Demand Paging

Mendel Rosenblum

Paging

Optional readings: Operating Systems: Principles and Practice: Chapter 9

Demand Paging Overview

Goal: allow programs to run without all of their information in memory

- Keep in memory physical pages being used
  - Idle pages kept on disk
- Pages stored on disk in a **paging file**
  - Also called **backing store**, or **swap space**
- Move information back and forth as needed
- Works because of **Locality of Reference**:
  - Most programs spend most of their time using a small fraction of their code and data

True "Virtual" Memory

Each page can be either:

- In memory (physical page frame)
  - DRAM is 100,000x faster than disk
  - DRAM is 1000x faster than SSD (flash memory)
- On disk (***backing store***)
  - Disk cost is ~100x less than DRAM
  - SSDs cost is ~100x less than DRAM

We would like to have a virtual memory that is cheap as disk, and as fast as DRAM

Page Faults - the mechanism

- Present bit in page map 0 for pages in the backing store
- CPU references to virtual addresses with present bit off trap to OS
  - Called a **page fault** trap
- Page fault handling
  - Checks to see if valid page (could be error)
  - Finds a free page in memory
  - Read the page from backing store
  - Update the page map entry to point at allocated page and set present bit
  - Resumes execution of the thread

Hardware support for paging

- Latch faulting address on a page table
  - X86-64 has a privileged register `CR2`
- Instructions need to be restartable
  - Typically undo any changes so instruction can be restarted.
  - Example: `push memory` - update stack pointer and write memory to stack

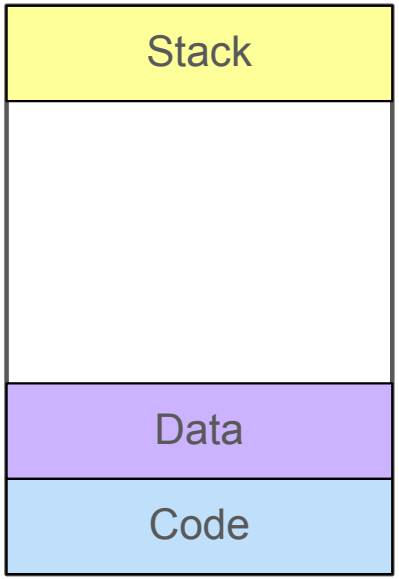
Paging policies

The OS has two policy decisions to make:

- Page fetching: when to bring pages into memory
- Page replacement: which pages to throw out of memory

Page fetching policy - Demand fetching

- **Demand fetching** - Pages are fetched when page faulted in
  - Start process with no pages loaded
    - Demand: Don't load a page into memory until it is referenced
  - Possible source locations:
    - Executable file
      - Read-only code pages fetched on PC fetch page fault
      - Initialized data pages fetched when data first accessed
    - Uninitialized data pages (stack, malloc) zero page returned
      - Data pages written to the backing store when modified
    - Backend store
      - Used for data pages



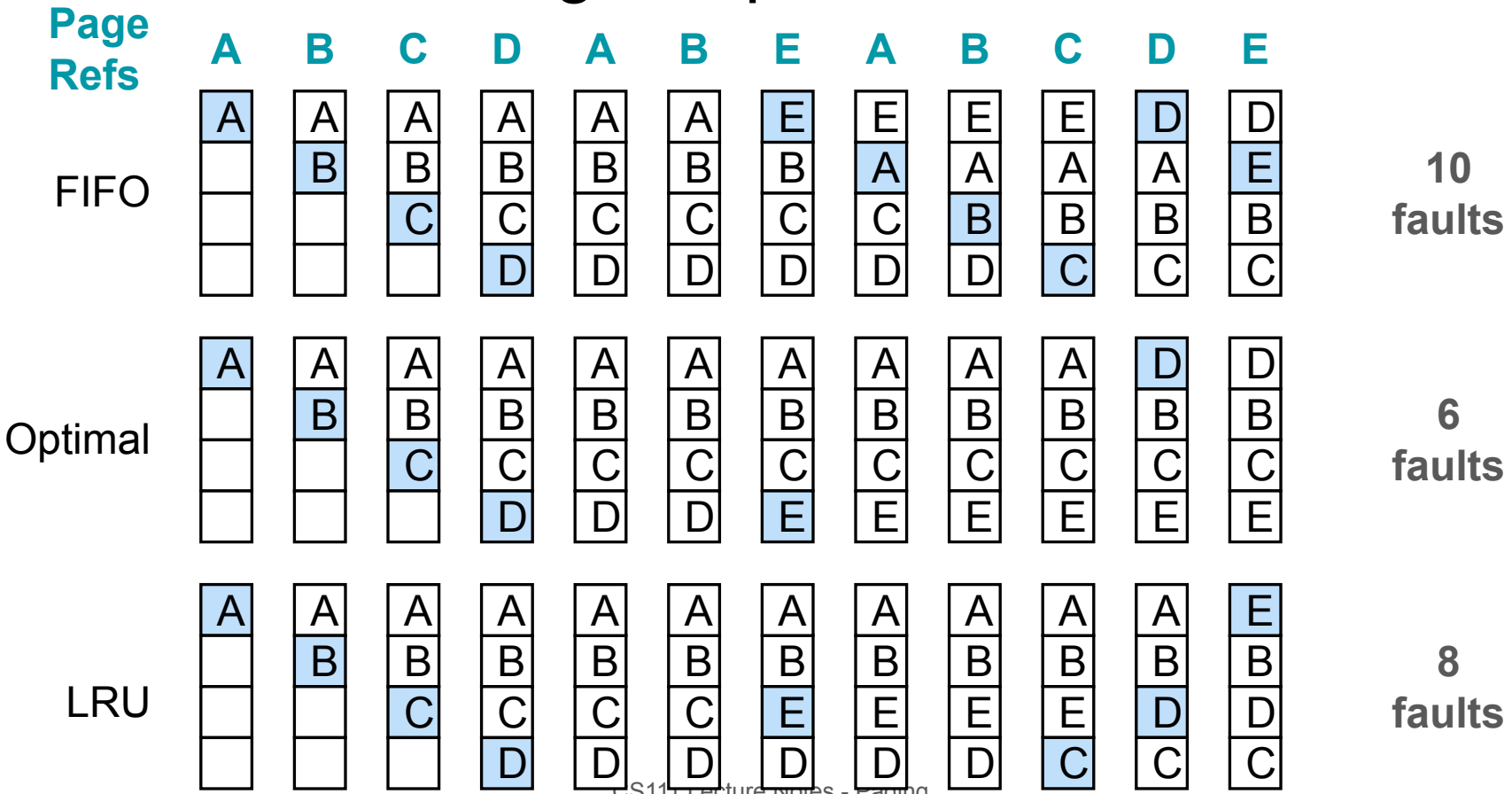
Page fetching policy - Prefetching

- OS to predict when pages will be needed
  - Load them ahead of time to avoid page faults
- Requires predicting the future, so hard to do
- One approach:
  - When taking a page fault, read a few pages instead of just one
    - Wins if program accesses memory sequentially
- Timings
  - Disk: 5-10 milliseconds for a page fault, fast prefetch .04 milliseconds
  - SSD: 50–100 microseconds for a page fault, fast prefetch 10–20 microseconds
- For comparison: DRAM 50-100 nanoseconds (even SSD 1000x slower)

Page Replacement Policy

- Once memory fills, need to throw out one page each time there is a page fault
- Possible policies:
  - Random - Choose a random page to replace
    - Easy to implement - low accounting overhead
    - Works surprisingly well
  - FIFO - Choose page in memory the longest
    - Easy to implement - low accounting overhead
    - Fair to the pages
  - MIN (optimal) - Replace page accessed farthest in future
    - Not practical since it requires us to predict the future
  - LRU (least recently used) - use the past to predict the future
    - Should approximate MIN if past does predict the future

Page Replacement



Implementing LRU

- Need hardware support to keep track of page accesses
- One proposed approach:
  - Add a timestamp register to each physical page
  - Store the system clock into the page's register on every memory access
  - On replacement, page with the oldest timestamp is LRU
- Not practical to implement
  - Hardware costs prohibitively high (lots of high speed registers)
  - Lot of scanning on replacement
- Not strictly needed:
  - LRU is an approximation of MIN, used an approximation of LRU
  - Just find an old page, not necessarily the oldest.

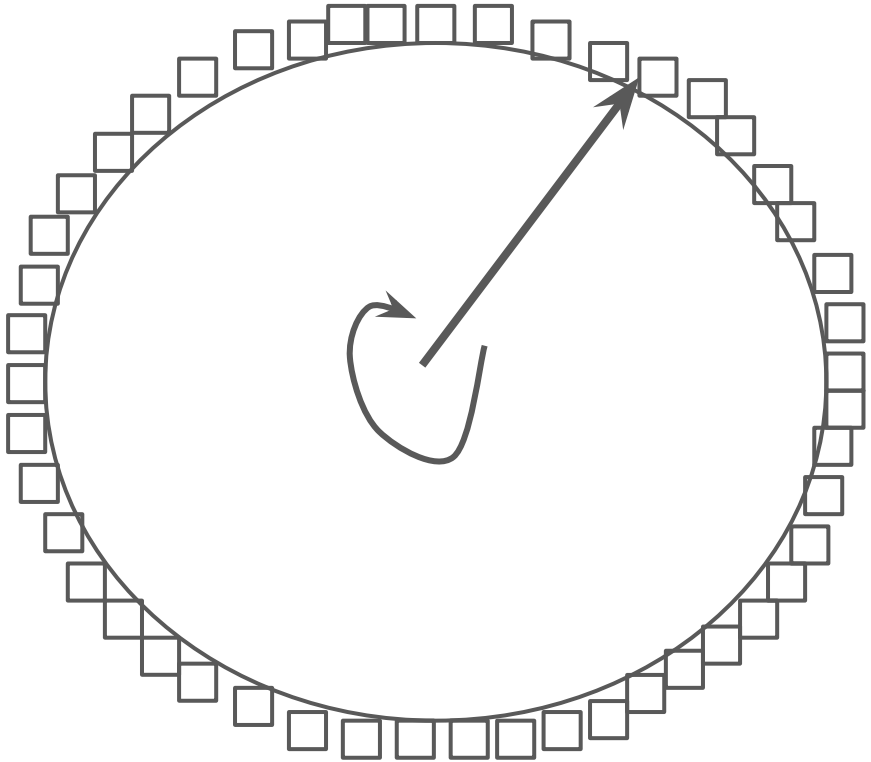
Common hardware support for page replacement

- Two extra bits in each page map entry:
  - **Referenced bit**: set whenever a page is read or writte. (Called **Accessed bit** on x86)
  - **Dirty bit**: set whenever the page is modified
- Can be set by the CPU/MMU during page table walk
  - Used by x86-64
- Some moderm CPU architectures have OS simulate bits (e.g ARM64)
  - Reference bits: Accesses to a page with 0 reference bit trap into OS which set the bit
  - Dirty bits: OS maps page with 0 dirty bit read-only so writes trap into OS which set the bit
  - Newest versions of ARM64 had hardware support to reduce traps

LRU approximation: Clock algorithm

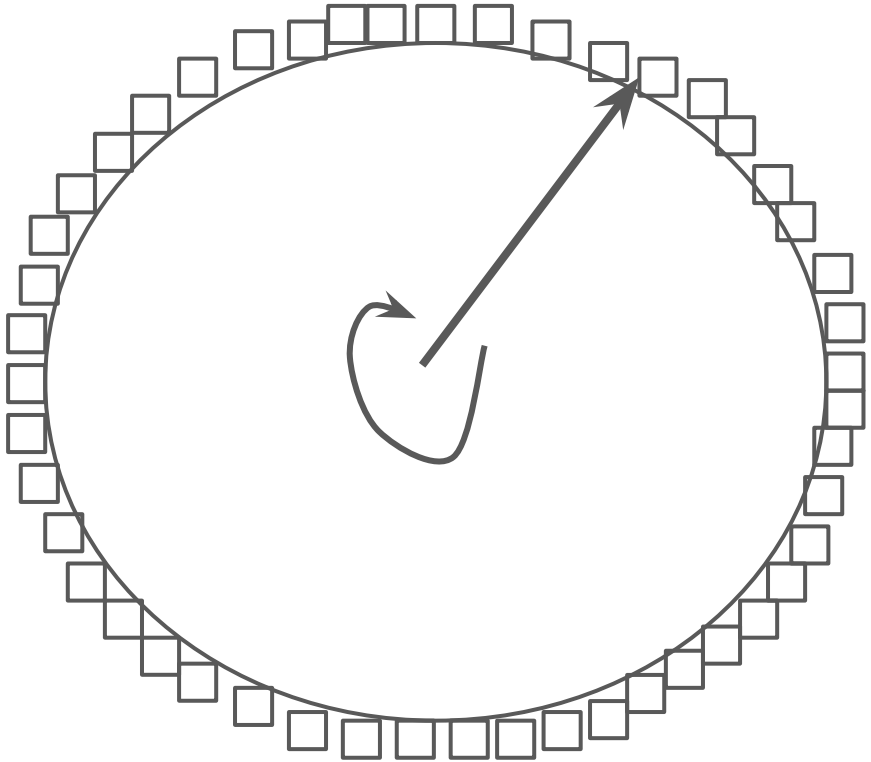
Also called **second chance algorithm**

- Arrange physical memory pages into circle
- Have a hand that points at a page
- Hand moves clockwise on each page fault that needs a page of memory
  - OS free list of memory pages empty



LRU approximation: Clock algorithm hand operation

- Page fault needs page: advance hand
- If reference bit is set: clear bit
  - Second chance
  - What was the first chance?
- If reference bit is cleared: replace it
  - Page hasn't been referenced since last check
  - If the dirty bit is set, clear the dirty bit and start writing the page to disk



Clock hand speed is an interesting metric

Is it better to have the hand moving quickly or slowly?

- Slowly?
- Quickly?

- Hand moves when physical pages are needed and free list is empty
  - Slowly: Machine has enough memory for workload, not many page faults
  - Quickly: Machine doesn't have enough memory for workload, many page faults

Page replacement global vs per-process

What should happen if we have multiple processes using memory?

- Global replacement:
  - All pages from all processes lumped into a single replacement pool
  - Each process competes with all the other processes for page frames
  - Not good for performance isolation
- Per-process replacement:
  - Each process has a separate pool of pages
  - A page fault in one process can only replace one of its own frames.
  - Eliminates interference from other processes.
  - How many page frames should each process get?
- Most systems use global replacement.

Thrashing

- What happens if memory gets overcommitted?
  - Pages being actively used by the current threads don't all fit in physical memory
- Each page fault replaces an active pages, so another page fault will occur soon.
  - All time spent reading and writing pages: won't get much work done.
- Timing:
  - DRAM memory takes 100 ns. Disk access take 10 ms (10,000,000 ns)
  - Memory is 1% too small: 1 in every 100 memory reference is a page fault
  - .99 * 100 ns + .1 * 10,000,000 ns = 100,099 ns (1000x slowdown!)
  - Wanted memory as big as disk and as fast a DRAM, got as fast a disk
- Called **Thrashing**
  - Serious problem in early timesharing machines with many users

Handling thrashing

- OS needs to stop running some processes for a white
- CPU scheduler should only run jobs together that fit in memory
- With personal computers, user can decide what to stop
- Inexpensive memory today:
  - Buy enough memory to avoid thrashing