\rightarrow$ Translation data $\rightarrow$ Mode bit
Switching from user process into kernel

- Faults and interrupts
  - Timer interrupts
  - Page faults
  - Why are these safe to transfer control to kernel?

- System calls
  - Process management: fork/exec
  - I/O: open, close, read, write
  - System management: reboot
  - …
System calls

- **When you call `cin` in your C++ program:**
  - `cin` calls `read()`, which executes assembly-language instruction `syscall`
  - `syscall` traps to kernel at pre-specified location
  - Kernel’s `syscall` handler calls kernel’s `read()`

- **To handle trap to kernel, hardware atomically**
  - Sets mode bit to kernel
  - Saves registers, PC, SP
  - Changes SP to kernel stack
  - Changes to kernel’s address space
  - Jumps to exception handler
Arguments to system calls

- Two options:
  - Store in registers
  - Store in memory (in whose address space?)

- Kernel must check validity of arguments
  - e.g., `read(int fd, void *buf, size_t size)`
Protection summary

- Safe to switch from user to kernel mode because control only transferred to certain locations
  - Where are these locations stored?
    - Interrupt vector table

- Who can modify interrupt vector table?

- Why is it easier to control access to interrupt vector table than mode bit?
Address Space Protection

- How are address spaces protected?
  - Separation of translation data

- How is translation data protected?
  - Can update translation data only if mode bit set

- How is mode bit protected?
  - Sets/reset mode bit when transitioning from user-level to kernel-level code and back
  - Transitions limited by interrupt vector table

- Protection boils down to init process which sets up interrupt vector table when system boots up
Project 3

- Look out for extra office hours

- Draw state machine for any virtual page
  - Spot differences between swap-backed and file-backed
  - Write simplest test case to exercise any path in state machine
  - Incremental development!

- Use assertions liberally
  - Check PTEs match rest of your state
  - Check that swap blocks are accounted for correctly
Project 3

- Regrade requests
  - Hand back entire exam by Tuesday
  - We may regrade entire exam, grade may go up or down

- Grade projections
  - $0.03 \times P1 + 0.45 \times P2 + 0.52 \times \text{midterm}$
  - $\geq 76$ fine: (A or B)
  - $\leq 65$ need to improve (C- or below)
  - Come find us if exam score was $\leq 43$
Process creation

- Steps
  - Allocate process control block
  - Initialize translation data for new address space
  - Read program image from executable into memory
  - Initialize registers
  - Set mode bit to “user”
  - Jump to start of program

- Need hardware support for last few steps
  - Similar to switching from kernel to user process after system call
Multi-process issues

- How to partition physical memory allocation among processes?
  - Fairness versus efficiency

- Global replacement
  - Can evict pages from this process or other processes

- Local replacement
  - Can evict pages only from this process
  - Must still determine how many pages to allocate to this process
Thrashing

- What would happen if many large processes all actively used their entire address space?

- Performance degrades rapidly as miss rate goes up
  - Avg access time = hit rate * hit time + miss rate * miss time
  - E.g., hit time = .0001 ms; miss time = 10 ms
    - Average access time (100% hit rate) = .0001 ms
    - Average access time (99% hit rate) = ?
    - Average access time (90% hit rate) = ?
Solutions to Thrashing

- Buy more DRAM
  - Price per GB fallen by 4x since 2009

- Run fewer processes for longer
  - Example: Longer time slice
  - Reduces page faults
Working set

- Thrashing depends on portion of address space actively used by each process
  - What do we mean by “actively using”?
- Working set = all pages used in last T seconds
  - Larger working set $\Rightarrow$ process needs more physical memory to run well (i.e., avoid thrashing)
- Sum of all working sets should fit in memory
  - Only run subset of processes that fit in memory
- How to measure size of working set?
  - Periodic sweep of clock hand in LRU clock