Computer Systems and Networks

ECPE 170 – Jeff Shafer – University of the Pacific

Memory Hierarchy

(Performance Optimization)

Lab Schedule

Activities

- **This Week**
  - **Lab 6 – Perf Optimization**
  - **Lab 7 – Memory Hierarchy**
- **Next Tuesday**
  - Intro to Python
- **Next Thursday**
  - ** **Midterm Exam** **

Assignments Due

- **Lab 6**
  - **Due by Mar 6th 5:00am**
- **Lab 7**
  - **Due by Mar 20th 5:00am**

Your Personal Repository

```
2017_spring_ecpe170\lab02
                     |lab03
                     |lab04
                     |lab05
                     |lab06
                     |lab07
                     |lab08
                     |lab09
                     |lab10
                     |lab11
                     |lab12
                     |.hg
```

Hidden Folder!
(name starts with period)

Used by Mercurial to track all repository history (files, changelogs, …)

Mercurial .hg Folder

- The existence of a `.hg` hidden folder is what turns a regular directory (and its subfolders) into a special Mercurial repository
- When you add/commit files, Mercurial looks for this `.hg` folder in the current directory or its parents

Memory Hierarchy

Memory Hierarchy

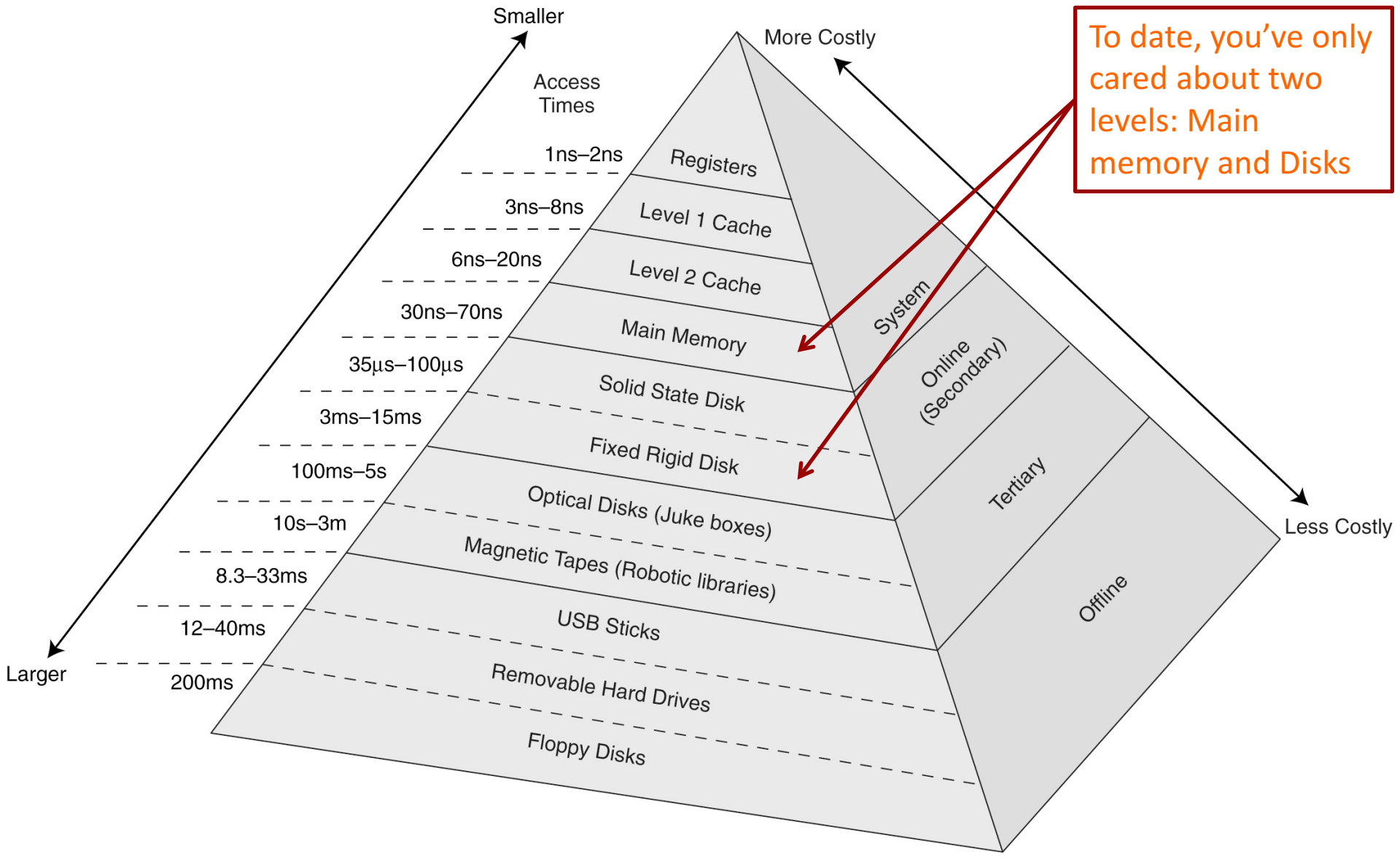
Goal as system designers:

Fast Performance **and** Low Cost

Tradeoff: Faster memory is
more expensive than slower memory

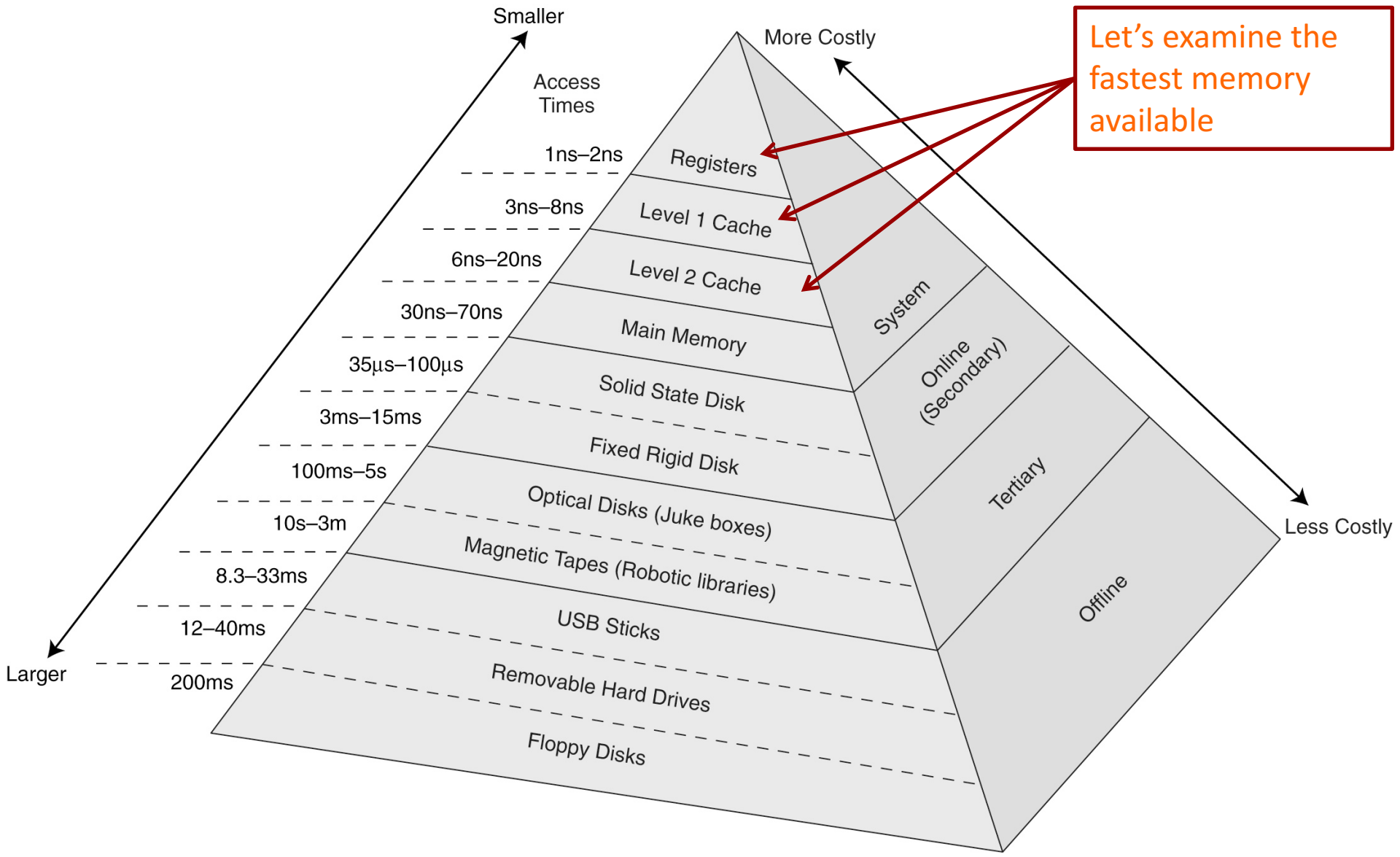
Memory Hierarchy

- To provide the best performance at the lowest cost, memory is organized in a hierarchical fashion
  - **Small**, **fast** storage elements are kept **in the CPU**
  - **Larger**, **slower** main memory are **outside the CPU** (and accessed by a data bus)
  - **Largest**, **slowest**, permanent storage (disks, etc…) is **even further** from the CPU



To date, you've only cared about two levels: Main memory and Disks

Memory Hierarchy
– Registers and Cache


Smaller
Access Times
More Costly
Let's examine the fastest memory available
1ns–2ns
Registers
3ns–8ns
Level 1 Cache
6ns–20ns
Level 2 Cache
30ns–70ns
Main Memory
35µs–100µs
Solid State Disk
3ms–15ms
Fixed Rigid Disk
100ms–5s
Optical Disks (Juke boxes)
10s–3m
Magnetic Tapes (Robotic libraries)
8.3–33ms
USB Sticks
12–40ms
Removable Hard Drives
200ms
Floppy Disks
System
Online (Secondary)
Tertiary
Offline
Larger
Less Costly

Memory Hierarchy – Registers

- Storage locations available **on the processor** itself
- **Manually** managed by the assembly programmer or compiler
- *You'll become intimately familiar with registers when we do assembly programming*

Memory Hierarchy – Caches

- **What is a cache?**
  - Speed up memory accesses by storing **recently used** data closer to the CPU
  - **Closer** than main memory – on the CPU itself!
  - Although cache is much smaller than main memory, its access time is much faster!
  - Cache is **automatically** managed by the hardware memory system
    - *Clever programmers can help the hardware use the cache more effectively*

Memory Hierarchy – Caches

- **How does the cache work?**
  - Not going to discuss how caches work internally
    - If you want to learn that, take ECPE 173!
  - This class is focused on *what does the programmer need to know about the underlying system*

Memory Hierarchy – Access

- CPU wishes to **read data** (needed for an instruction)
  1. Does the instruction say it is in a register or memory?
     - If register, go get it!
  2. If in memory, send request to nearest memory (the cache)
  3. If not in cache, send request to main memory
  4. If not in main memory, send request to the disk

(Cache) Hits versus Misses

Hit

- When data is found at a given memory level (e.g. a cache)

Miss

- When data is **not** found at a given memory level (e.g. a cache)

You want to write programs that produce a lot of **hits**, not misses!

Memory Hierarchy – Cache

- Once the data is located and delivered to the CPU, it will also be saved into cache memory for future access
  - We often save more than just the specific byte(s) requested
  - Typical: Neighboring 64 bytes (called the **cache line size**)

Cache Locality

Principle of Locality

Once a data element is accessed, it is likely that a nearby data element (or even the same element) will be needed soon

Cache Locality

- **Temporal locality** – Recently-accessed data elements tend to be accessed again
  - Imagine a *loop counter*…
- **Spatial locality** - Accesses tend to cluster in memory
  - Imagine scanning through all elements in an array, or running several sequential instructions in a program

Programs with good locality **run faster** than programs with poor locality

A program that randomly accesses memory addresses (but never repeats) will gain no benefit from a cache

Recap – Cache

- **Which is bigger – a cache or main memory?**
  - Main memory
- **Which is faster to access – the cache or main memory?**
  - Cache – It is **smaller** (which is faster to search) and **closer** to the processor (signals take less time to propagate to/from the cache)
- **Why do we add a cache between the processor and main memory?**
  - Performance – hopefully frequently-accessed data will be in the faster cache (so we don't have to access slower main memory)

Recap – Cache

- **Which is manually controlled – a cache or a register?**
  - Registers are manually controlled by the assembly language program (or the compiler)
  - Cache is automatically controlled by hardware
- **Suppose a program wishes to read from a particular memory address. Which is searched first – the cache or main memory?**
  - Search the cache first – otherwise, there's no performance gain

Recap – Cache

- **Suppose there is a cache miss (data not found) during a 1 byte memory read operation. How much data is loaded into the cache?**
  - Trick question – we always load data into the cache **1 "line" at a time**.
  - Cache line size varies – 64 bytes on a Core i7 processor

Cache Q&A

- **Imagine a computer system only has main memory (no cache was present). Is *temporal* or *spatial locality* important for performance when repeatedly accessing an array with 8-byte elements?**
  - **No.** Locality is not important in a system without caching, because every memory access will take the same length of time.

Cache Q&A

- **Imagine a memory system has main memory and a 1-level cache, but each cache line size is only 8 bytes in size. Assume the cache is much smaller than main memory. Is *temporal* or *spatial locality* important for performance here when repeatedly accessing an array with 8-byte elements?**
  - Only 1 array element is loaded at a time in this cache
  - Temporal locality is important (access will be faster if the same element is accessed again)
  - Spatial locality is not important (neighboring elements are not loaded into the cache when an earlier element is accessed)

Cache Q&A

- **Imagine a memory system has main memory and a 1-level cache, and the cache line size is 64 bytes. Assume the cache is much smaller than main memory. Is *temporal* or *spatial locality* important for performance here when repeatedly accessing an array with 8-byte elements?**
  - 8 elements (64B) are loaded into the cache at a time
  - **Both** forms of locality are useful here!

Cache Q&A

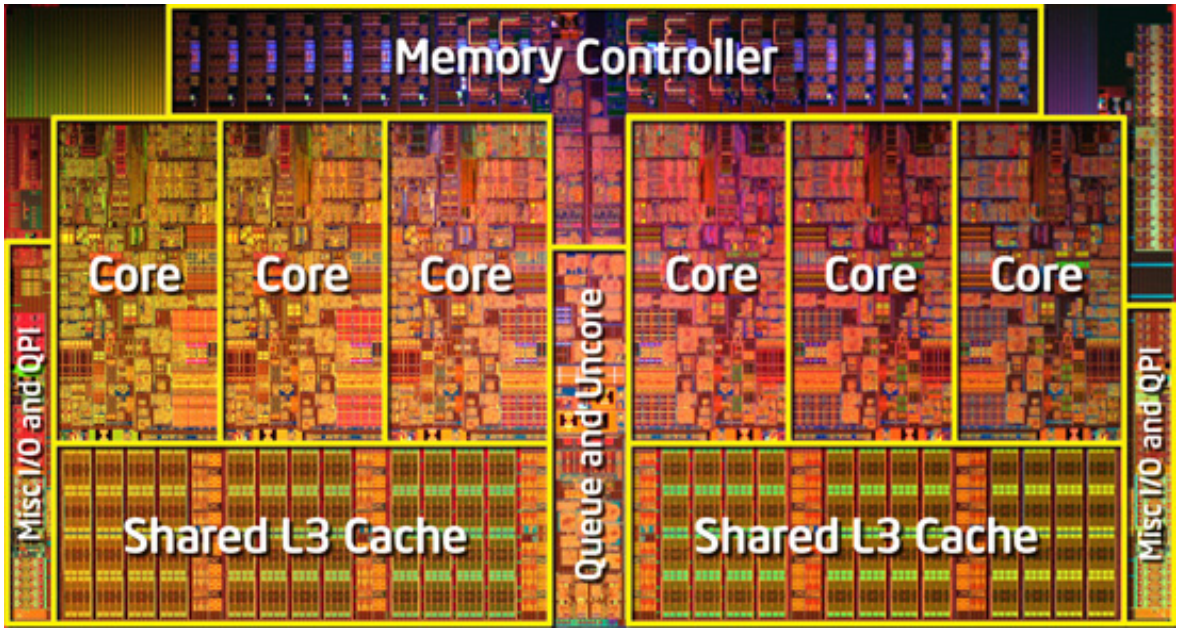
- **Imagine your program accesses a 100,000 element array (of 8 byte elements) once from beginning to end with stride 1. The memory system has a 1-level cache with a line size of 64 bytes. No pre-fetching is implemented. How many cache misses would be expected in this system?**
  - **12500** cache misses. The array has 100,000 elements. Upon a cache miss, 8 adjacent and aligned elements (one of which is the miss) is moved into the cache. Future accesses to those remaining elements should hit in the cache. Thus, only 1/8 of the 100,000 element accesses result in a miss

Cache Q&A

- **Imagine your program accesses a 100,000 element array (of 8 byte elements) once from beginning to end with stride 1. The memory system has a 1-level cache with a line size of 64 bytes. *A hardware prefetcher is implemented*. In the best-possible case, how many cache misses would be expected in this system?**
  - **1 cache miss** - This program has a trivial access pattern with stride 1. In the perfect world, the hardware prefetcher would begin guessing future memory accesses after the initial cache miss and loading them into the cache. Assuming the prefetcher can stay ahead of the program, then all future memory accesses with the trivial +1 pattern should result in cache hits

Cache Example – Intel Core i7 980x

- 6 core processor with a sophisticated multi-level cache hierarchy
- 3.5GHz, 1.17 billion transistors



Cache Example – Intel Core i7 980x

- Each processor core has its own a L1 and L2 cache
  - 32kB Level 1 (L1) data cache
  - 32kB Level 1 (L1) instruction cache
  - 256kB Level 2 (L2) cache (both instruction and data)
- The entire chip (all 6 cores) **share** a single 12MB Level 3 (L3) cache

Cache Example – Intel Core i7 980x

- Access time? (Measured in 3.5GHz clock cycles)
  - 4 cycles to access L1 cache
  - 9-10 cycles to access L2 cache
  - 30-40 cycles to access L3 cache
- Smaller caches are faster to search
  - And can also fit closer to the processor core
- Larger caches are slower to search
  - Plus we have to place them further away

Caching is Ubiquitous!

Many types of "cache" in computer science, with different meanings

Type	What Cached	Where Cached	Managed By
TLB	Address Translation (Virtual->Physical Memory Address)	On-chip TLB	Hardware MMU (Memory Management Unit)
Buffer cache	Parts of files on disk	Main memory	Operating Systems
Disk cache	Disk sectors	Disk controller	Controller firmware
Browser cache	Web pages	Local Disk	Web browser

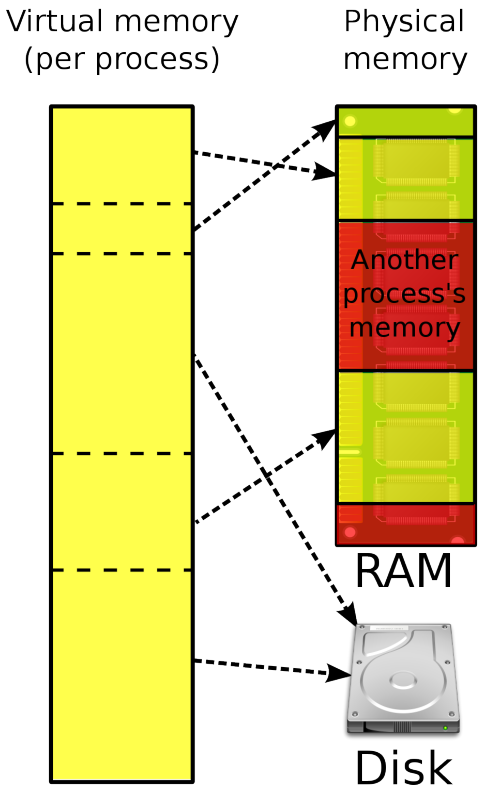
Memory Hierarchy – Virtual Memory

Virtual Memory

Virtual Memory is a BIG LIE!

- We **lie** to your application and tell it that the system is simple:
  - Physical memory is infinite! (or at least huge)
  - You can access *all* of physical memory
  - Your program starts at *memory address zero*
  - Your memory address is *contiguous* and *in-order*
  - Your memory is *only RAM* (main memory)

What the System Really Does



Why use Virtual Memory?

- We want to run multiple programs on the computer concurrently *(multitasking)*
  - Each program needs its own separate memory region, so physical resources must be divided
  - The amount of memory each program takes could vary dynamically over time (and the user could run a different mix of apps at once)
- We want to use multiple types of storage (main memory, disk) to increase performance and capacity
- We don't want the programmer to worry about this
  - Make the processor architect handle these details

Pages and Virtual Memory

- Main memory is divided into **pages** for virtual memory
  - Pages size = 4kB
  - Data is moved between main memory and disk at a page granularity
    - i.e. like the cache, we don't move single bytes around, but rather big groups of bytes

Pages and Virtual Memory

- Main memory and virtual memory are divided into equal sized pages
- The entire address space required by a process need not be in memory at once
  - Some pages can be on disk
    - Push the unneeded parts out to slow disk
  - Other pages can be in main memory
    - Keep the frequently accessed pages in faster main memory
- The pages allocated to a process do not need to be stored contiguously-- either on disk or in memory

Virtual Memory Terms

- **Physical address** – the actual memory address in the *real* main memory
- **Virtual address** – the memory address that is seen in your program
  - Special hardware/software translates virtual addresses into physical addresses!
- **Page faults** – a program accesses a virtual address that is not currently resident in main memory (at a physical address)
  - The data must be loaded from disk!
- **Pagefile** – The file on disk that holds memory pages
  - Usually twice the size of main memory

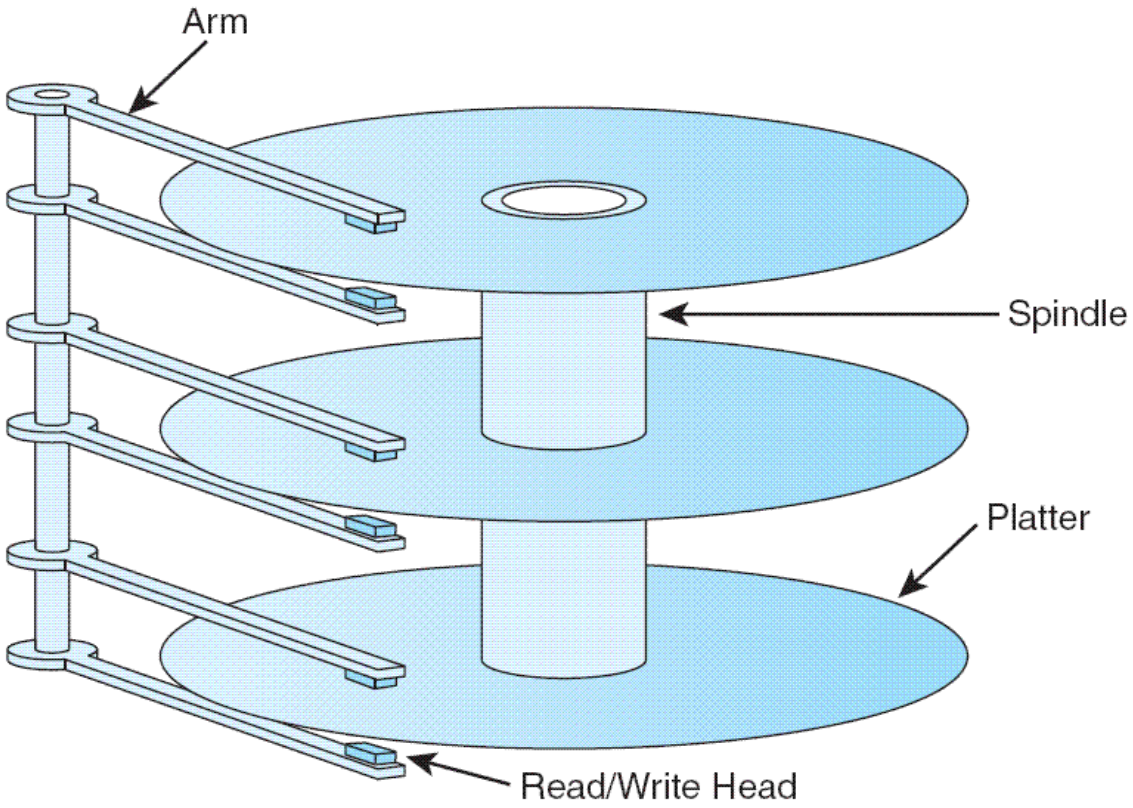
Cache Memory vs Virtual Memory

- Goal of **cache memory**
  - Faster memory access speed (**performance**)
- Goal of **virtual memory**
  - Increase memory **capacity** without actually adding more main memory
    - Data is written to disk
    - If done carefully, this can improve performance
    - If overused, performance suffers greatly!
  - Increase system flexibility when running multiple user programs (as previously discussed)

Memory Hierarchy – Magnetic Disks

Magnetic Disk Technology

- Hard disk platters are mounted on spindles
- Read/write heads are mounted on a comb that swings radially to read the disk
  - All heads move **together**!



Magnetic Disk Technology

- There are a number of *electromechanical* properties of hard disk drives that determine how fast its data can be accessed
- **Seek time** – time that it takes for a disk arm to move into position over the desired cylinder
- **Rotational delay** – time that it takes for the desired sector to move into position beneath the read/write head
- Seek time + rotational delay = **access time**

How Big Will Hard Drives Get?

- Advances in technology have defied all efforts to define the ultimate upper limit for magnetic disk storage
  - In the 1970s, the upper limit was thought to be around 2Mb/in^2
- As data densities increase, bit cells consist of proportionately fewer magnetic grains
  - There is a point at which there are too few grains to hold a value, and a 1 might spontaneously change to a 0, or vice versa
  - This point is called the **superparamagnetic limit**

How Big Will Hard Drives Get?

- **When will the limit be reached?**
- In 2006, the limit was thought to lie between 150Gb/in^2 and 200Gb/in^2 *(with longitudinal recording technology)*
- 2010: Commercial drives have densities up to 667Gb/in^2
- 2012: Seagate demos drive with 1 Tbit/in^2 density
  - *With heat-assisted magnetic recording* – they use a laser to heat bits before writing
  - Each bit is ~12.7nm in length (a dozen atoms)

Memory Hierarchy – SSDs

Emergence of Solid State Disks (SSD)

- **Hard drive advantages?**
  - Low cost per bits
- **Hard drive disadvantages?**
  - Very slow compared to main memory
  - Fragile (ever dropped one?)
  - Moving parts wear out
- Reductions in flash memory cost has created another possibility: **solid state drives** (SSDs)
  - SSDs appear like hard drives to the computer, but they store data in non-volatile **flash memory** circuits
  - Flash is **quirky!** Physical limitations pose engineering challenges…

Flash Memory

- Typical flash chips are built from dense arrays of NAND gates
- Different from hard drives – we **can't** read/write a single bit (or byte)
  - **Reading or writing?** Data must be read from an entire **flash page** (2kB-8kB)
    - Reading much faster than writing a page
    - It takes some time before the cell charge reaches a stable state
  - **Erasing?** An entire **erasure block** (32-128 pages) must be erased (set to all 1's) first before individual bits can be written (set to 0)
    - Erasing takes two orders of magnitude more time than reading

Flash-based Solid State Drives (SSDs)

Advantages

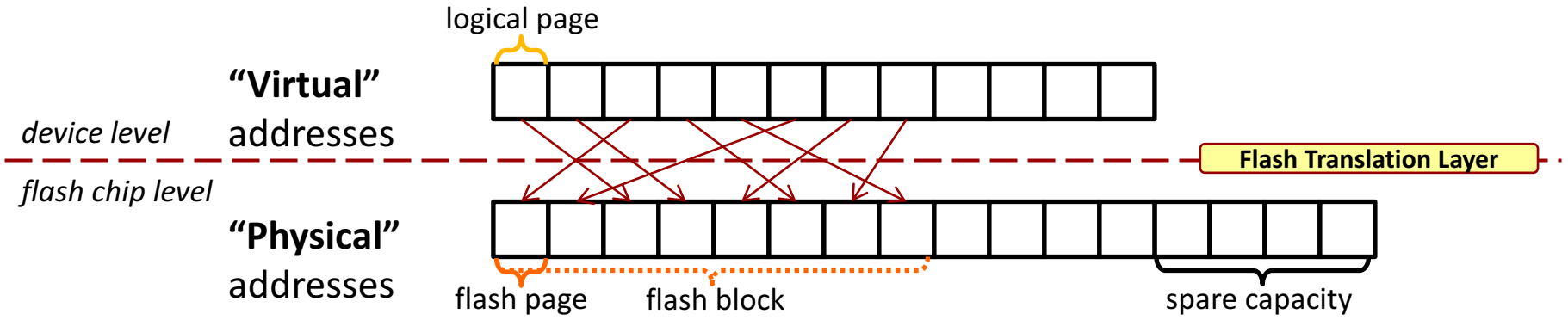
- Same block-addressable I/O interface as hard drives
- No mechanical latency
  - Access latency is independent of the access pattern
    - Compare this to hard drives
- Energy efficient (no disk to spin)
- Resistant to extreme shock, vibration, temperature, altitude
- Near-instant start-up time

Challenges

- Limited endurance and the need for **wear leveling**
- Very slow to erase blocks (needed before reprogramming)
  - Erase-before-write
- Read/write asymmetry
  - Reads are faster than writes

Flash Translation Layer

- **Flash Translation Layer (FTL)**
  - Necessary for flash reliability and performance
  - **"Virtual" addresses** seen by the OS and computer
  - **"Physical" addresses** used by the flash memory
- Perform writes out-of-place
  - Amortize block erasures over many write operations
- Wear-leveling
  - Writing the same "virtual" address repeatedly won't write to the same physical flash location repeatedly!



```
#!/bin/bash
function happy_birthday(){
        get cake
        light candles
        open gifts
        i=0
    while cake_count>i

Output=''
for i in {1..4}
do output=$output"Happy Birthday"
 if [ $i -eq 3 ]
 then
  output=$output"Dear Aaron"
else
 output=$output
     "to you"
echo -e $output
 3
```

```
if [$date +%d%b]-eq '22 Oct'
  then
     happy_birthday
  echo "Happy Birthday Aaron!"

  else
    y=$(date --date'22 oct'+%j)
    x=$(date +%j)
    ((z=${y}-${x}))

      echo  "$(z) days
        until Aaron's
        next birthday!"

        fi
        done
```