EECS 482
Introduction to Operating Systems

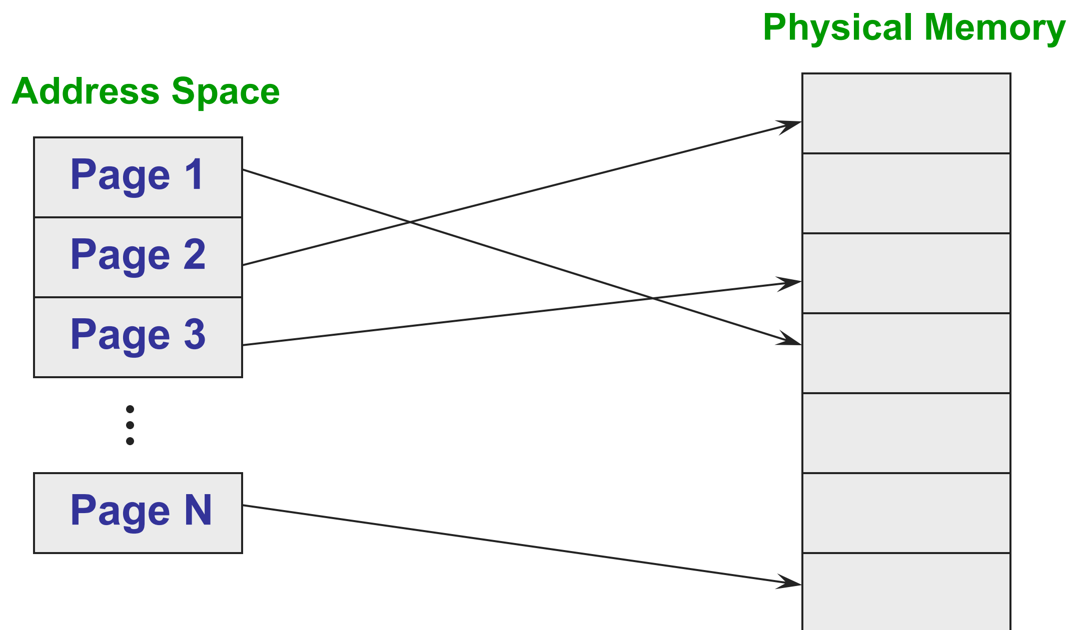
Fall 2018

Baris Kasikci

Slides by: Harsha V. Madhyastha

Recap: Paging

- Both address spaces and physical memory broken up into fixed size pages



Paging

Virt. page #	Phys. page #	resident	protected
0	105	1	0
1	15	0	0
2	283	1	1
3	invalid		
…	invalid		

```
if (virtual page is invalid or non-resident or protected) {
        trap to OS fault handler; retry
} else {
        physical page # = pageTable[virtual page #].physPageNum
}
```

Page Replacement

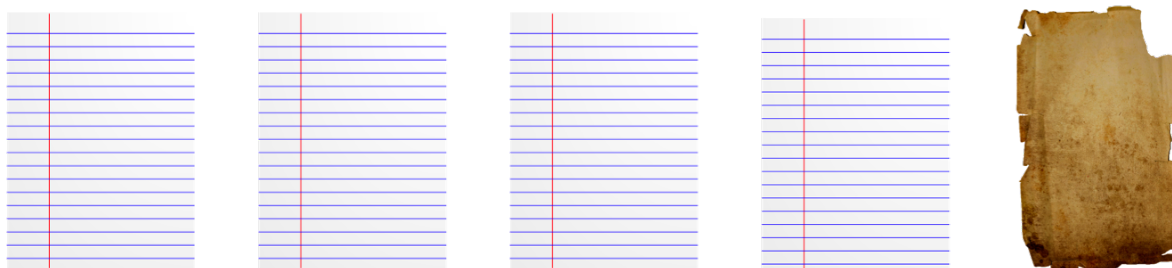
- Not all valid pages may fit in physical memory
  - Some pages are swapped out to disk
- To read in a page from disk, some resident page must be swapped out to disk
- Which page to evict when you need a free page?
  - Goal: minimize page faults

Replacement policies

- Random
- FIFO
  - Replace page brought into memory longest time ago
  - May replace pages that continue to be frequently used
- Optimal (OPT)?
  - Replace page that won't be used for the longest time in the future
  - Minimizes misses, but requires knowledge of the future

Replacement policies

- LRU (least recently used)
  - Approximates OPT by using past reference pattern
    - » If page hasn't been used for a while, it probably won't be used for a long time in the future



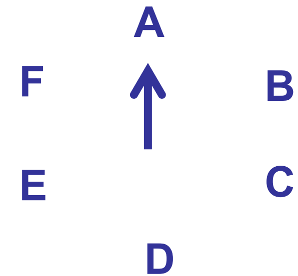
- LRU is hard to implement exactly
  - Can we simplify LRU by approximating it?

The “referenced” bit

- Most MMUs maintain a “referenced” bit for each resident page
  - Set by MMU when page is read or written
  - Can be cleared by OS
- How to use reference bit to identify old pages?
- Why maintain reference bit in hardware?

Clock replacement algorithm

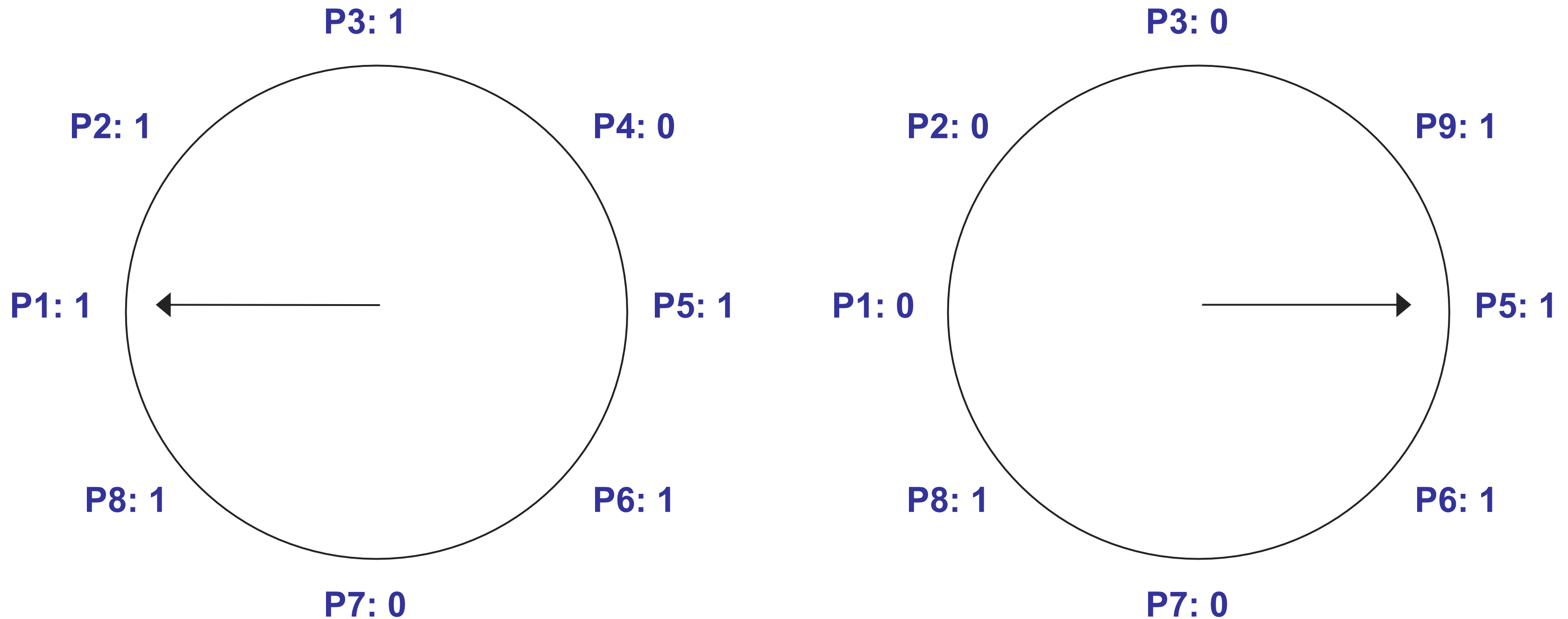
- Arrange resident pages around a clock



- Algorithm to select page for eviction:
  - Consider page pointed to by clock hand
  - If not referenced, page hasn't been accessed since last sweep → Evict
  - If referenced, page has been referenced since last sweep
    - » What to do?

- What if all pages have been referenced since last sweep?
- What about new pages?

LRU Clock



- A nice feature of clock is that it only adds overhead when you need to evict a page

Page eviction

- Where to evict page to?
- When do you NOT need to write page to disk?
  - Rely on hardware/MMU to maintain dirty bit in PTE

- Why not write to disk on every store?
- The page that is brought from disk must wait while some page is evicted
  - How could you optimize eviction?
- DON'T use these optimizations in Project 3!

Page table contents

```
if (virtual page is non-resident or protected) {
        trap to OS fault handler
        retry access
} else {
        physical page # = pageTable[virtual page #].physPageNum
        pageTable[virtual page #].referenced = true
        if (access is write) {
                pageTable[virtual page #].dirty = true
        }
        access physical memory
}
```

Page table contents

Physical page #	Resident	Read/Write enabled	Dirty	Referenced

- Why no valid bit in PTE?
  - All invalid virtual pages are non-resident
- For valid non-resident pages, does PTE contain disk block?
  - OS must maintain this, MMU simply traps to OS
- Can we make do without resident bit?
  - Use protection bits

Page table contents

Physical page #	Read/Write enabled	Dirty	Referenced

- Can we make do without dirty bit?
  - Use protection bits
    - » Make clean pages write-protected
    - » Change the write-enabled bit after page becomes dirty
      - If the page is indeed not write-protected
  - Won’t this increase # of page faults a lot?

Page table contents

Physical page #	Read/Write enabled	Referenced

- Can we make do without referenced bit?
- Application too may want to control protection
  - Not in project 3

Page table contents

Physical page #	read_enabled	write_enabled

Midterm

- Review different synchronization techniques we have learned
- You should be comfortable with project-2 concepts/code

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 User Page Tables

- Two options:
  - In physical memory
  - In kernel's virtual address space
- Difference: Is PTBR a physical or virtual addr?
- Pros and cons
  - Can page out user page tables
  - Kernel page table must be kept in physical memory

- Project 3 uses second option
  - 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)