Lecture 13: Working Set, Thrashing, and Storage Devices

Yiying Zhang
Outline

- Finishing up page replacement and memory section
- Storage devices
- Homework 3 Due tomorrow
[lec12] LRU Clock (Not Recently Used)

- Clock algorithm – Used by Unix
- Idea: Replace page that is “old enough”
- Arrange all of physical page frames in a big circle (clock)
- A clock hand is used to select a good LRU candidate
  - Sweep through the pages in circular order like a clock
  - If the ref bit is off, it hasn’t been used recently
    - Pick it for page replacement (victim page)
    - What is the minimum “age” if ref bit is off?
  - If the ref bit is on, turn it off and go to next page. (why turn off?)
- Low overhead when plenty of memory
[lec12] Enhanced Clock

• Same as the basic Clock, except that it considers both (reference bit, modified bit)
  ♦ (0,0): neither recently used nor modified (good)
  ♦ (0,1): not recently used but dirty (not as good)
  ♦ (1,0): recently used but clean (not good)
  ♦ (1,1): recently used and dirty (bad)

• On page fault, follow hand to inspect pages:
  ♦ Round 1:
    » If bits are (0,0), take it
    » if bits are (0,1), record 1st instance
    » Clear ref bit for (1,0) and (1,1), if (0,1) not found yet
  ♦ At end of round 1, if (0,1) was found, take it
  ♦ If round 1 does not succeed, try 1 more round
What else can we do to improve miss latency?
Page Fault Handling in demand paging

1. MMU (TLB)
2. Page fault
3. Page replacement
4. Adjust PTE of victim pg
5. Swap in page from swap
6. Resume faulting intr
Page out on critical path?

- If no free page in physical memory, swap in has to wait till a current page in physical memory is swapped out
  - Page fault handling time = proc. overhead + 2 * I/Os
- There is a chance of swapped out page being referenced soon
Page buffering techniques

OS maintains a pool of free pages
- When a page fault occurs, victim page chosen as before
- But desired page swapped into a free page (a slot in the free page pool) right away before victim page paged out
- OS swaps out dirty victim pages in the background, off the page fault critical path (to make more room in the free page pool)
Page buffering techniques

- Maintaining a list of free physical pages enables another important optimization.
- Recall that the page replacement algorithm is a rough approximation of LRU:
  - Can certainly make mistakes.
  - LRU does not necessarily work well for all program behaviors.
- Idea: If a page is on the free list, and it is accessed by a process before being reallocated, rescue it from the free list and give it back to the process:
  - Recovers from poor choices made by replacement algorithm.
Key observation

- Locality in memory references
  - Spatial and temporal

- Want to keep a set of pages in memory that would avoid a lot of page faults
  - “Hot” pages

- Can we formalize it?
Working Set Model

• A working set of a process is used to model the dynamic locality of its memory usage
  ◦ Defined by Peter Denning in 60s

• Definition
  ◦ WS(t,w) = \{all the pages that were referenced in the time interval (t, t-w)\}
  ◦ t: time, w: working set window (measured in page refs)

• A page is in the working set (WS) only if it was referenced in the last w references
Working Set Size

- Working set size is the number of unique pages in the WS
  - The number of pages referenced in the interval \((t, t-w)\)
- The working set size changes with program locality
  - During periods of poor locality, you reference more pages
  - Within that period of time, the working set size is larger
- Intuitively, want the working set to be the set of pages a process needs in memory to prevent heavy faulting
  - Each process has a parameter \(w\) that determines a working set with few faults
  - Denning: Don’t run a process unless working set is in memory

\[
\begin{align*}
WS(t_1) &= \{1,2,5,6,7\} \\
WS(t_2) &= \{3,4\}
\end{align*}
\]
Working Sets in the Real World

transition, stable

Working set size
Working Set Problems

• Problems
  ♦ How do we determine \( w \)?
  ♦ How do we know when the working set changes?

• Too hard to answer
  ♦ So, working set is not used in practice as a page replacement algorithm

• However, it is still used as an abstraction
  ♦ The intuition is still valid
  ♦ When people ask, “How much memory does Firefox need?”,
    they are in effect asking for the size of Firefox’s working set
The BIG picture

- We’ve talked about single evictions
- Most computers are multiprogrammed
  - Single eviction decision still needed
  - New concern – processes compete for resources
  - How to be “fair enough” and achieve good overall throughput
Possible replacement strategies

• Global replacement:
  - All pages from all processes are lumped into a single replacement pool
  - Most flexibility, least (performance) isolation

• Local replacement
  - Per-process replacement:
    » Each process has a separate pool of pages
  - Per-user replacement:
    » Lump all processes for a given user into a single pool

• In local replacement, must have a mechanism for (slowly) changing the allocations to each pool
Improving CPU utilization in multiprogramming

- In multiprogramming, when OS sees the CPU utilization is low,
  - It thinks most processes are waiting for I/O
  - it needs to increase the degree of multiprogramming (actual behavior of early paging systems)
  - It adds loads another process to the system
When there are not enough page frames

- Suppose many processes are making frequent references to 50 pages, memory has 49
- Assuming LRU
  - Each time one page is brought in, another page, whose content will soon be referenced, is thrown out
- Btw, what is the optimal strategy here?
  - MRU

- What is the average memory access time?
- The system is spending most of its time paging!

- The progress of programs makes it look like “memory access is as slow as disk”, rather than “disk being as fast as memory”
Thrashing

- Thrashing
  - When most of the time is spent by the OS in paging data back and forth from disk
  - Little time spent doing useful work (making progress)
  - In this situation, the system is overcommitted
Thrashing can lead to vicious cycle

• If a process does not have “enough” pages, the page-fault rate is very high. This leads to:
  ♦ low CPU utilization
  ♦ OS thinks that it needs to increase the degree of multiprogramming (actual behavior of early paging systems)
  ♦ another process added to the system
  ♦ page fault rate goes even higher
Thrashing (Cont.)
What causes thrashing?

• The system does not know it has taken more work than it can handle

• What do humans do when thrashing?
  ♦ Dropping or degrading a course if taking too many than you can handle 😊
Intuitively, what to do about thrashing?

- If a single process’s locality too large for memory, what can OS do?
  - e.g., pin most data (hotter data) in memory, sacrifice the rest

- If the problem arises from the sum of several processes?
  - Figure out how much memory each process needs — “locality”
  - What can we do?
    » Can limit effects of thrashing using local replacement
    » Or, bring a process’ working set before running it
    » Or, wait till there is enough memory for a process’s need
Summary

• Page replacement algorithms
  ♦ Belady’s – optimal replacement (minimum # of faults)
  ♦ FIFO – replace page loaded furthest in past
  ♦ LRU – replace page referenced furthest in past
    » Approximate using PTE reference bit
  ♦ Clock – replace page that is “old enough”
  ♦ Enhanced Clock – pick clean pages first (for lower miss latency)
  ♦ Working Set – keep the set of pages in memory that has minimal fault rate (the “working set”)

• Multiprogramming
  ♦ Global vs. local replacement
  ♦ Thrashing
Memory Hierarchy

- Storage device (e.g., Disk, SSD)
  - bottom of memory hierarchy
A More General/Realistic I/O System

- I/O peripherals: disks, input devices, displays, network cards, ...
  - With built-in or separate I/O (or DMA) controllers
  - All connected by a system bus
File and Storage Systems

- The third part of the course (and OS)

- First we’ll discuss properties of storage devices
- Then how file systems support users and programs
- End with how file systems are implemented
Disks and the OS

• Disks are messy physical devices:
  ♦ Errors, bad blocks, missed seeks, etc.

• The job of the OS is to hide this mess from higher level software
  ♦ Low-level device control (initiate a disk read, etc.)
  ♦ Higher-level abstractions (files, databases, etc.)

• The OS may provide different levels of disk access to different clients
  ♦ Physical disk (surface, cylinder, sector)
  ♦ Logical disk (disk block #)
  ♦ Logical file (file block, record, or byte #)
What's Inside a Disk Drive?

- Arm
- Spindle
- Platters
- Actuator
- Electronics
- SCSI connector
Disk Components

- Read/Write Head
- Upper Surface Platter
- Lower Surface
- Cylinder
- Track
- Sector
- Arm
- Actuator
Disk Head Position
Rotation is Counter-Clockwise
About to Read Blue Sector
After Reading Blue Sector

After BLUE read
Red Request Scheduled Next

After BLUE read
Seek to Red’s Track

After **BLUE** read

Seek for **RED**
Wait for Red Sector to Reach Head

- After **BLUE** read
- Seek for **RED**
- Rotational latency
Read Red Sector

After **BLUE** read  
Seek for **RED**  
Rotational latency  
After **RED** read
Disk Performance

• Disk request performance depends upon three steps
  ♦ Seek – moving the disk arm to the correct cylinder
    » Slowest part of disk accesses, bound by physical laws
    » Depends on how fast disk arm can move (increasing very slowly)
  ♦ Rotation – waiting for the sector to rotate under the head
    » Depends on rotation rate of disk (increasing, but slowly)
  ♦ Transfer – transferring data from surface into disk controller electronics, sending it back to the host
    » Depends on density (increasing quickly)

• When the OS uses the disk, it tries to minimize the cost of all of these steps
  ♦ Particularly seeks
**Disk Interaction**

- Specifying disk requests requires a lot of info:
  - Cylinder #, surface #, track #, sector #, transfer size…

- Older disks required the OS to specify all of this
  - The OS needed to know all disk parameters

- Modern disks are more complicated
  - Not all sectors are the same size, sectors are remapped, etc.

- Current disks provide a higher-level interface
  - The disk exports its data as a logical array of blocks [0…N]
    - Disk maps **logical blocks** to cylinder/surface/track/sector
    - Block size can be configured via low-level formatting
    - This interface is called the **block interface**

- Only need to specify the logical block # to read/write
- But now the disk parameters are hidden from the OS
Disk Observations

• Getting first byte from disk read is slow
  ♦ high *latency*

• Peak disk bandwidth good, but rarely achieved

• Towards mitigate disk performance impact
  ♦ Disk caches (read-ahead and write buffer)
  ♦ Move some disk data into main memory – file caching
  ♦ Disk scheduling
    » There are often multiple disk requests outstanding
    » Schedule requests to shorten seeks!

♦ *What else can we try?*
  » *Adding multiple disk arms to the disk?*
RAID

- Invented by Dave Patterson

- Two motivations
  - (initially) Operating in parallel can increase disk throughput
    » RAID = Redundant Array of Inexpensive Disks
  - (today) Redundancy can increase reliability
    » RAID = Redundant Array of Independent Disks
RAID -- Two main ideas

• Parallel reading/writing (striping) (for performance)
  ♦ Splitting data blocks across multiple disks and read/write them in parallel

• Mirroring (for reliability)
  ♦ Have a “mirror” (shadow) disk that stores the same data
  ♦ Every write performed on both disks
    » Can read from either disk
Raid Level 0: Stripe Only

- Level 0 is **non-redundant** disk array
- Files are striped across disks, no redundant info
- High read throughput
- Best write throughput among RAID levels (no redundant info to write)
- Any disk failure results in data loss
Raid Level 1: Mirroring

- Data is written to two places
  - On failure, just use surviving disk
- On read, choose fastest to read
  - Write performance is same as single drive, read performance is 2x better
- Expensive (but used by quite a few companies)
Parity

• What do you need to do in order to detect and correct a one-bit error?
  ♦ Suppose you have a binary number, represented as a collection of bits: <b3, b2, b1, b0>, e.g. 0110

• Detection is easy

• Parity:
  ♦ Count the number of bits that are on, see if it’s odd or even
    » EVEN parity is 0 if the number of 1 bits is even
  ♦ Parity(<b3, b2, b1, b0 >) = P0 = b0 ⊗ b1 ⊗ b2 ⊗ b3
  ♦ Parity(<b3, b2, b1, b0, p0>) = 0 if all bits are intact
  ♦ Parity(0110) = 0, Parity(01100) = 0
  ♦ Parity(11100) = 1 => ERROR!
  ♦ Parity can detect a single bit error, but can’t tell you which of the bits got flipped
Raid Level 4

- Block-level parity with striping
- Lower transfer rate for each block (by single disk)
- Higher overall rate (many small files, or a large file)
- Large writes → parity bits can be written in parallel
- Small writes → 2 reads + 2 writes!
- Heavy load on the parity disk
Raid Level 5

- Block Interleaved Distributed Parity
- Like parity scheme, but distribute the parity info over all disks (as well as data over all disks)
- Better (large) write performance
  - No single disk as performance bottleneck
Flash Memory and Flash-Based SSDs
Number of NAND Flash Units (millions)

Source: iSuppli Q1 2011
Flash-Based Solid State Disks

• SSDs are a relatively new storage technology
  ♦ Memory that does not require power to remember state
• No physical moving parts → faster than hard disks
  ♦ No seek and no rotation overhead
  ♦ But…more expensive, not as much capacity
• Generally speaking, the block interface and file systems can remain unchanged when using SSDs
  ♦ Some optimizations no longer necessary (e.g., layout policies, disk head scheduling), but basically can leave FS code as is
  ♦ New file systems designed for flash and SSDs
    » E.g., flash-based file system in Samsung phones
Flash-Based SSD

OS

Read/Write (data, sector, size)

Block Interface

Logical

Physical

SSD

Controller

RAM

Flash Memory
Non-Volatile Memory (NVM)

- A generation of new technologies that provide non-volatile (persistent) memory
  - Phase change (PCM), spin-torque transfer (STTM), etc.
  - Intel Optane (3D Xpoint) commercially available!
- Performance close to DRAM
  - But persistent!
- Byte-addressable
  - SSD is in units of a page (e.g., 4KB)
- NVM will have a dramatic effect on both OSes and applications
The Landscape of Memory and Storage
Is Changing!

Active research at UCSD:
my own group, Steve Swanson, Jishen Zhao

Latency

$\downarrow$

Byte-Addressable

<table>
<thead>
<tr>
<th>DRAM</th>
</tr>
</thead>
</table>

Next-Gen NVM

| 3D Xpoint |
| Phase Change Memory |
| Memristor |
| STT-RAM |

Persistent

| Flash |

Disk

$\uparrow$

$\$
Next time...

- Chapters 39, 40, 41
Backup Slides
Disk Specifications

- Seagate Enterprise Performance 3.5" (server)
  - capacity: 600 GB
  - rotational speed: 15,000 RPM
  - sequential read performance: 233 MB/s (outer) – 160 MB/s (inner)
  - seek time (average): 2.0 ms

- Seagate Barracuda 3.5" (workstation)
  - capacity: 3000 GB
  - rotational speed: 7,200 RPM
  - sequential read performance: 210 MB/s - 156 MB/s (inner)
  - seek time (average): 8.5 ms

- Seagate Savvio 2.5" (smaller form factor)
  - capacity: 2000 GB
  - rotational speed: 7,200 RPM
  - sequential read performance: 135 MB/s (outer) - ? MB/s (inner)
  - seek time (average): 11 ms