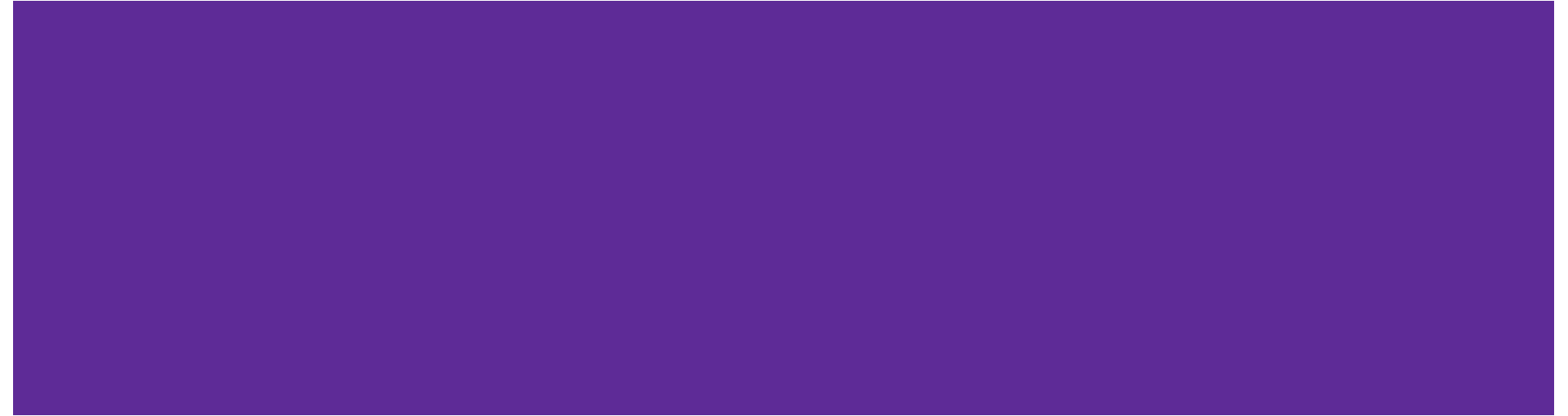


CSE 374 Programming Concepts and Tools

Lecture 16 - Buffer Overflow, Memory Architecture



**HW4 Released,
Due Monday night**

This Week

Today: Memory Architecture

Friday: C++ Intro

Next Week

Monday: C++ Classes

Wednesday: C++ Class Details

Friday: C++ Templates, STL

Today

Buffer Overflow

- Definition
- Security Vulnerabilities
- Examples

Memory Hierarchy

- Digging deeper into memory
- Caches and Locality
 - Spatial and temporal locality

Buffer Overflow

Review: General Memory Layout

Stack

- Local variables (procedure context)

Heap

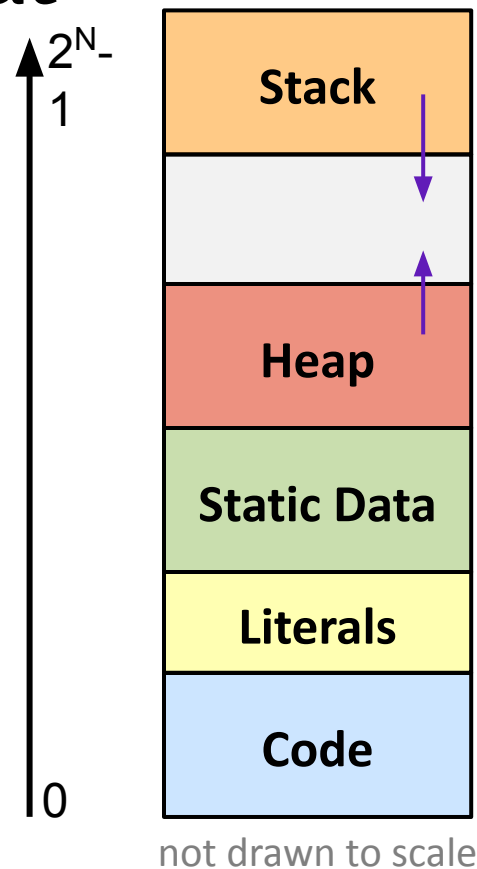
- Dynamically allocated as needed
- `malloc()`, `calloc()`, `new`, ...

Statically allocated Data

- Read/write: global variables (Static Data)
- Read-only: string literals (Literals)

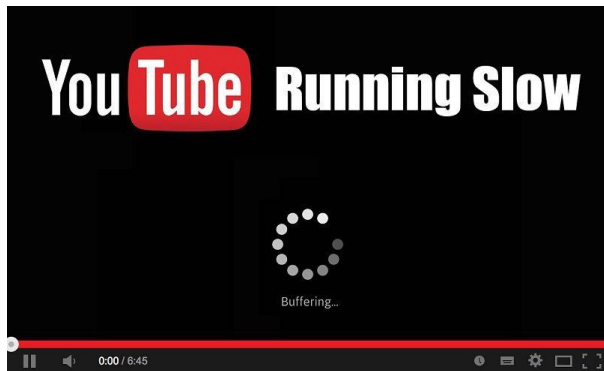
Code/Instructions

- Executable machine instructions
- Read-only



What Is a Buffer?

- A **buffer** is an array used to temporarily store data
- You've probably seen "video buffering..."
 - The video is being written into a buffer before being played
- Buffers can also store user input



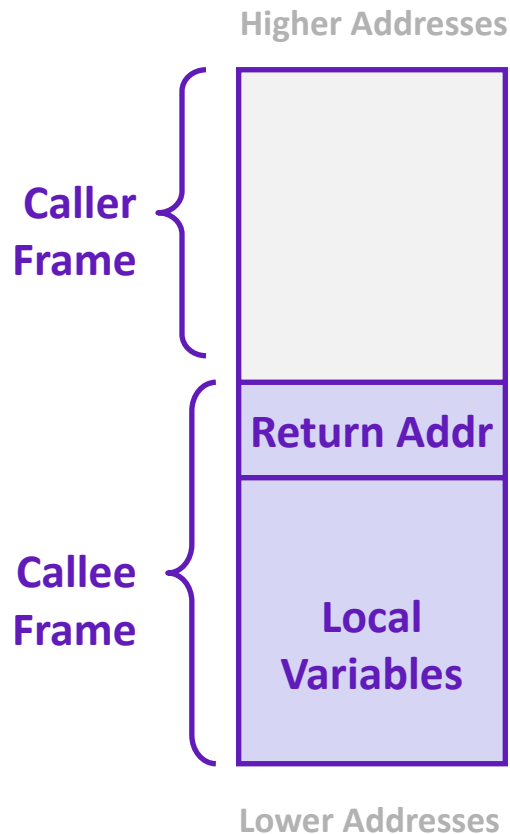
Function Stack Frame

Stack stores active functions & local variables.

Each function call creates a frame:

- **Caller's** Stack Frame
- Current/ **Callee** Stack Frame
 - **Return address**: the next instruction after the function call in the program
 - This is how the callee returns to the caller!
 - Local variables
(if they can't be kept in CPU registers)

This general idea applies to Linux, Windows, and macOS



Buffer Overflow

C does not check array bounds

- Many Unix/Linux/C functions don't check argument sizes
- Allows overflowing (writing past the end) of buffers (arrays)

“Buffer Overflow” = Writing past the end of an array

Characteristics of the traditional Linux memory layout provide opportunities for malicious programs

- Stack grows “backwards” in memory
- Data and instructions both stored in the same memory

Example

```
#include <stdio.h>
void echo();
int main() {
    printf("Enter a string: ");
    echo();
    return 0;
}
void echo() {
    char buf[8];
    gets(buf);
    printf("%s", buf);
}
```

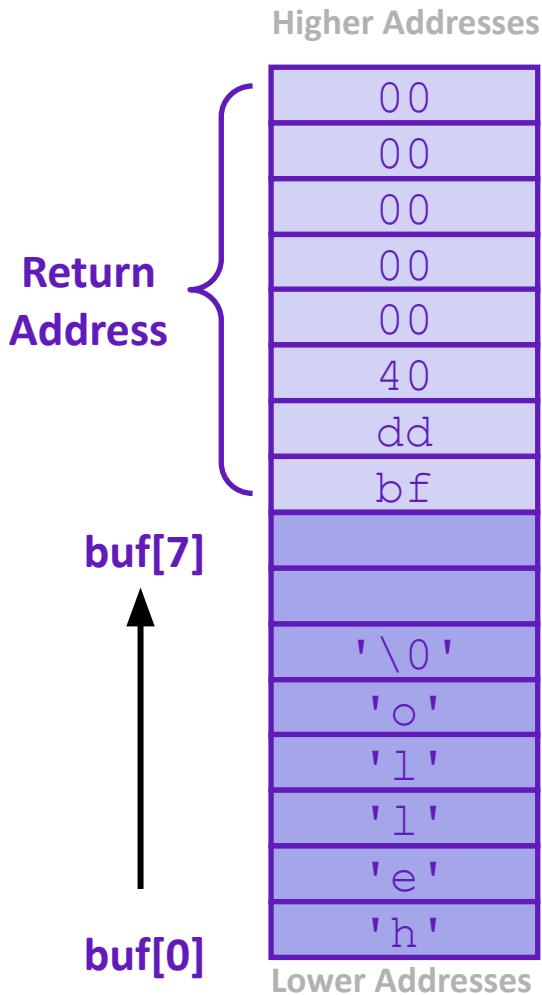
[gets\(\) documentation](#)

Example

Stack grows *down* towards lower addresses

Buffer grows *up* towards higher addresses

If we write past the end of the array, we overwrite data on the stack!

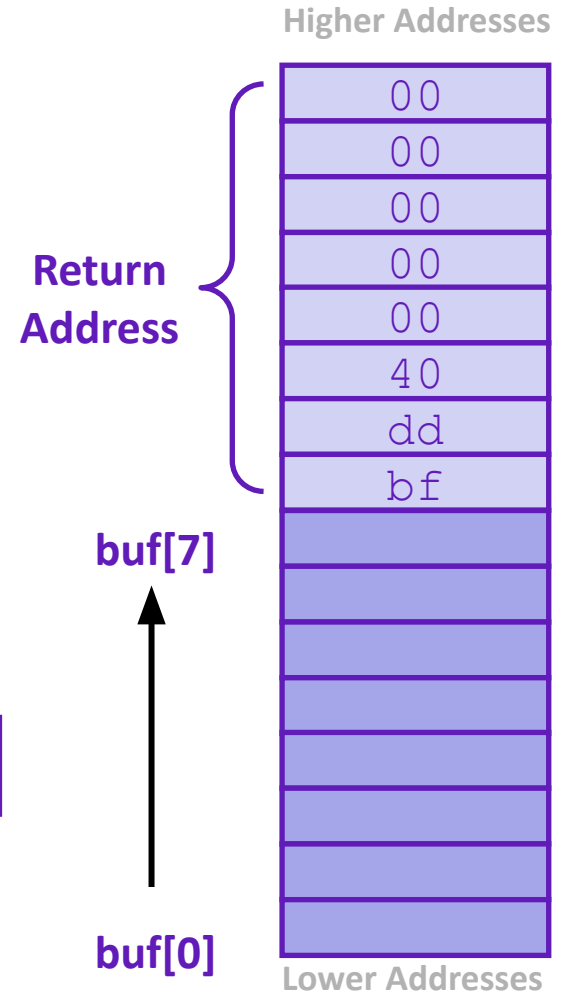


Enter input: hello

No overflow! 😊

Example

So, what happens if our input is too long?

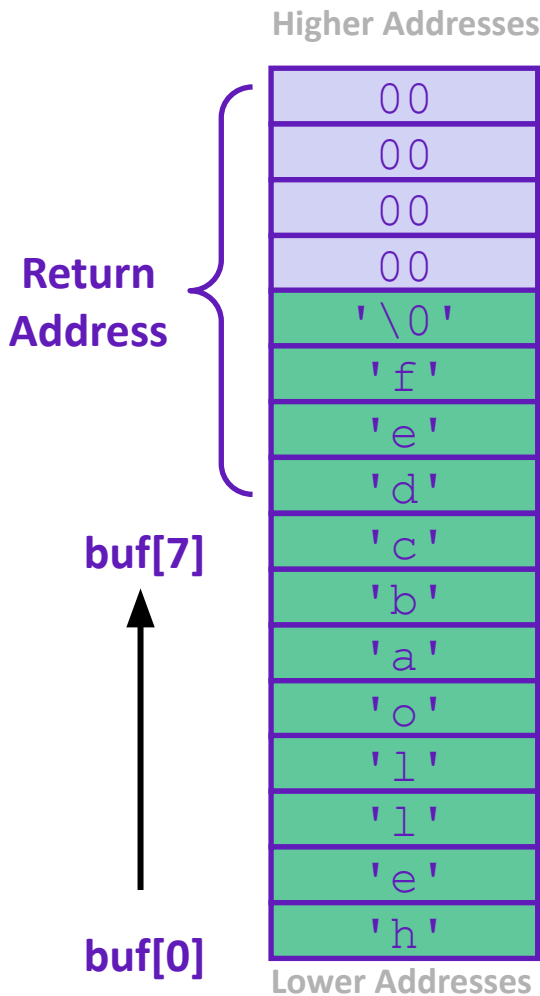


Enter input: helloabcdef

Example

If we write past the end of the array, we overwrite data on the stack!

- The data overwritten can be quite arbitrary. If the data overwritten is stack frame “bookkeeping” data or something like a function pointer, things could go bad.



Enter input: helloabcdef

Buffer overflow! 😞

Buffer Overflow in a Nutshell

Buffer overflows on the stack can overwrite “interesting” data

- Attackers just choose the right inputs

Simplest form (sometimes called “stack smashing”)

- Unchecked length on string input into bounded array causes overwriting of stack data
- [Return-oriented programming](#)
- Try to change the return address of the current procedure

Why is this a big deal?

- It was the #1 *technical* cause of security vulnerabilities
- (#1 *overall* cause is social engineering / user ignorance)

Polling Time!

Please answer in the [LP16 assignment on Gradescope](#)!

If the beginning of a local buffer is **8 bytes below the return address** of the current stack frame, *how many bytes* total do we need to write into the buffer to **completely overwrite** all of the return address?

Hint: What is the size in bytes of an address on a *64-bit* machine? (Think *word* size.)

- 8
- 9
- 12
- 16

Discussion:

Why is altering the return address so catastrophic?

Think about it on your own for 30 seconds, then discuss with a neighbor

Food for thought:

- What happens when the function tries to return after the return address is overwritten?
- What else might an attacker *inject* into the system in the buffer flow attack?

Demo: Code Injection (if time)

Exploits Based on Buffer Overflows

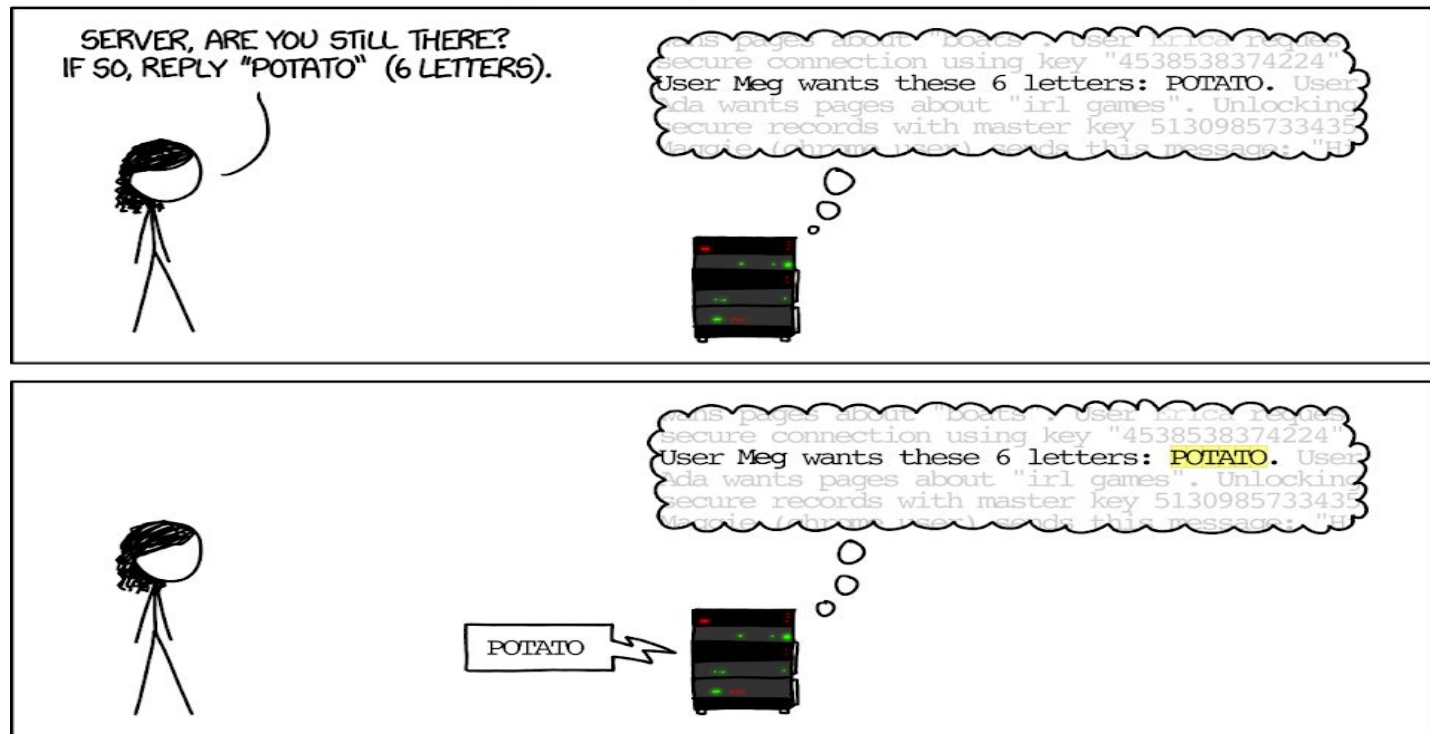
Examples across the decades

- Original “Internet worm” (aka [Morris Worm](#)) (1988)
 - Later became one of the founders of YCombinator.
- Heartbleed (2014, affected 17% of servers)
 - Similar issue in Cloudbleed (2017)
- Hacking embedded devices
 - Cars, Smart homes, Planes

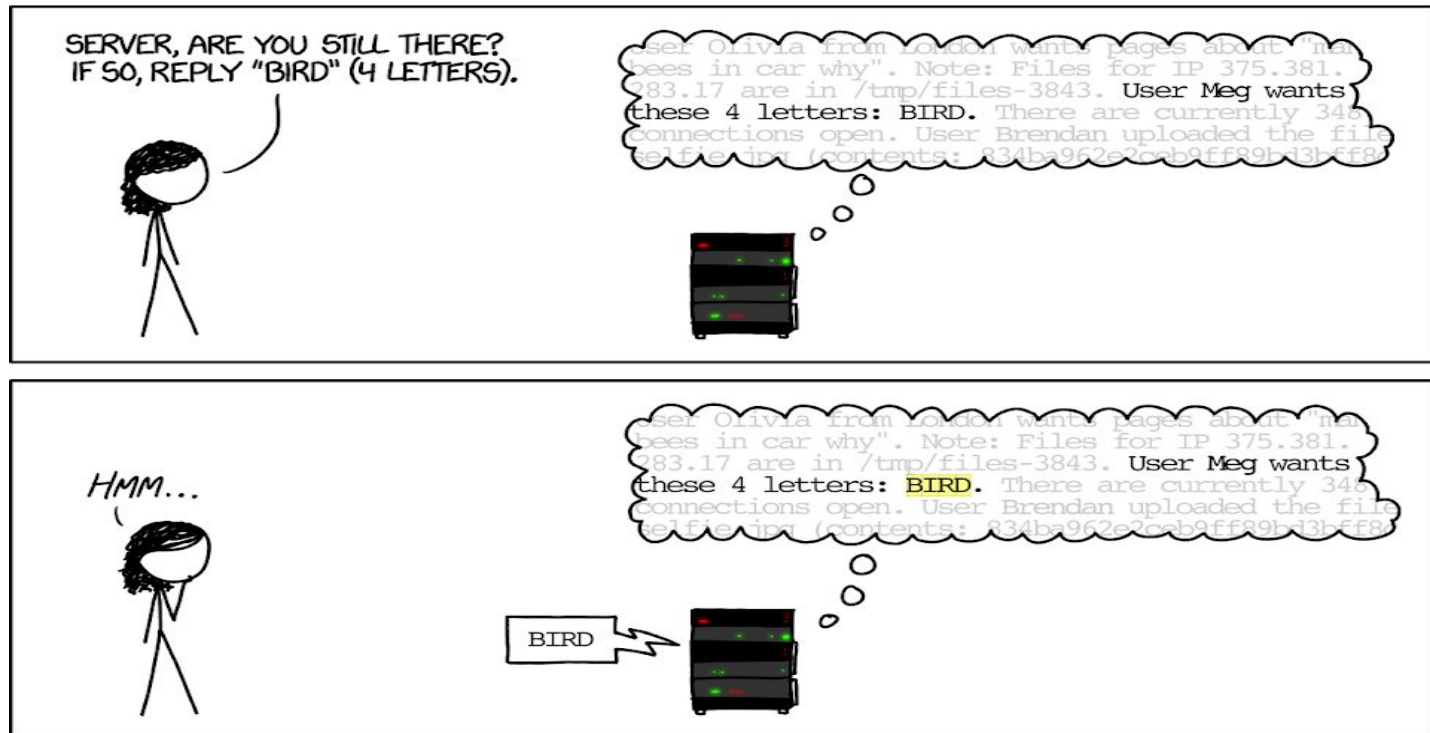
Buffer overflow bugs can allow attackers to execute arbitrary code on victim machines

Example: Heartbleed

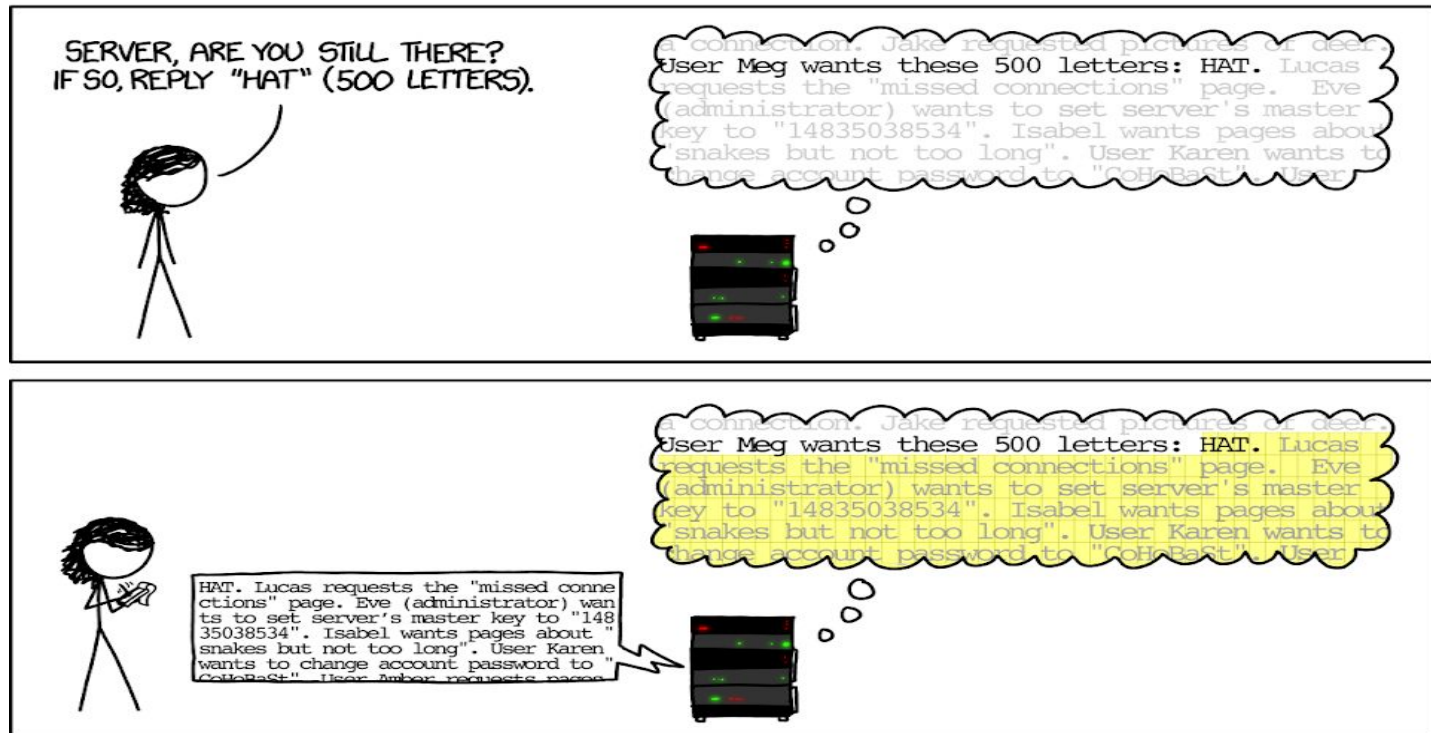
HOW THE HEARTBLEED BUG WORKS:



Example: Heartbleed



Example: Heartbleed



Heartbleed (2014)

Buffer over-read in OpenSSL

- Open source security library
- Bug in a small range of versions

“Heartbeat” packet (part of Heartbeat ext.)

- Specifies length of message
- Server echoes it back
- Library just “trusted” this length
- Allowed attackers to read contents of memory anywhere they wanted

Est. 17% of Internet affected



Heartbeat – Normal usage



Heartbeat – Malicious usage



By FenixFeather - Own work, CC BY-SA 3.0,
<https://commons.wikimedia.org/w/index.php?curid=32276981>

Hacking Cars

UW CSE [research from 2010](#) demonstrated wirelessly hacking a car using buffer overflow

Overwrote the onboard control system's code

- Disable brakes
- Unlock doors
- Turn engine on/off



Hacking DNA Sequencing Tech

- Potential for malicious code to be encoded in DNA!
- Attacker can gain control of DNA sequencing machine when malicious DNA is read
- Ney et al. (2017)
 - <https://dnasec.cs.washington.edu/>

Computer Security and Privacy in DNA Sequencing

Paul G. Allen School of Computer Science & Engineering, University of Washington

There has been rapid improvement in the cost and time necessary to sequence and analyze DNA. In the past decade, the cost to sequence a human genome has decreased 100,000 fold or more. This rapid improvement was made possible by faster, massively parallel processing. Modern sequencing techniques can sequence hundreds of millions of DNA strands simultaneously, resulting in a proliferation of new applications in domains ranging from personalized medicine, ancestry, and even the study of the microorganisms that live in your gut.

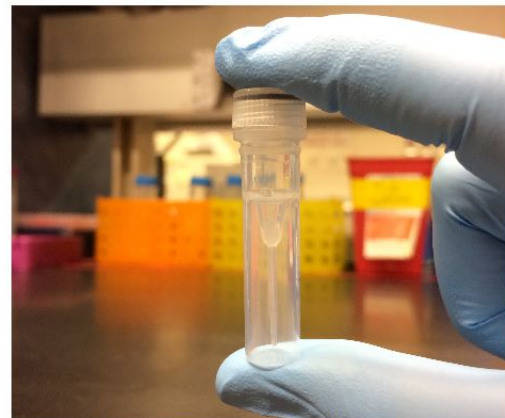


Figure 1: Our synthesized DNA exploit

Equifax (2017)

Very similar to Heartbleed.

Exploited the Content-Type HTTP header that would cause a buffer overflow.

Attackers gained access to ~147 million people's worth of personal information!

- Names
- Birth dates
- SSNs
- Driver's license
- Credit card information



Dealing with Buffer Overflow Attacks

Avoid vulnerabilities in the first place

- Use library functions that limit string length
 - fgets instead of gets
 - strncpy instead of strcpy
- *Use a language that makes them impossible*

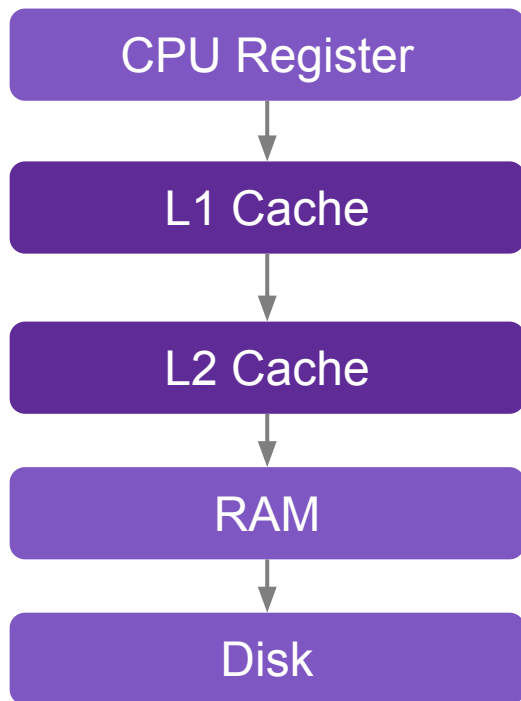
```
-bash@17[main]$ gcc -Wall -std=c11 -o echo echo.c
echo.c:13:5: warning: 'gets' is deprecated: This function is provided for compatibility reasons only. Due to security concerns inherent in the design of gets(3), it is highly recommended that you use fgets(3) instead. [-Wdeprecated-declarations]
     gets(buf);
     ^
```

System-level protections

- Make stack non-executable
- Have compiler insert “stack canaries” (just like a networking header checksum)
- Put a special value between buffer and return address
- Check for corruption before leaving function
- Randomized Stack offsets

Memory Hierarchy

Review: Memory Architecture

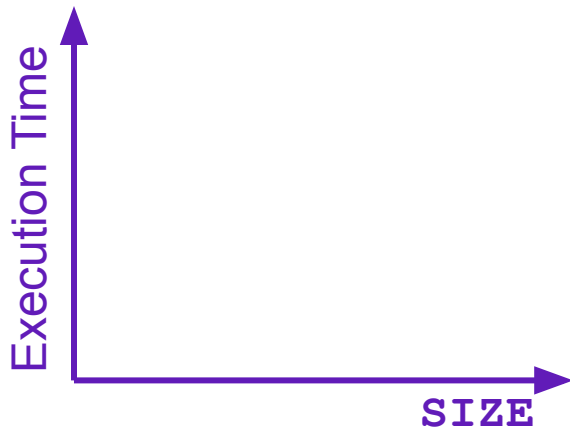


What is this?	Typical Size	Time
Part of the brain of the computer!	64 bits	≈free
Extra memory to make accessing it faster	128KB	0.5 ns
Extra memory to make accessing it faster	2MB	7 ns
Working memory, what your program need	8GB	100 ns
Large, longtime storage	1TB	8,000,000 ns

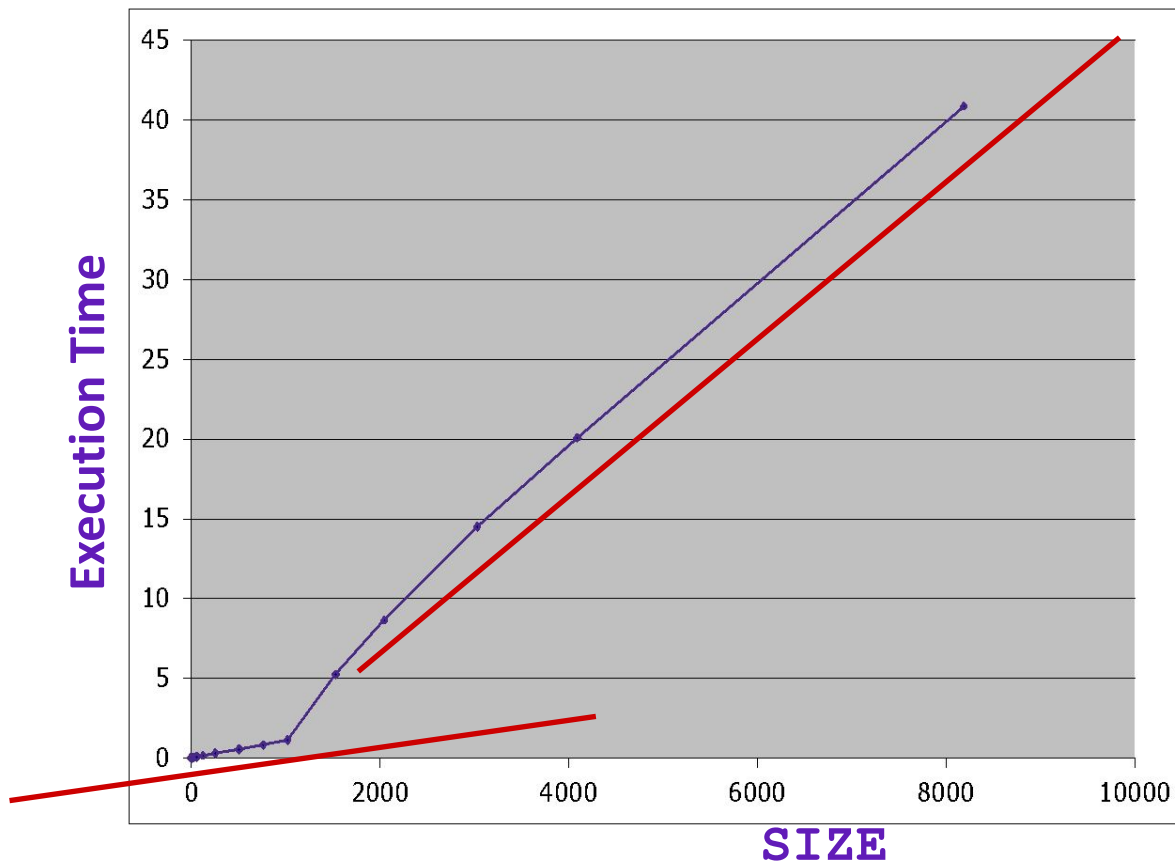
How does execution time grow with SIZE?

```
int array[SIZE];  
// initialize array somewhere else  
int sum = 0;  
for (int i = 0; i < 200000; i++)  
    for (int j = 0; j < SIZE; j++)  
        sum += array[j];
```

Plot:



Actual Data



Incorrect Assumptions

Accessing memory is always quick and constant-time operation



- 👉 Sometimes accessing memory is cheaper and easier than at other times
- 👉 Sometimes accessing memory is very slow

Aside: Moore's Law

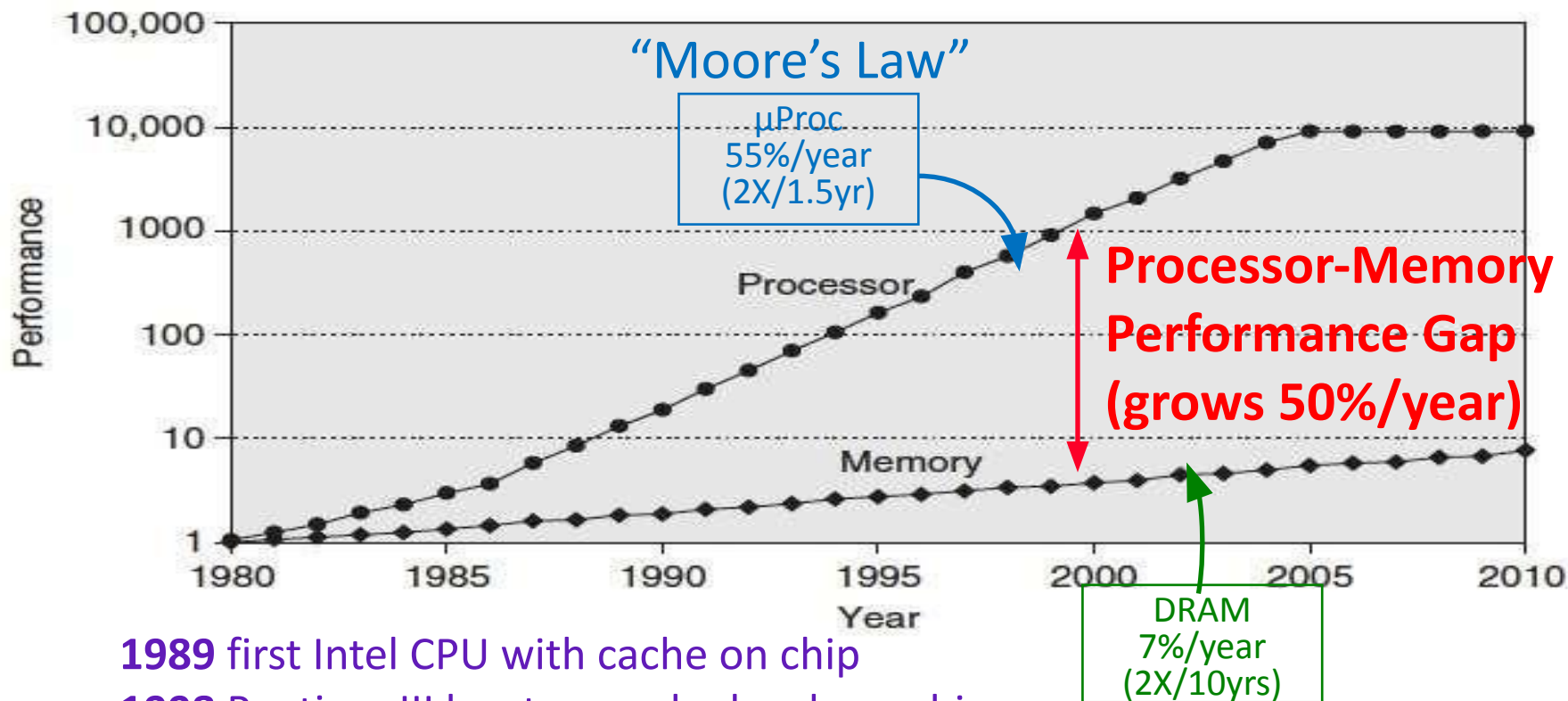
The claim that the number of transistors in an integrated circuit doubles approximately every two years.

- Had a huge impact on computing innovation and advancement.
- Was held true for decades (i.e. 1965-2010)

Nvidia CEO Jensen Huang claimed that Moore's law is dead in 2022.

- There is a physical limit for the amount of space on the device.
- Not everyone agrees (e.g. Intel's CEO, Pat Gelsinger)

Processor-Memory Gap



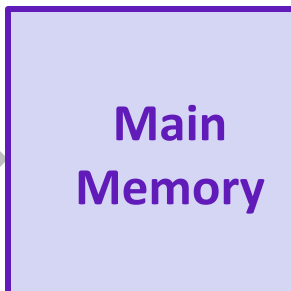
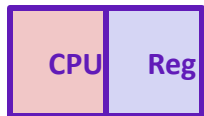
1989 first Intel CPU with cache on chip

1998 Pentium III has two cache levels on chip

Problem: Processor-Memory Bottleneck

Processor performance
doubled about
every 18 months

Bandwidth evolved much slower



Core 2 Duo:
Can process at least
256 Bytes/cycle



Core 2 Duo:
Bandwidth
2 Bytes/cycle
Latency
100-200 cycles (30-60ns)



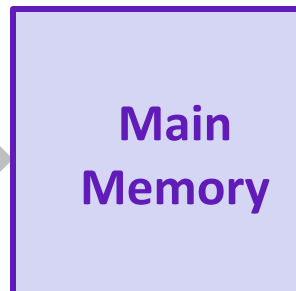
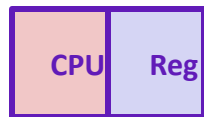
Problem: lots of waiting on memory

cycle: single machine step (fixed-time)

Problem: Processor-Memory Bottleneck

Processor performance
doubled about
every 18 months

Bandwidth evolved much slower



Core 2 Duo:
Can process at least
256 Bytes/cycle



Core 2 Duo:
Bandwidth
2 Bytes/cycle
Latency
100-200 cycles (30-60ns)



Solution: caches

cycle: single machine step (fixed-time)



Cache

Pronunciation: “cash”

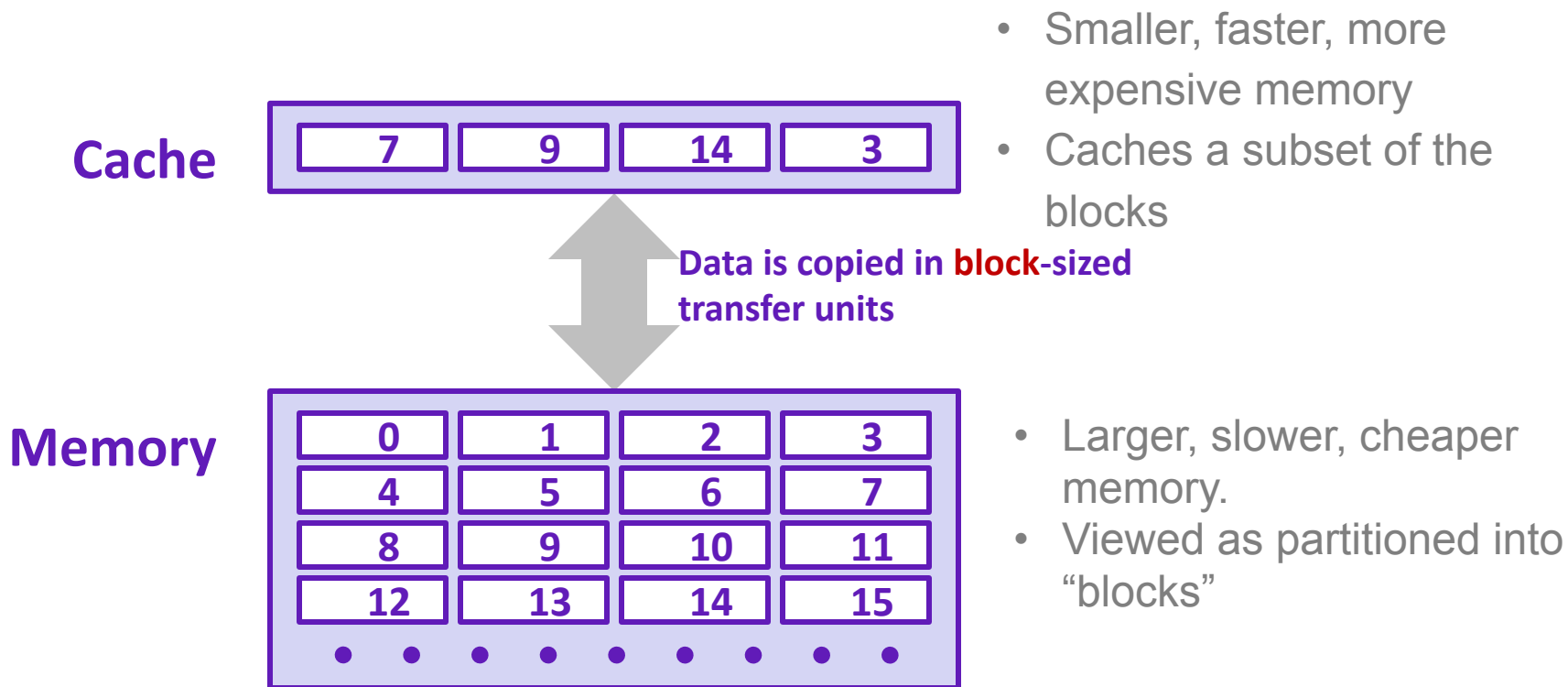
English: A hidden storage space

- for provisions, weapons, and/or treasures

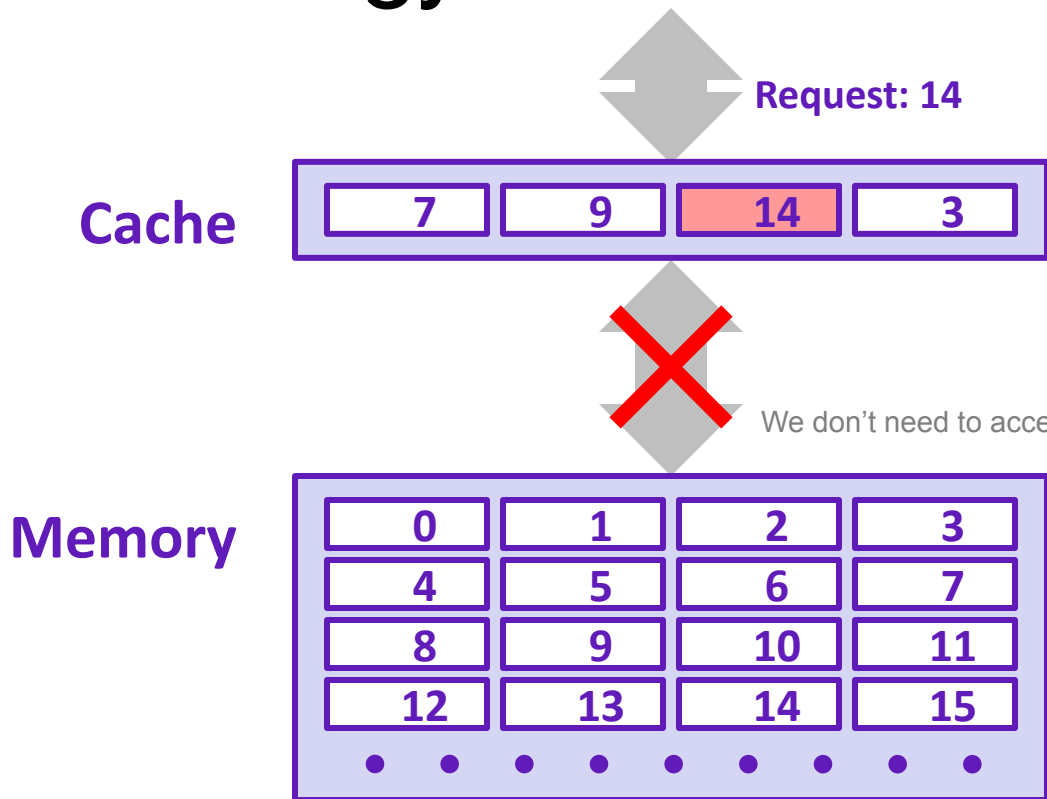
Computer: Memory with short access time used for the storage of frequently or recently used data

- *More generally:* Used to optimize data transfers between any system elements with different characteristics (network interface cache, I/O cache, etc.)

General Cache Mechanics



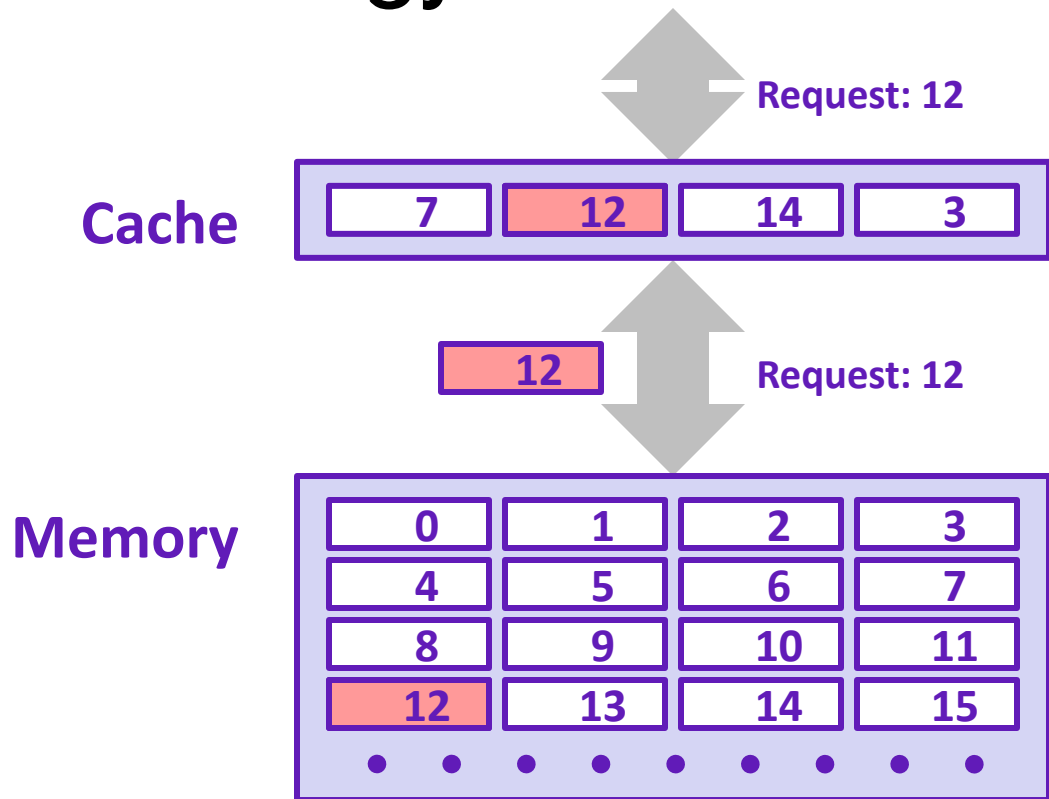
Terminology: **Cache Hit**



Data in block 14 is
needed
Block 14 is in
cache:
Hit!

Data is returned to CPU

Terminology: **Cache Miss**



Data in block 12 is needed
Block 12 is not in cache:
Miss!

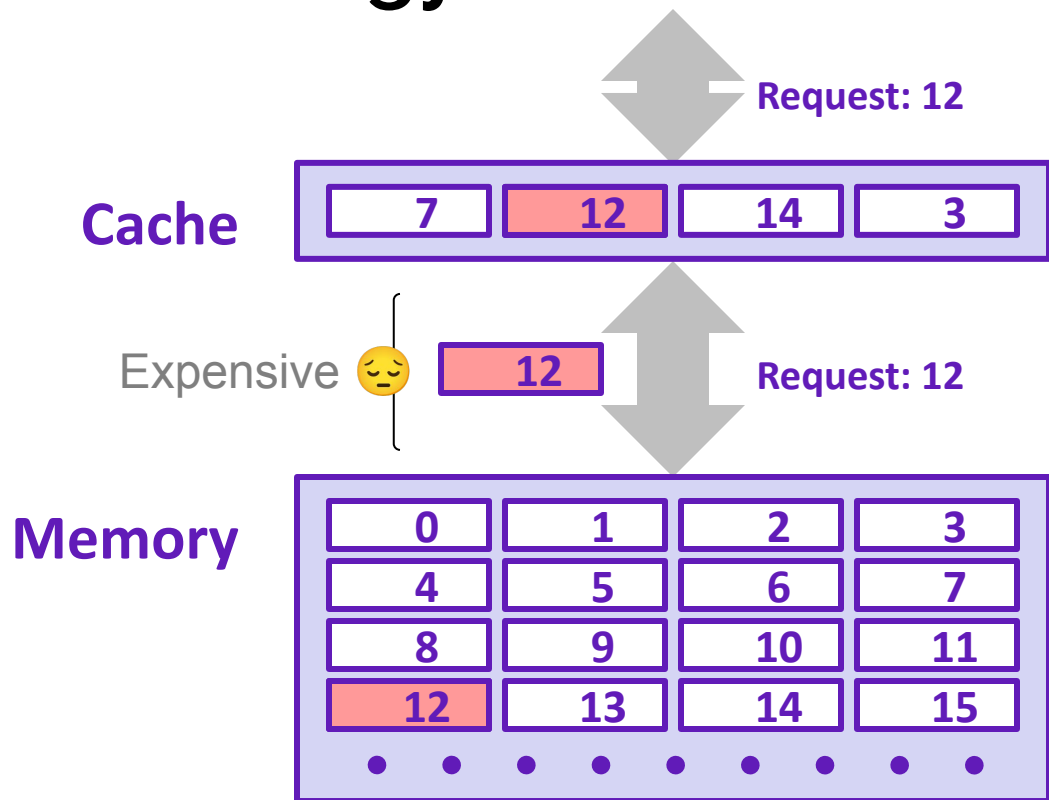
Block b is fetched from memory

Block b is stored in cache

- **Placement policy:**
determines where b goes
- **Replacement policy:**
determines which block gets evicted (victim)

Data is returned to CPU

Terminology: **Cache Miss**



Data in block 12 is needed
Block 12 is not in cache:
Miss!

Block b is fetched from memory

Block b is stored in cache

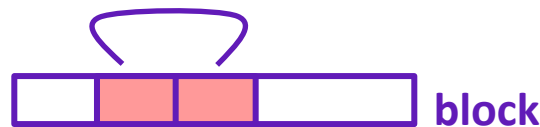
- **Placement policy:**
determines where b goes
- **Replacement policy:**
determines which block gets evicted (victim)

Data is returned to CPU

Why Caches Work

Locality: Programs tend to use data and instructions with addresses near or equal to those they have used recently

- *Temporal locality:*
 - Recently referenced items are *likely* to be referenced again in the near future
- *Spatial locality:*
 - Items with nearby addresses *tend* to be referenced close together in time



Analogy: You are eating a sandwich cut into two halves. You will probably take bites out of one half first before eating the other half.

How do caches take advantage of this?

How do caches take advantage of locality?

Whenever you read from memory, an entire “line” of cache is filled

- The cache grabs more than just the memory you asked for: it grabs surrounding bytes too!
- This takes advantage of spatial locality

Caches fill up quickly, but they try to keep the most recently accessed cache lines

- Takes advantage of temporal locality
- Many different “cache replacement policies” – heuristics to decide which lines to swap out when cache is full

Example: Any Locality?

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

- Temporal: `sum` referenced in each iteration
- Spatial: consecutive elements of array `a[]` accessed

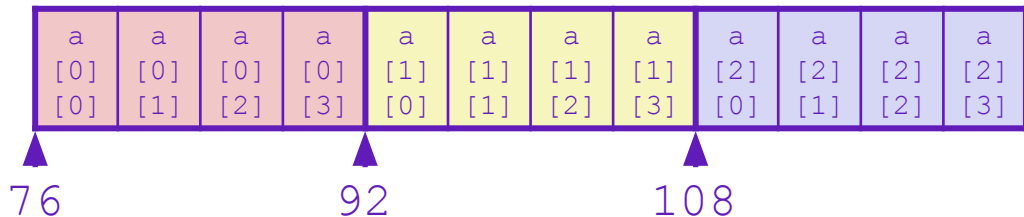
Locality Example #1

```
int sum_array_rows(int a[M][N]) {  
    int i, j, sum = 0;  
    for (i = 0; i < M; i++)  
        for (j = 0; j < N; j++)  
            sum += a[i][j];  
    return sum;  
}
```

Locality Example #1

```
int sum_array_rows(int a[M][N]) {  
    int i, j, sum = 0;  
    for (i = 0; i < M; i++)  
        for (j = 0; j < N; j++)  
            sum += a[i][j];  
    return sum;  
}
```

Layout in Memory



Note: 76 is just one possible starting address of array a

M = 3, N = 4

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

Access Pattern:
stride = 1

1)	a[0][0]
2)	a[0][1]
3)	a[0][2]
4)	a[0][3]
5)	a[1][0]
6)	a[1][1]
7)	a[1][2]
8)	a[1][3]
9)	a[2][0]
10)	a[2][1]
11)	a[2][2]
12)	a[2][3]

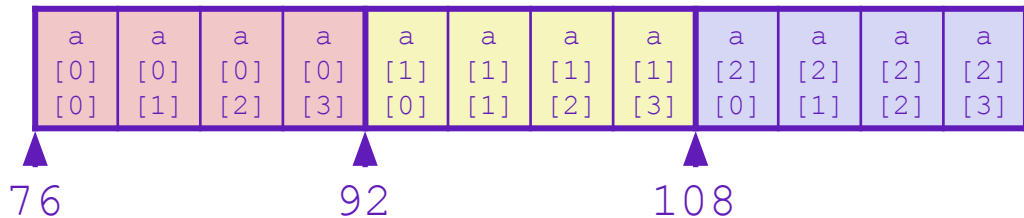
Locality Example #2

```
int sum_array_cols(int a[M][N]) {  
    int i, j, sum = 0;  
    for (j = 0; j < N; j++)  
        for (i = 0; i < M; i++)  
            sum += a[i][j];  
    return sum;  
}
```

Locality Example #2

```
int sum_array_cols(int a[M][N]) {  
    int i, j, sum = 0;  
    for (j = 0; j < N; j++)  
        for (i = 0; i < M; i++)  
            sum += a[i][j];  
    return sum;  
}
```

Layout in Memory



M = 3, N = 4

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

Access Pattern:
stride = 3

1)	a[0][0]
2)	a[1][0]
3)	a[2][0]
4)	a[0][1]
5)	a[1][1]
6)	a[2][1]
7)	a[0][2]
8)	a[1][2]
9)	a[2][2]
10)	a[0][3]
11)	a[1][3]
12)	a[2][3]

Compute Points on a Line (Original)

```
// Compute  $y = mx + b$  for each point
void compute_line(float m, float b, float* points, int n) {
    for(int i = 0; i < n; i++) {
        points[i] = m * points[i];
    }
    for(int i = 0; i < n; i++) {
        points[i] = b + points[i];
    }
}
```

Compute Points on a Line (Improved)

```
// Compute  $y = mx + b$  for each point
void compute_line(float m, float b, float* points, int n) {
    for(int i = 0; i < n; i++) {
        points[i] = m * points[i];
        points[i] = b + points[i];
    }
}
```

- Better temporal locality!

Polling Time!

Please answer in the [LP16 assignment on Gradescope](#)!

Which of the following statements is correct?

- To *increase* **temporal** locality, put *fewer* memory accesses in your loop body
- To *increase* **spatial** locality, access the *same* memory several times
- As you allocate *more* memory, you will get *more* cache hits
- To *increase* **temporal** locality, access the *same* memory sooner rather than later

Memory Hierarchies

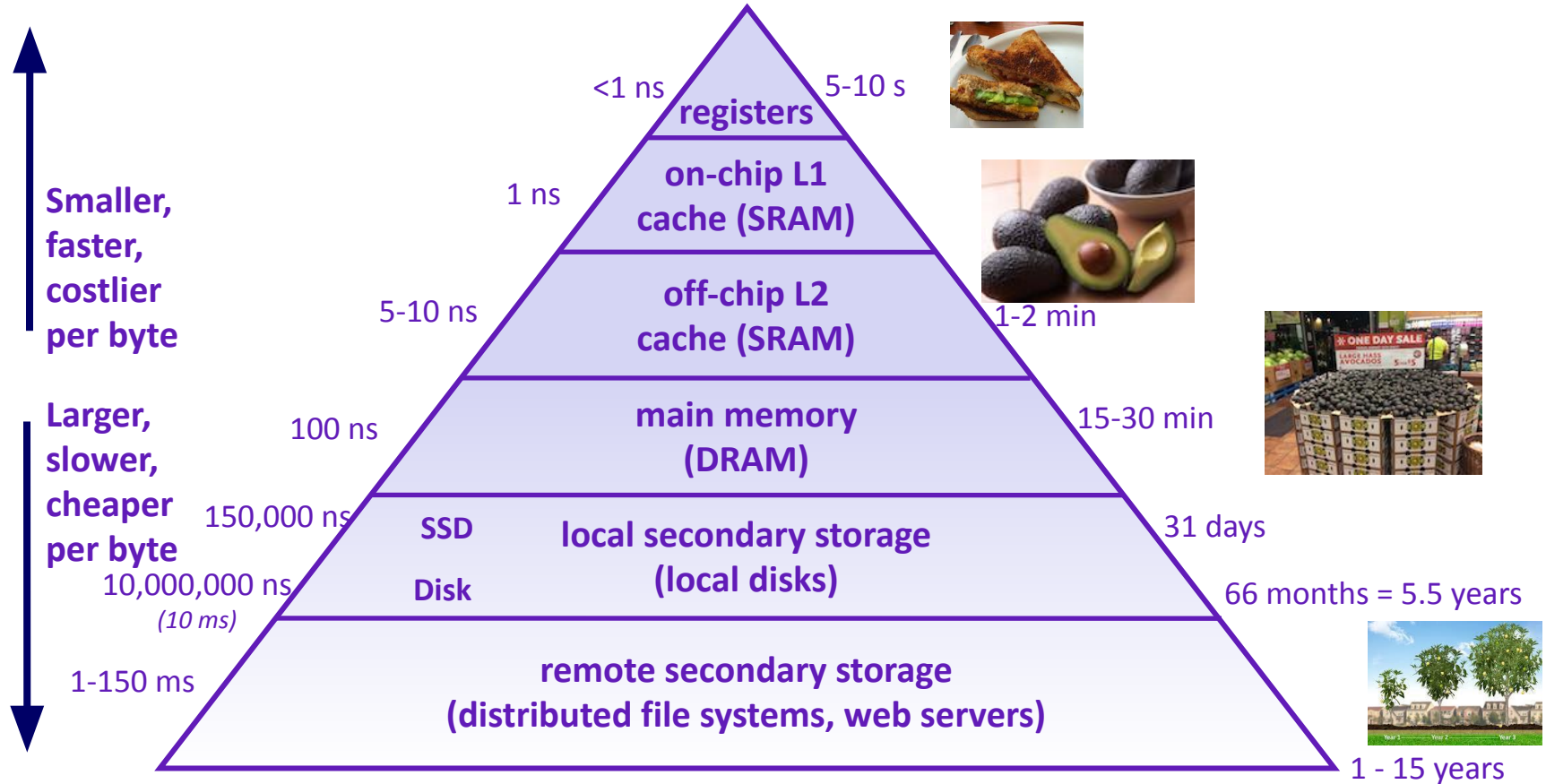
Some fundamental and enduring properties of hardware and software systems:

- **Faster** storage technologies almost always cost more per byte and have lower capacity
- **Well-written** programs tend to exhibit good locality

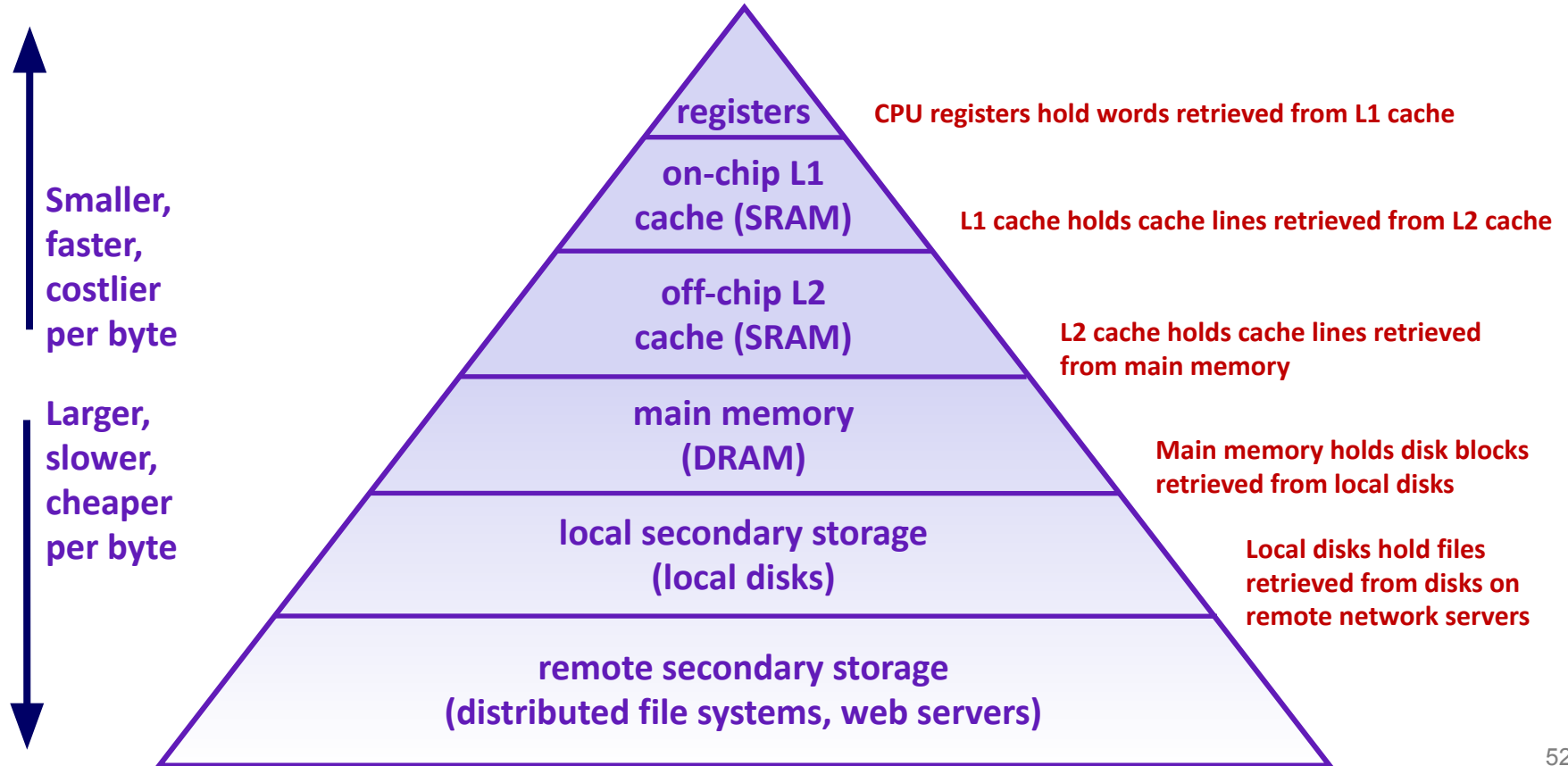
These properties complement each other beautifully

- They suggest an approach for organizing memory and storage systems known as a **memory hierarchy**
 - For each level k , the faster, smaller memory system at level k serves as a cache for the larger, slower memory system at level $k+1$

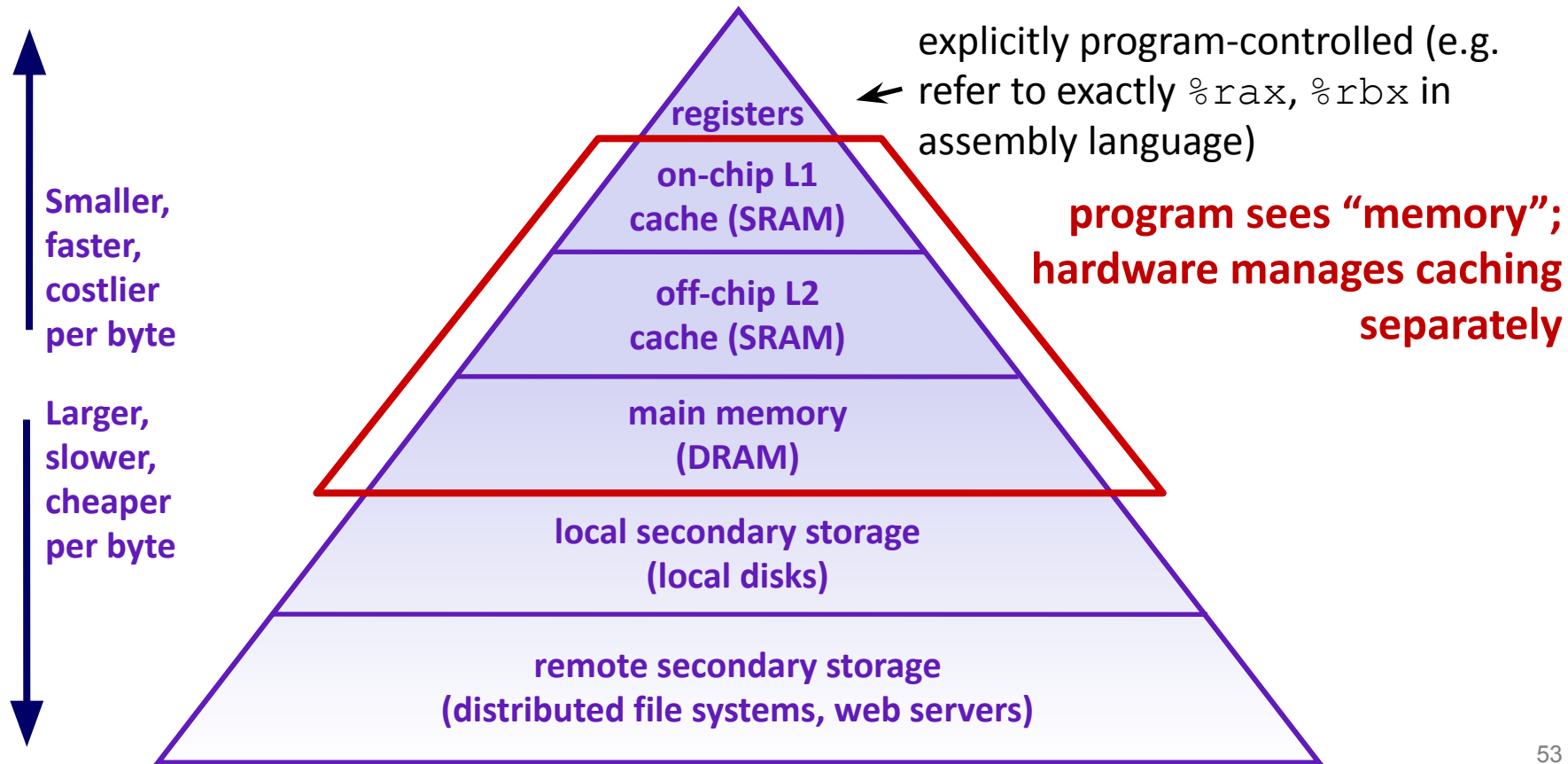
An Example Memory Hierarchy



An Example Memory Hierarchy



Why haven't we seen caches before?



All systems favor “cache-friendly code”

Write code that has locality!

- Spatial: access data contiguously, use small strides
- Temporal: make sure access to the same data is not too far apart in time, keep working set reasonably small

How can you achieve locality?

- Adjust memory accesses in *code* (software) to lower the cache miss rate (MR)
 - Requires knowledge of *both* how caches work as well as your system's parameters
- Proper choice of algorithm
- Loop transformations

Summary

Memory Hierarchy

- Successively higher levels contain “most used” data from lower levels
- Accessing the disk is very slow
 - This is why we *discourage* excess I/O in homework assignments!
- Exploits *temporal and spatial locality*
- Caches are intermediate storage levels used to optimize data transfers between any system elements with different characteristics

Cache Performance

- Ideal case: found in cache (hit)
- Bad case: not found in cache (miss), search in next level

Assignment Reminders

EX14 due **Monday before class**

HW4: Testing **due Monday @11:59pm**

- Write tests based on the provided interface
- Just writing tests, *not* the implementation! Stay tuned for HW5 :)

Make sure to submit LP16 before the next class as well.

Look forward to starting **C++** programming next lecture! 🙄🎉