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?
A working set of a process is used to model the dynamic locality of its memory usage.

- \( WS(t,w) = \{ \text{all pages referenced in the time interval (t, t-w)} \} \)

- Working set size is the number of unique pages in the WS.
- The working set size changes with program execution.

![Diagram showing the relationship between working set size and page faults](image)
Working Sets in the Real World

Working set size

transition, stable
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
When there are not enough page frames

- Suppose many processes are making frequent references to 50 pages; there are 49 physical pages.
- Assuming LRU
  - Each time one page is brought in, another page, whose content will soon be referenced, is thrown out.
- 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".
- Btw, what is the optimal strategy here?
  - MRU
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
  - another process added to the system
  - page fault rate goes even higher
Thrashing (Cont.)

![Graph showing the relationship between CPU utilization and degree of multiprogramming. The graph indicates a curve that peaks and then declines, with a label for 'thrashing' on the x-axis.](image-url)
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
  - 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”)

- We are finally done with memory management!
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
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
Disks and the OS

• Disks are messy physical devices:
  ♦ With many physical parts
  ♦ 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.)
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 (we’ll see an example later on)
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
  ♦ Sectors can be remapped, etc.

• Modern 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
  ♦ OS only needs 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?
    » Disk parallelism (see later slides on RAID)
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

SSD

Controller

Flash Memory

RAM

Logical

Physical
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) (discontinued)
- Performance close to DRAM
  - But persistent!
- Byte-addressable
  - SSD is in units of a page (e.g., 4KB)
The Landscape of Memory and Storage

Latency

- 1ns
- 10ns
- 100ns
- 1μs
- 10μs
- 100μs
- 1ms
- 10ms

Persistent

- Flash
- Disk

Byte-Addressable

- DRAM
- NVM

$
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 can you do to recover from losing a block in a stripe?
  ♦ Let’s say, we have three disks, A, B, C, storing a stripe of data 011000
  ♦ A:01, B:10, C:00

• Parity:
  ♦ Calculate the XOR of data blocks in data disks
  ♦ Parity(01,10,00) = XOR(01, 10, 00) = 11
  ♦ Now we store parity (11) in a separate disk (D)
  ♦ XOR(A, B, C, D) = 0
  ♦ If we lose a disk (e.g., B)?
    » B = XOR(A, C, D) = XOR(01, 00, 11) = 10
**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

Diagram:
- Stripe 0
- Stripe 1
- Stripe 2
- Stripe 3
- P0-3
- Stripe 4
- Stripe 5
- Stripe 6
- P4-7
- Stripe 7
- Stripe 8
- Stripe 9
- Stripe 10
- P8-11
- Stripe 11

(data and parity disks)
Next time...

- Read Chapters 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