Computer Systems and Networks

ECPE 170 – Dr. Pallipuram– University of the Pacific

Memory Hierarchy
(Performance Optimization)

Slides are courtesy of Dr. Shafer

Lab Schedule

Activities

↗ **Labs**

   ↗ **Lab 6 – Perf Optimization**

   ↗ **Lab 7 – Memory Hierarchy**

Assignments Due

↗ **Lab 6**

   ↗ **Due by OCT 17th 5 PM**

↗ **** Video Presentation #1 ****

   ↗ **Released TODAY!**

   ↗ **Due OCT 14th 11:59pm**

Video Presentation #1

↗ Prepare a 8-10 minute **recorded video** demonstrating "lab-like" technical skills

↗ Topics chosen from Lab 1 – Lab 5

↗ The recording should include both your computer monitor and your video and voice as a narrator

↗ Graded for **both technical accuracy** and **communication skills**
  - ↗ **Technical Content** (50% of grade) – Does the video provide correct technical information?
  - ↗ **Verbal Explanation** (50% of grade) – Does the video explain why an action is being taken, in sufficient detail for an engineering student who has not yet taken ECPE 170?

A video that presents perfect technical content but has no explanation about what is being done or why it is being done, will only receive half credit.

Recap

Malloc − 1D

```
int *array;  //array of integers
```

array (pointer variable)

value:	**60**
pointer addr: 32	

```
array = (int *)malloc(sizeof(int)*5);
```

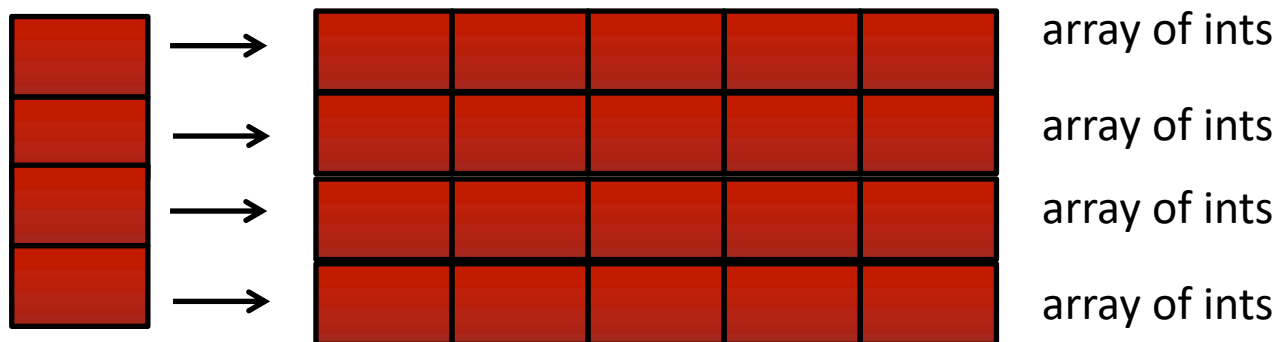| address:	**60**	**64**	**68**	**72**	**76**
value:	array[0]	array[1]	array[2]	array[3]	array[4]

Malloc – 2D
Allocate 4x5 integers

```
int **array; //a double pointer

array = (int **)malloc(sizeof(int *)*4);


for(i=0;i<4;i++)
  array[i] = (int *)malloc(sizeof(int)*5);
```

array of ints

array of ints

array of ints

array of ints

an array of integer pointers

Malloc – 3D

```
int ***array; //a triple pointer
```

an array of
double pointers

a matrix of
single pointers

a 'cuboid' of integers
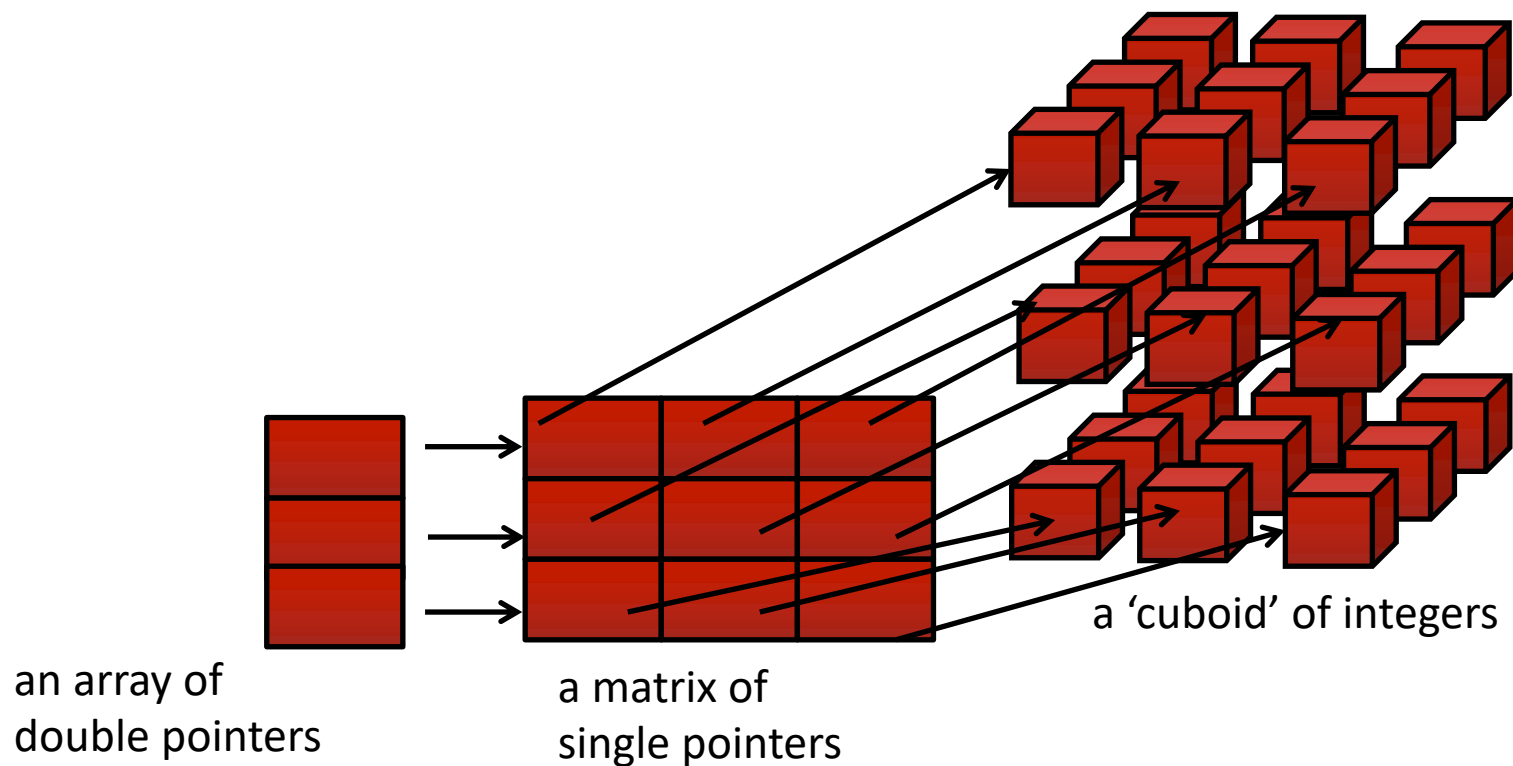
Problem 1 – Array Addresses

↗ Write a C code snippet to print the **addresses** of elements in a 2-D array: `array[row][col]` Visit this array in row-major format (row 0, then row 1, and so on..)
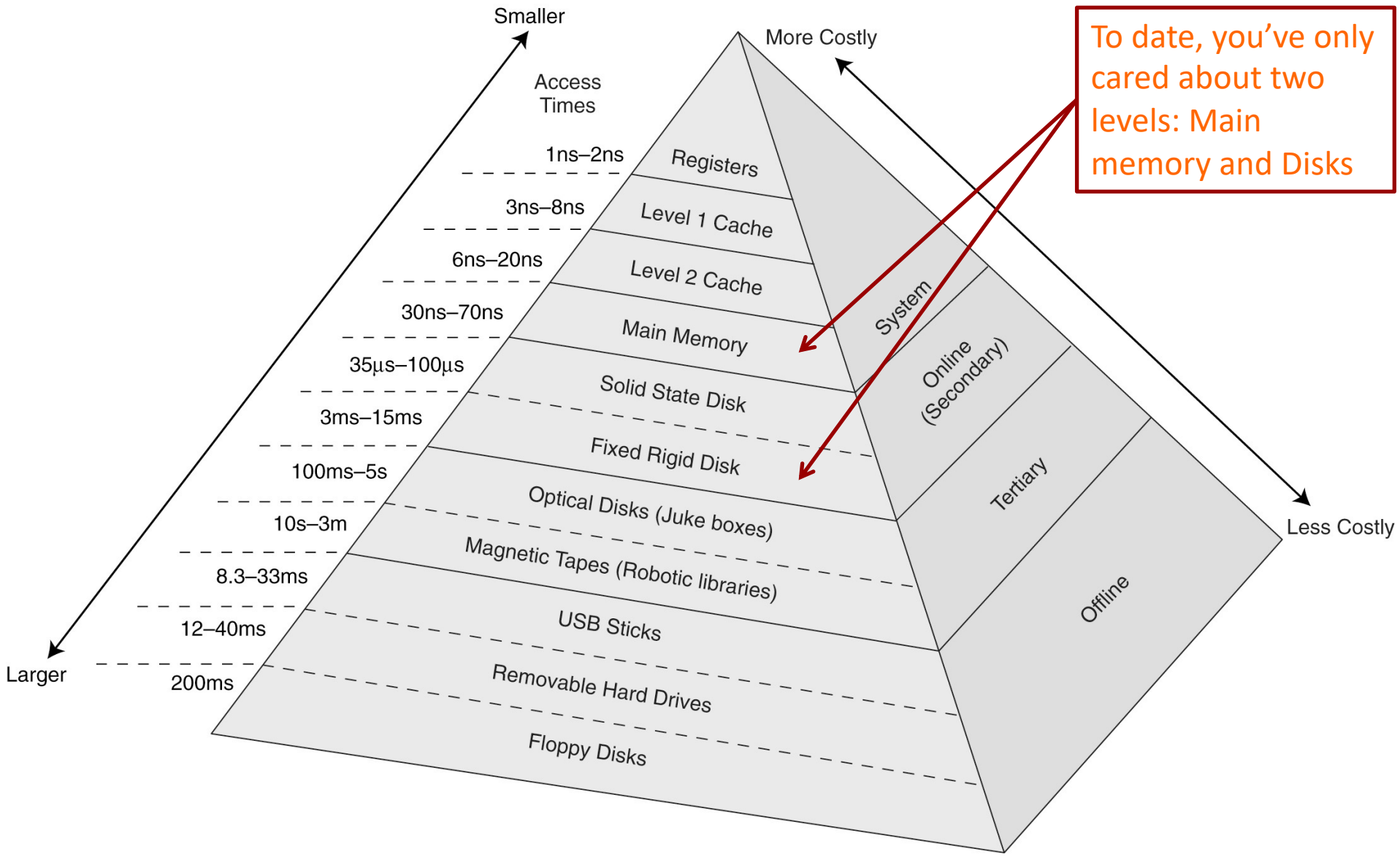
P1

Memory Hierarchy

Memory Hierarchy

Goal as system designers:

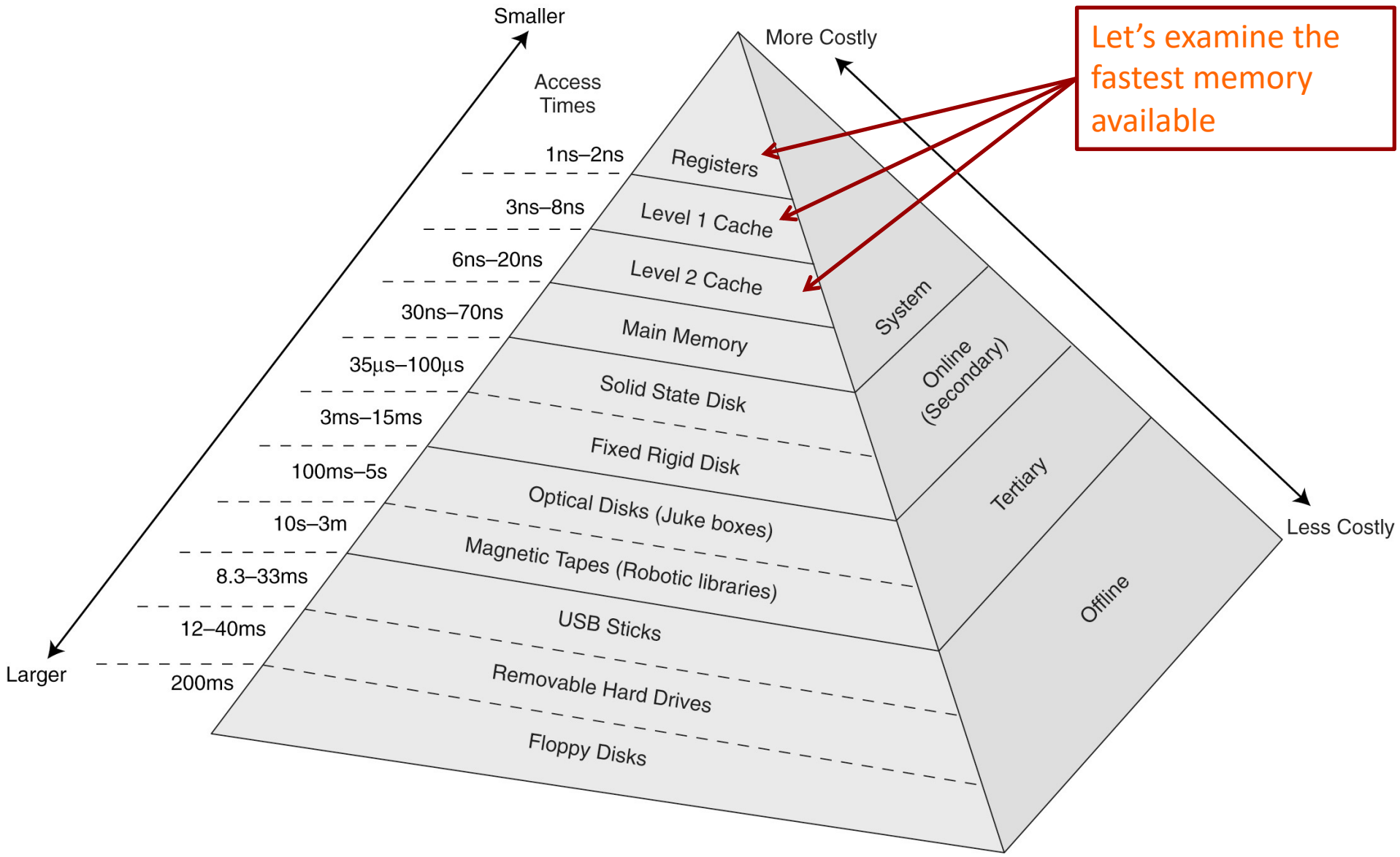Fast Performance **and** Low Cost

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

Smaller

More Costly

Access Times

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

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

Larger

Less Costly

Smaller

More Costly

Let's examine the fastest memory available

Access Times

Access Time	Memory Level
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

Less Costly

Larger

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 **<u>recently used</u>** 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)
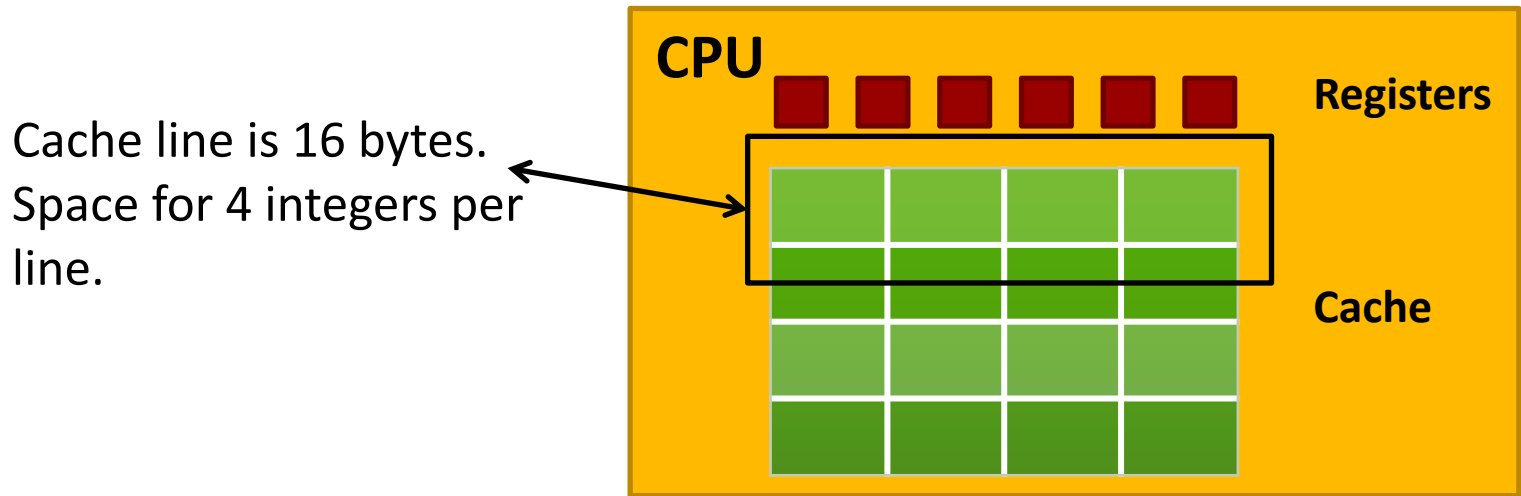
Miss

↗ 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!

Cache Example

➚ Hypothetical cache for pseudocode that reads all elements of `a[]`
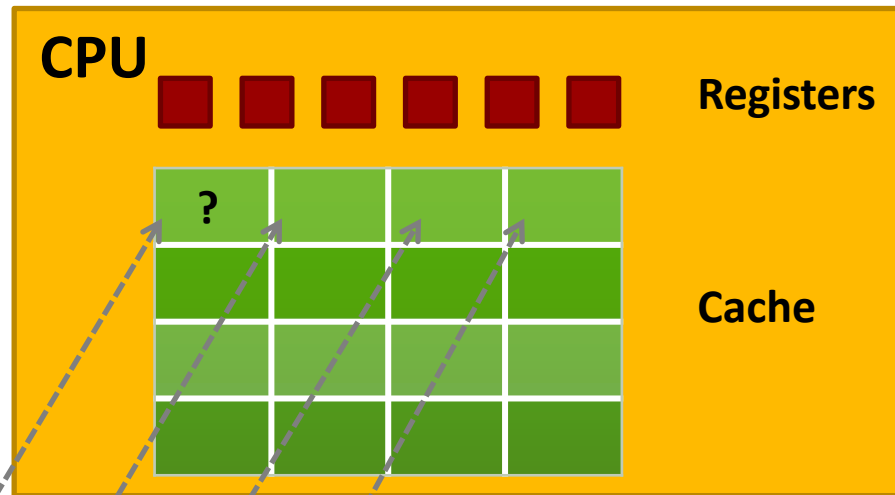
```
for(i=0; i<30; i++)
{
  a[i];
}
```

CPU

Registers

Cache line is 16 bytes.
Space for 4 integers per line.

Cache

How does CPU get array elements a[0], a[1], a[2], …?

```
for(i=0;i<30;i++)
    a[i];
```

Main memory (RAM)

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
a[10]	a[11]	a[12]	a[13]	a[14]	a[15]	a[16]	a[17]	a[18]	a[19]
a[20]	a[21]	a[22]	a[23]	a[24]	a[25]	a[26]	a[27]	a[28]	a[29]

CPU

Registers

Cache line is 16 bytes. Space for 4 integers per line.

?

Cache

Access a[0]

1. Query the Cache for a[0]
2. Result: a[0] not present – **Cache Miss**!
3. Fetch a[0] <u>and entire cache line</u> from main memory

Main memory (RAM)

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
a[10]	a[11]	a[12]	a[13]	a[14]	a[15]	a[16]	a[17]	a[18]	a[19]
a[20]	a[21]	a[22]	a[23]	a[24]	a[25]	a[26]	a[27]	a[28]	a[29]

# 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

- ↗ In this example: cache line width is 16 bytes (space for 4 integers), providing 3 hits for every 4 integers

  - ↗ If cache width is for `m` integers and the data access is contiguous, then only 1 miss for every `m` integer accesses

  - ↗ Typical on modern CPUs: Cache line size is 64 bytes

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

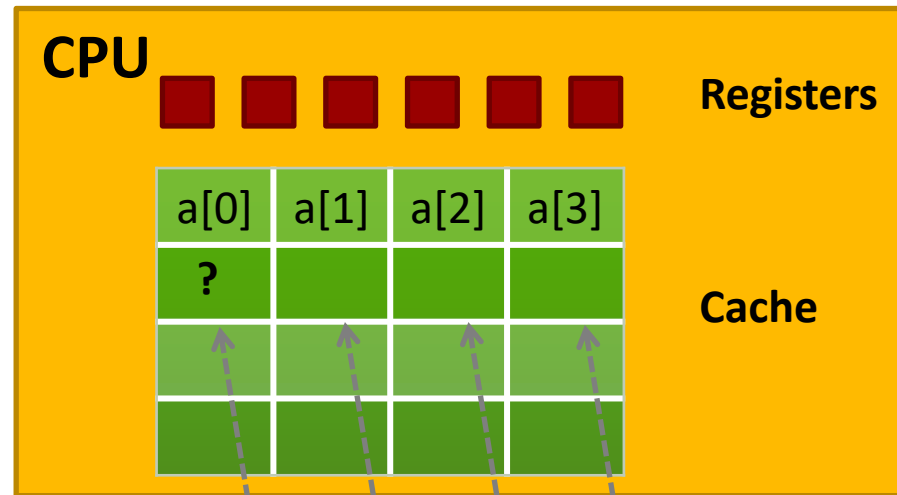Cache line is 16 bytes.
Space for 4 integers per line.

CPU

Registers
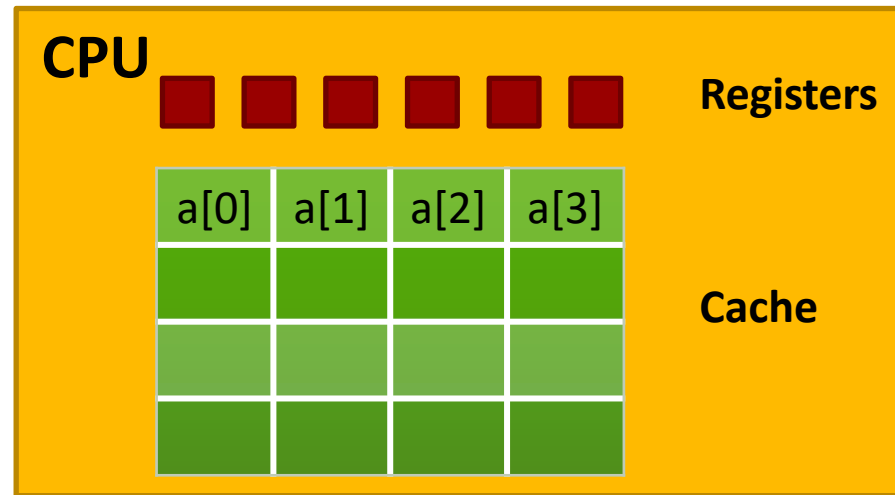
a[0]	a[1]	a[2]	a[3]

Cache

1. Access a[1] – **Cache Hit!**
2. Access a[2] – **Cache Hit!**
3. Access a[3] – **Cache Hit!**

Main memory (RAM)

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
a[10]	a[11]	a[12]	a[13]	a[14]	a[15]	a[16]	a[17]	a[18]	a[19]
a[20]	a[21]	a[22]	a[23]	a[24]	a[25]	a[26]	a[27]	a[28]	a[29]

Cache line is 16 bytes. Space for 4 integers per line.

CPU

Registers

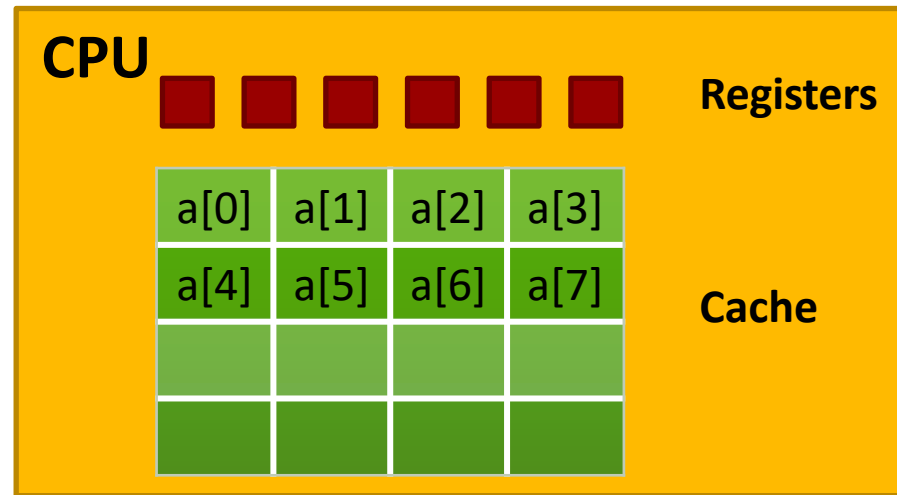| a[0]	a[1]	a[2]	a[3]
?			

Cache

Access a[4]

1. Query the Cache for a[4]
2. Result: a[4] not present – **Cache Miss**!
3. Fetch a[4] and entire cache line from main memory

Main memory (RAM)

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
a[10]	a[11]	a[12]	a[13]	a[14]	a[15]	a[16]	a[17]	a[18]	a[19]
a[20]	a[21]	a[22]	a[23]	a[24]	a[25]	a[26]	a[27]	a[28]	a[29]

Cache line is 16 bytes.
Space for 4 integers per line.

CPU

Registers

a[0]	a[1]	a[2]	a[3]
a[4]	a[5]	a[6]	a[7]

Cache

1. Access a[5] – **Cache Hit!**
2. Access a[6] – **Cache Hit!**
3. Access a[7] – **Cache Hit!**

Main memory (RAM)

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
a[10]	a[11]	a[12]	a[13]	a[14]	a[15]	a[16]	a[17]	a[18]	a[19]
a[20]	a[21]	a[22]	a[23]	a[24]	a[25]	a[26]	a[27]	a[28]	a[29]

Cache Locality

- ↗ **Spatial locality** - Accesses tend to cluster in memory
    - ↗ Imagine scanning through all elements in an array, or running several sequential instructions in a program

- ↗ **Temporal locality** – Recently-accessed data elements tend to be accessed again
    - ↗ Imagine a *loop counter*…
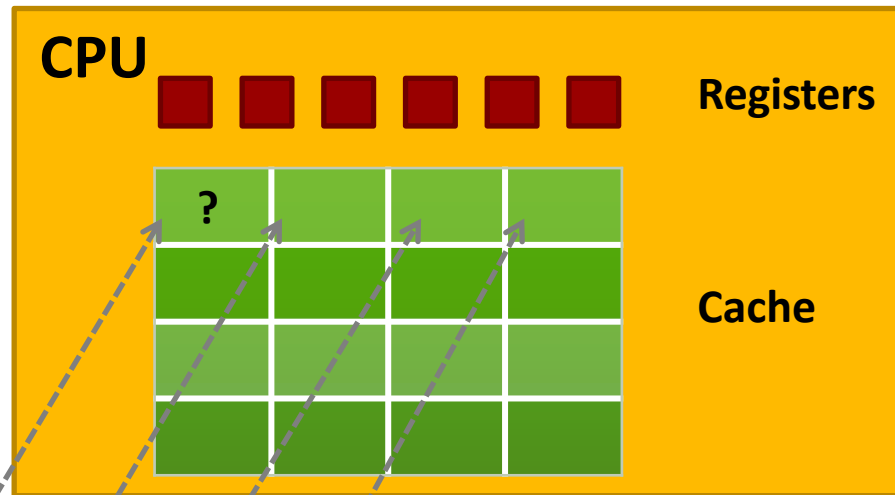
In Class Cache Simulation

Problem 2

↗ On a computer system with a cache line width of 16 bytes, how many cache hits will this code get? Assume `sizeof(int)` is 4.

```
int a[24];
int sum=0;
for(i=0;i<24;i=i+4)
{
    sum += a[i];
}
```

Stride!

P2
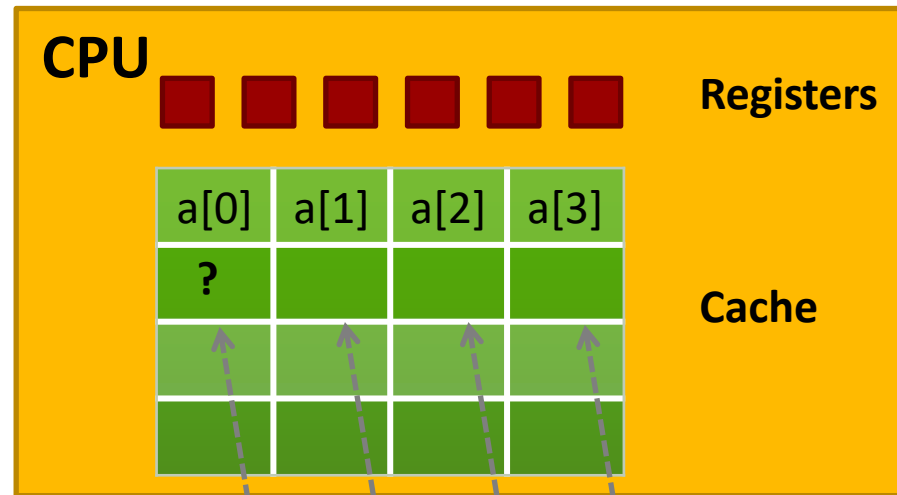
Cache line is 16 bytes. Space for 4 integers per line.

CPU

Registers

Cache

Access a[0]

1. Query the Cache for a[0]
2. Result: a[0] not present – **Cache Miss**!
3. Fetch a[0] and entire cache line from main memory

Main memory (RAM)

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
a[10]	a[11]	a[12]	a[13]	a[14]	a[15]	a[16]	a[17]	a[18]	a[19]
a[20]	a[21]	a[22]	a[23]	a[24]	a[25]	a[26]	a[27]	a[28]	a[29]

Cache line is 16 bytes.
Space for 4 integers per line.

CPU

Registers

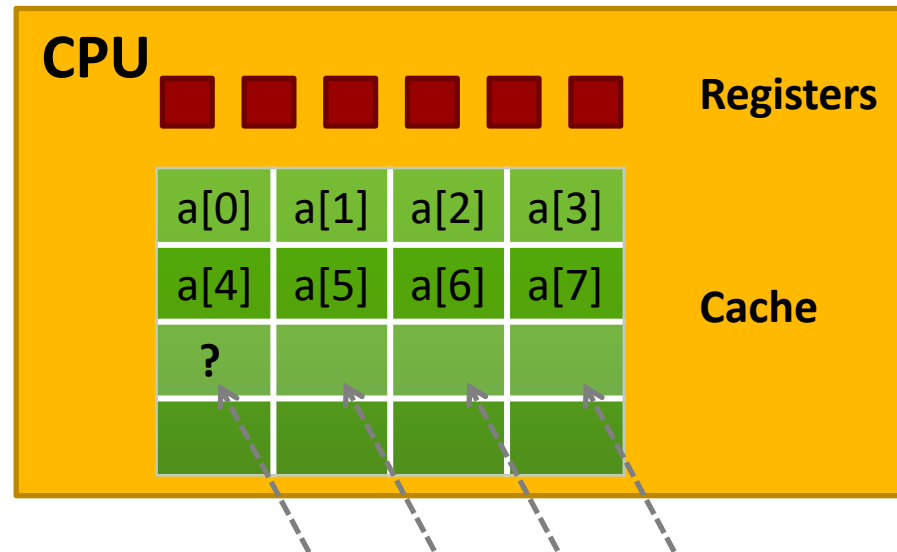| a[0]	a[1]	a[2]	a[3]
?			

Cache

Access a[4]

1. Query the Cache for a[4]
2. Result: a[4] not present – **Cache Miss**!
3. Fetch a[4] <u>and entire cache line</u> from main memory

Main memory (RAM)

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
a[10]	a[11]	a[12]	a[13]	a[14]	a[15]	a[16]	a[17]	a[18]	a[19]
a[20]	a[21]	a[22]	a[23]	a[24]	a[25]	a[26]	a[27]	a[28]	a[29]

Cache line is 16 bytes. Space for 4 integers per line.

CPU

Registers

a[0]	a[1]	a[2]	a[3]
a[4]	a[5]	a[6]	a[7]
?			

Cache

Access a[8]

1. Query the Cache for a[8]
2. Result: a[8] not present – **Cache Miss**!
3. Fetch a[8] <u>and entire cache line</u> from main memory

Main memory (RAM)

| a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
a[10]	a[11]	a[12]	a[13]	a[14]	a[15]	a[16]	a[17]	a[18]	a[19]
a[20]	a[21]	a[22]	a[23]	a[24]	a[25]	a[26]	a[27]	a[28]	a[29]

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

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

Cache Example – Intel Core i7 980x

↗ 6 core processor with a sophisticated multi-level cache hierarchy

↗ 3.5GHz, 1.17 billion transistors
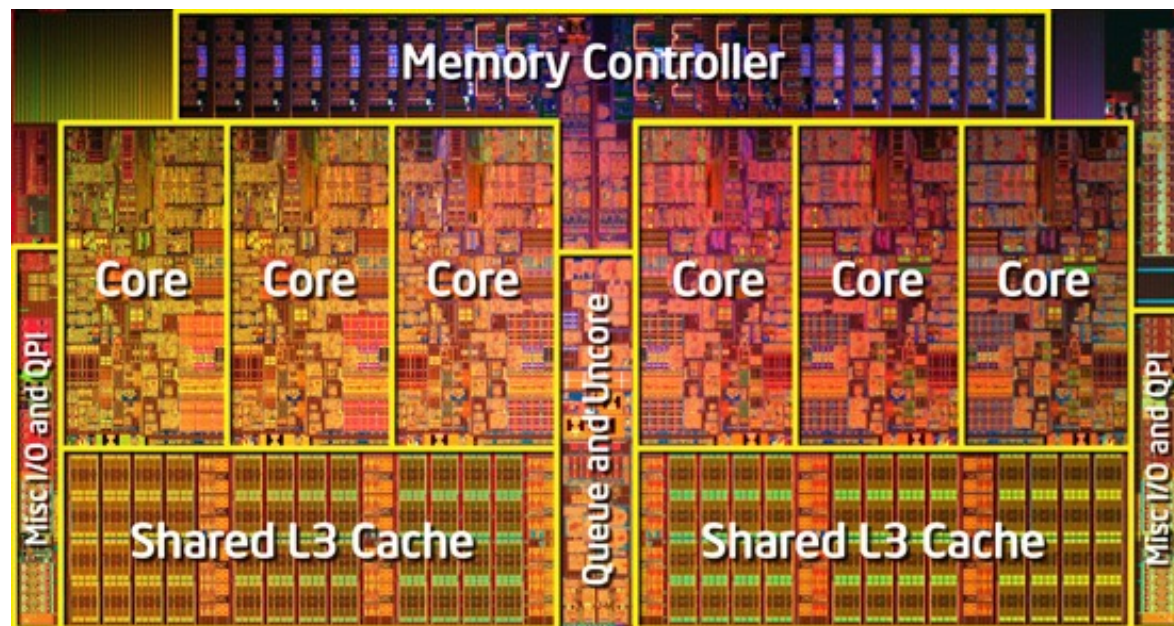
Cache Example – Intel Core i7 980x

➚ 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

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

Problem 3

➚ **Imagine a computer system <u>only has main memory</u> (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.

P3

Problem 4

↗ **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)

P4

Problem 5

➚ **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. 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

P5

Problem 6

↗ **Which code will have more cache hits? Assume array size larger than cache**

(A)

```
for (i=0;i<row;i++)
  for(j=0;j<col;j++)
    sum+=array[i][j];
```

(B)

```
for (j=0;j<col;j++)
  for(i=0;i<row;i++)
    sum+=array[i][j];
```

P6

Memory Hierarchy – Virtual Memory

Virtual Memory

Virtual Memory is a <u>BIG LIE</u>!

↗ 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

Virtual memory (per process) Physical memory

Another process's memory

RAM

Disk
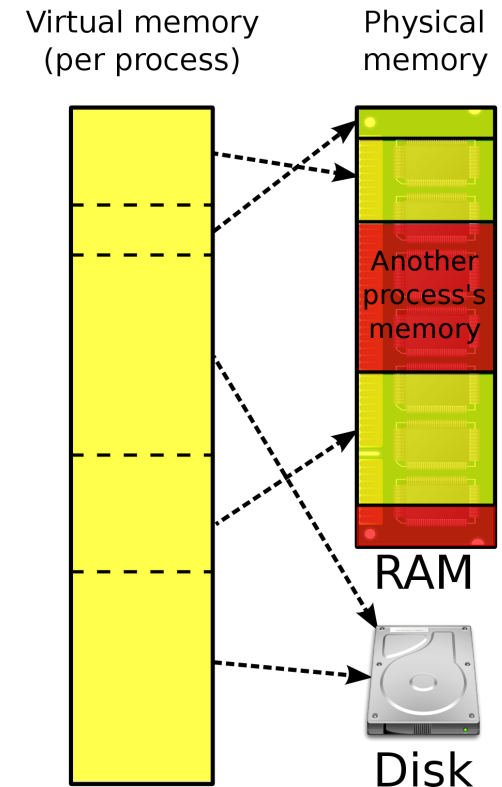
Why use Virtual Memory?

➚ 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)