What do you think?

Discuss!

What considerations should be made while choosing numerical data types?

What are the advantages of narrower types?

What are the advantages of wider types?

CSE 374: Lecture 19

Memory Management

Allocating array memory

An array IS a pointer

A String is an array of char, terminating with \0

strsize returns length of string, minus the final \0 character

Allocate enough space for (strsize+1) chars

```
// copy original string

int strsize = strlen(s)+1;
// result = (char *)malloc(strsize);
result =(char*)malloc(strsize*sizeof(char));
printf ("sizeof char: %d \n", sizeof(char));
strncpy(result, s, strsize);

// from final_reverse.c, lect. 11
```

Using pointer manipulation for good

```
void fill_mem (void* ptr, size) {

  uintptr_t memadd = (uintptr_t) ptr;
  for (int i=0; i<16 && i <size; i++) {

    *((unsigned char*)(memadd+1)) = 0xFE;

  }
}
```

Buffer Overflow

What is buffer overflow?

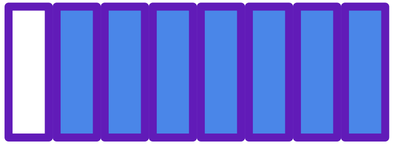
‘Gets’ doesn’t check for buffer size; if the string is more than 8 characters, it will write onto the memory at the end of buf.

Why is that so bad? (see the stack!)

```
void echo() {
        char buf[8];
        gets(buf);
        puts(buf);
}
```

The stack

<- stack



Stack stores active functions & <u>local</u> variables

Each function gets a frame, moving down in memory

Last frame is completed, deleted

then the next most recent frame.

(Last in-first out)

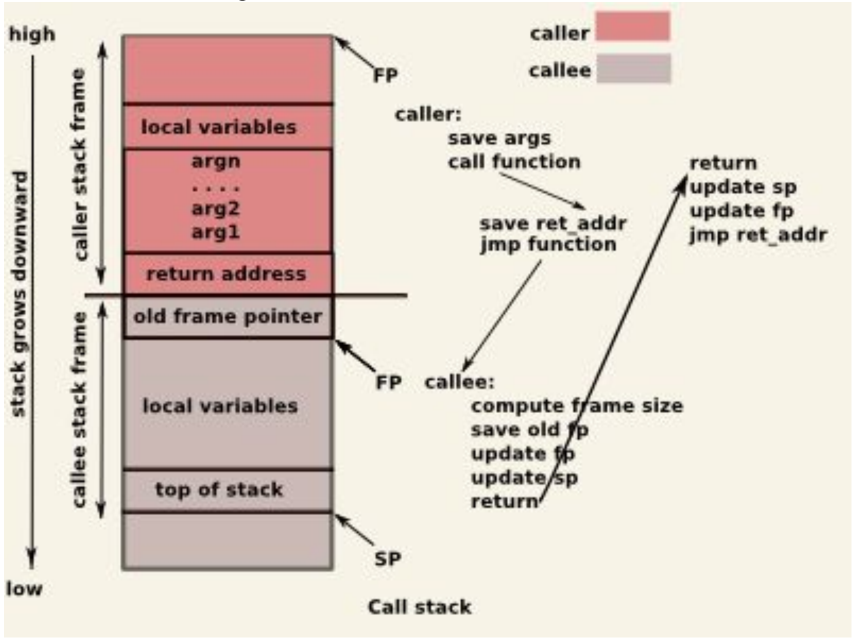
Each function call creates a frame

Containing:

Arguments, return address,

Pointer-to-last-frame,

local variables



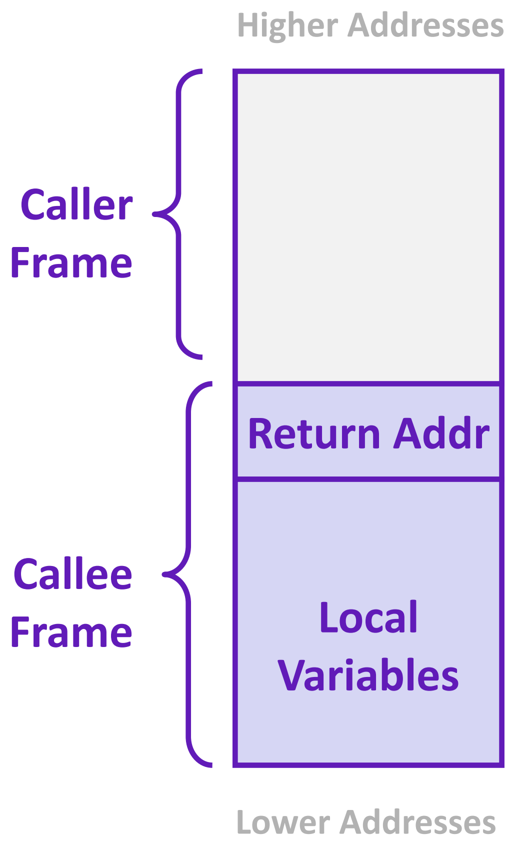
Linux Stack Frame

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 is also applicable to 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

Thanks to A. McKinney

Example

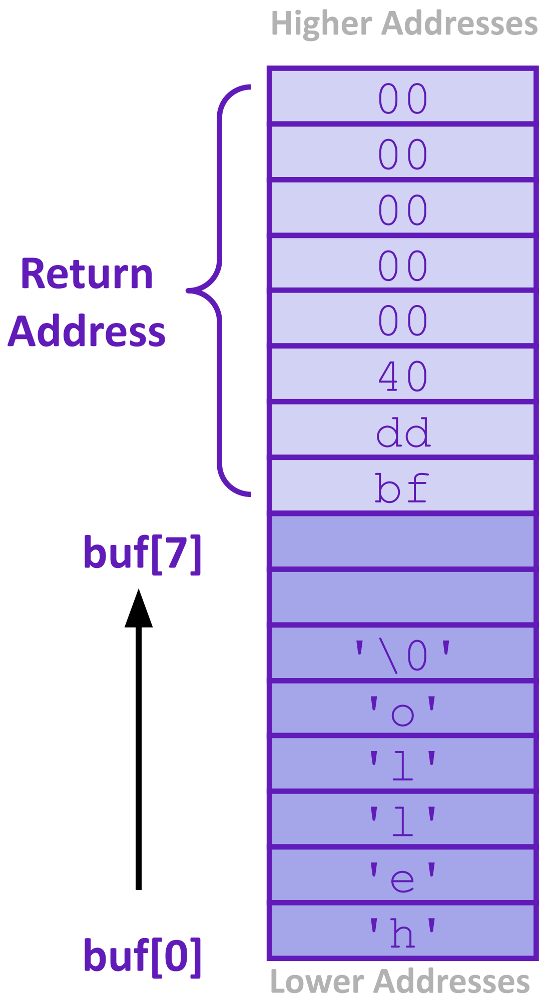
Stack grows *down* towards lower addresses

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! ☺️



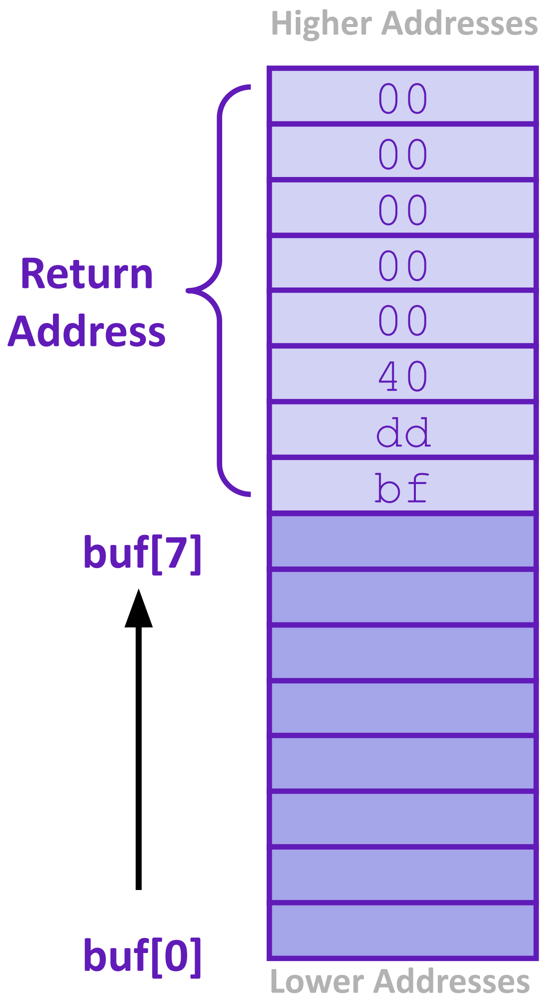
Example

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: helloabcdef
```



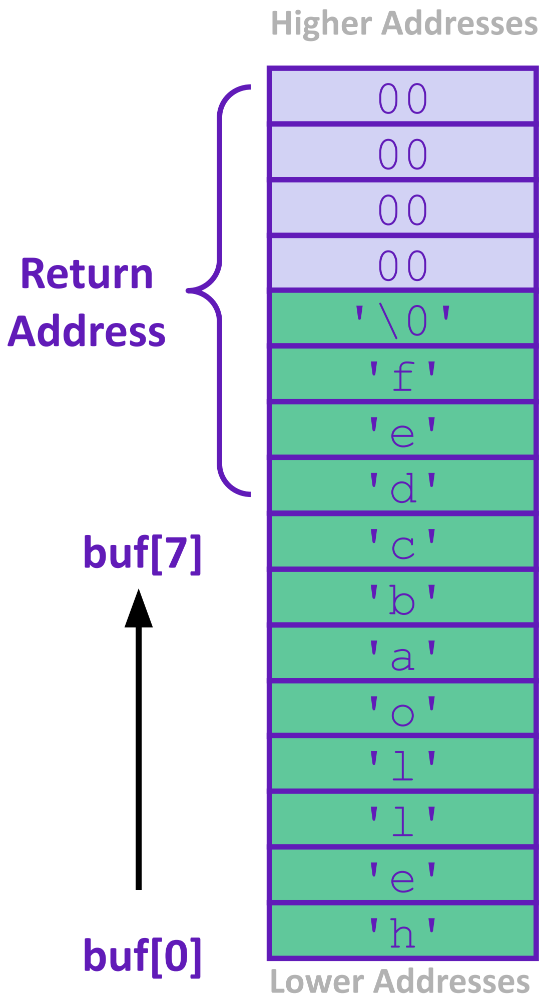
Example

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

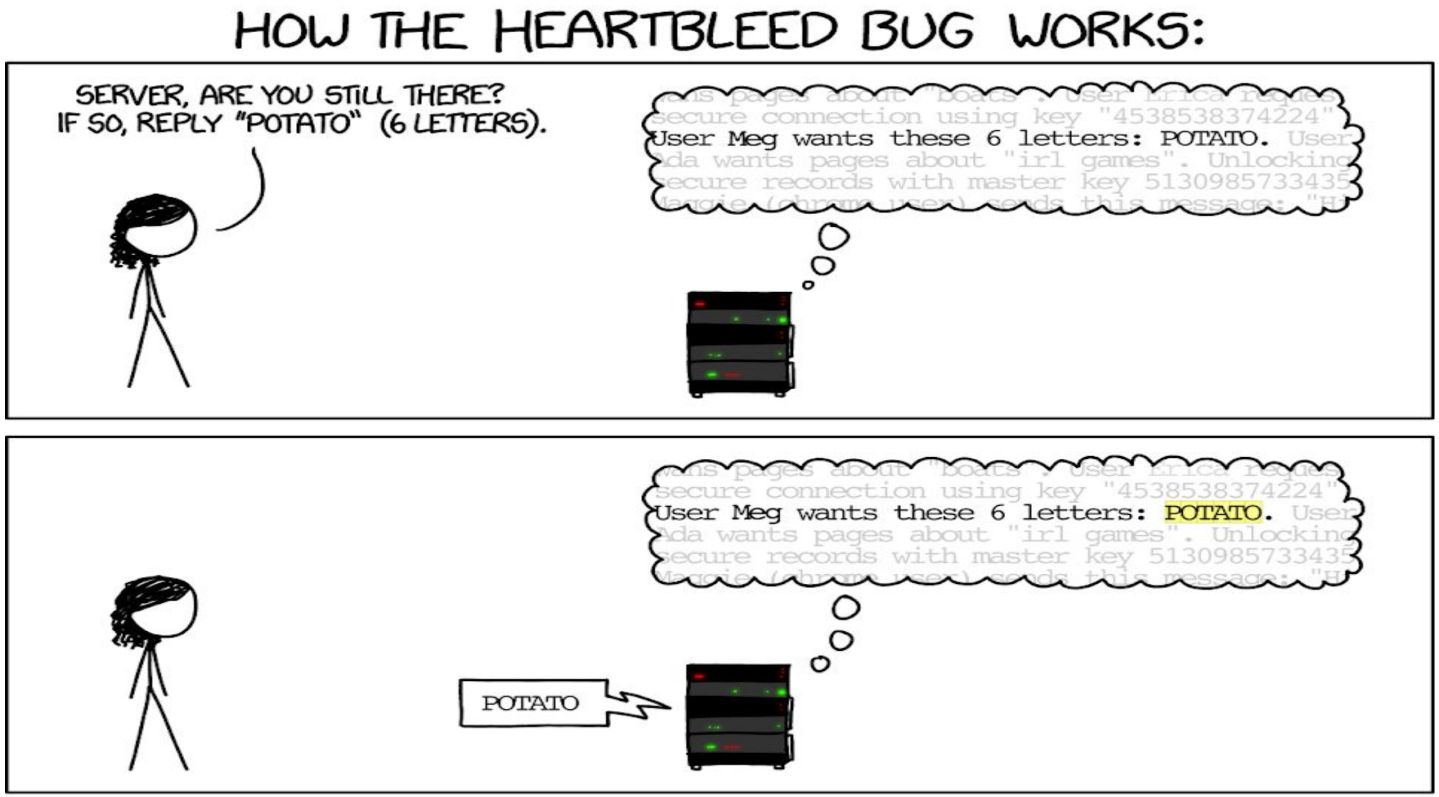
Exploits Based on Buffer Overflows

Examples across the decades

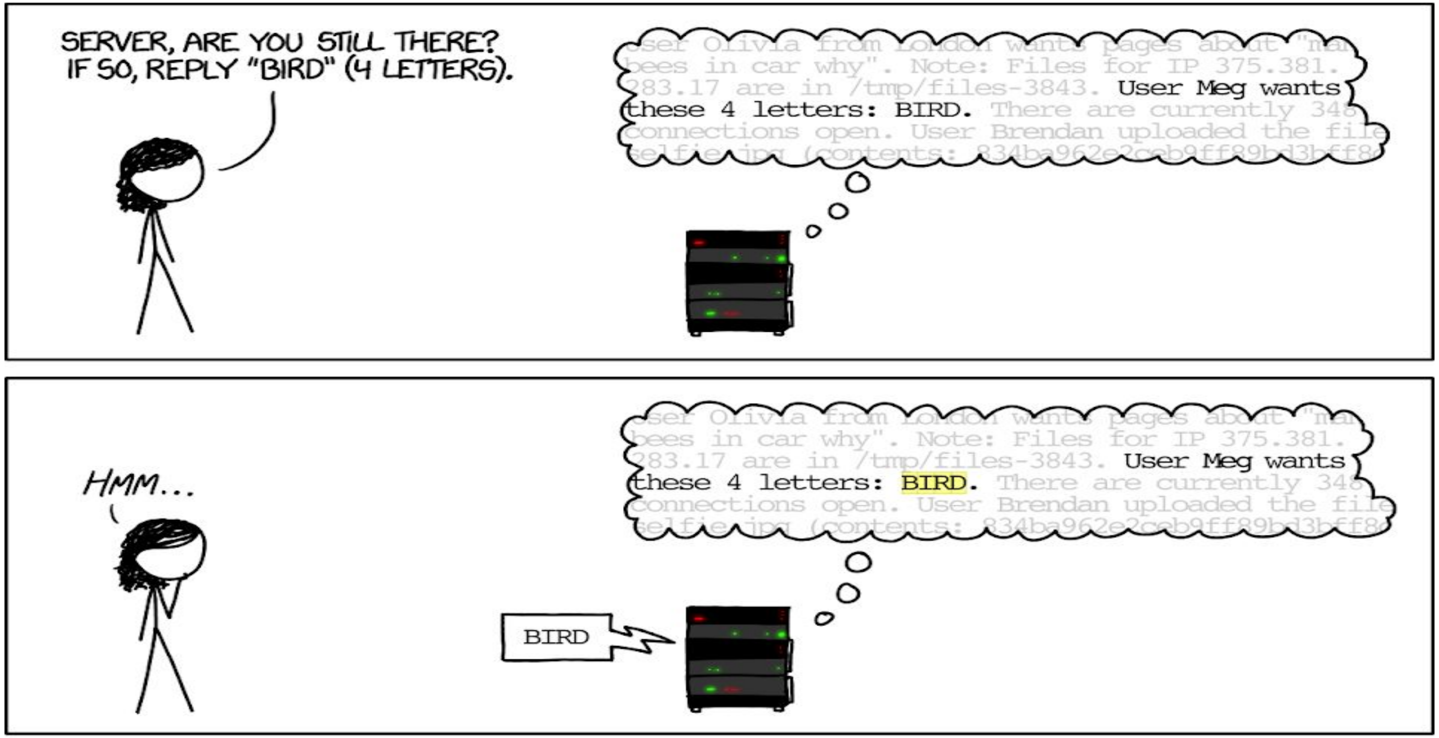
- Original “Internet worm” (aka Morris Worm) (1988)
  - Later became one of the founders of YCombinator.
- Heartbleed (2014, affected 17% of servers)
  - Similar issue in Cloudbleed (2017)
- Hacking embedded devices
  - Cars, Smart homes, Planes

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

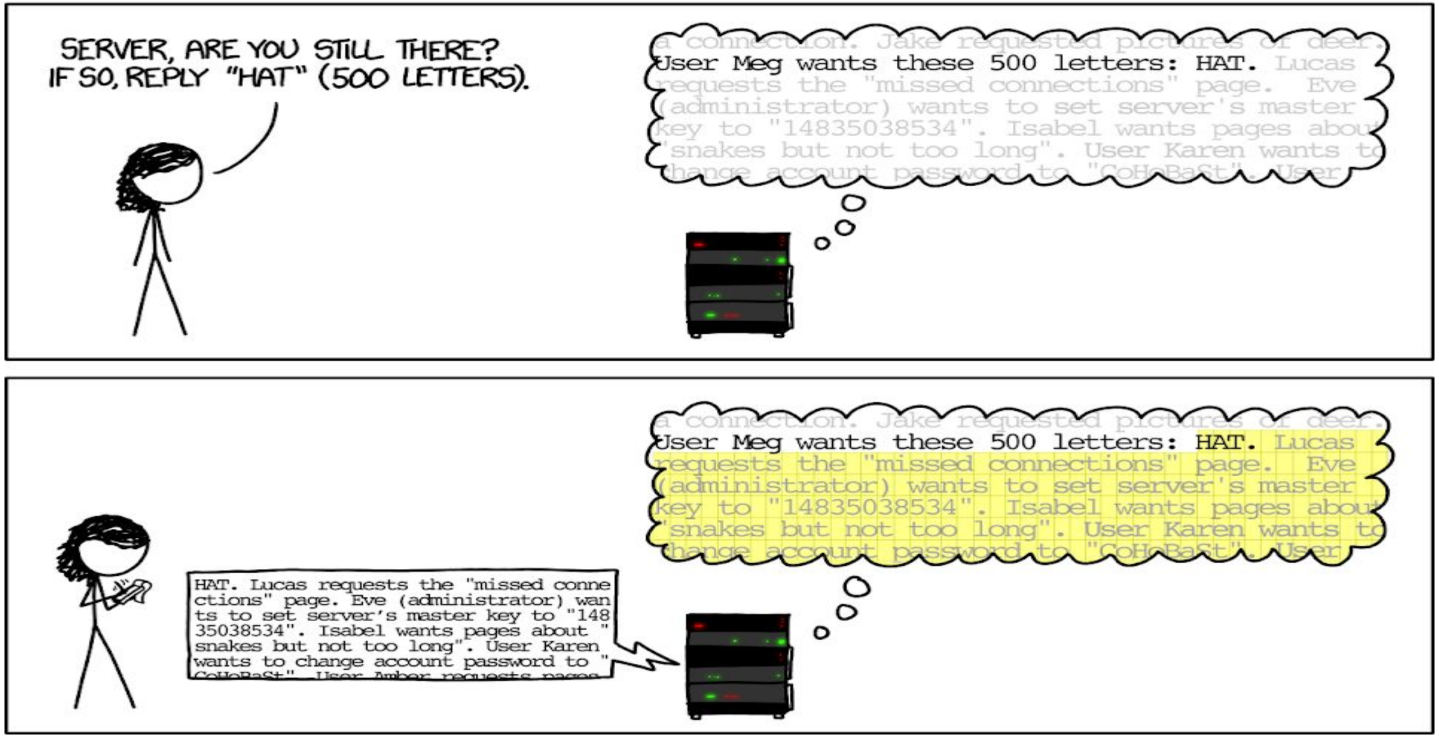
Example: Heartbleed


HOW THE HEARTBLEED BUG WORKS:
SERVER, ARE YOU STILL THERE? IF SO, REPLY "POTATO" (6 LETTERS).
wants pages about "boats". User Erica reques secure connection using key "4538538374224" User Meg wants these 6 letters: POTATO. User Ada wants pages about "irl games". Unlocking secure records with master key 5130985733435 Maggie (chrome user) sends this message: "H
POTATO

Example: Heartbleed


SERVER, ARE YOU STILL THERE? IF SO, REPLY "BIRD" (4 LETTERS).
User Olivia from London wants pages about "ma
bees in car why". Note: Files for IP 375.381.
283.17 are in /tmp/files-3843. User Meg wants
these 4 letters: BIRD. There are currently 348
connections open. User Brendan uploaded the file
selfie.jpg (contents: 834ba962e2ceb9ff89bd3bff8
HMM...
User Olivia from London wants pages about "ma
bees in car why". Note: Files for IP 375.381.
283.17 are in /tmp/files-3843. User Meg wants
these 4 letters: BIRD. There are currently 348
connections open. User Brendan uploaded the file
selfie.jpg (contents: 834ba962e2ceb9ff89bd3bff8
BIRD

Example: Heartbleed


SERVER, ARE YOU STILL THERE? IF SO, REPLY "HAT" (500 LETTERS).
a connection. Jake requested pictures of deer. User Meg wants these 500 letters: HAT. Lucas requests the "missed connections" page. Eve (administrator) wants to set server's master key to "14835038534". Isabel wants pages about "snakes but not too long". User Karen wants to change account password to "CoHoBaSt". User
HAT. Lucas requests the "missed connections" page. Eve (administrator) wants to set server's master key to "14835038534". Isabel wants pages about "snakes but not too long". User Karen wants to change account password to "CoHoBaSt". User Amber requests pages

Heartbleed (2014)

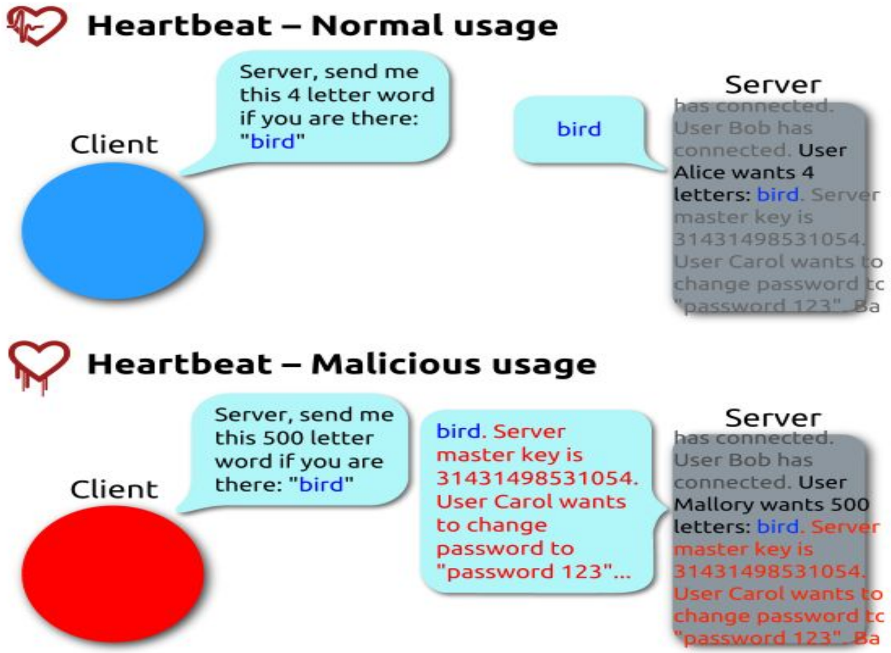
Buffer over-read in OpenSSL

- 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
Client
Server, send me this 4 letter word if you are there: "bird"
bird
Server
has connected. User Bob has connected. User Alice wants 4 letters: bird. Server master key is 31431498531054. User Carol wants to change password to "password 123". Ba
Heartbeat – Malicious usage
Client
Server, send me this 500 letter word if you are there: "bird"
bird. Server master key is 31431498531054. User Carol wants to change password to "password 123"...
Server
has connected. User Bob has connected. User Mallory wants 500 letters: bird. Server master key is 31431498531054. User Carol wants to change password to "password 123". Ba

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

Hacking Cars

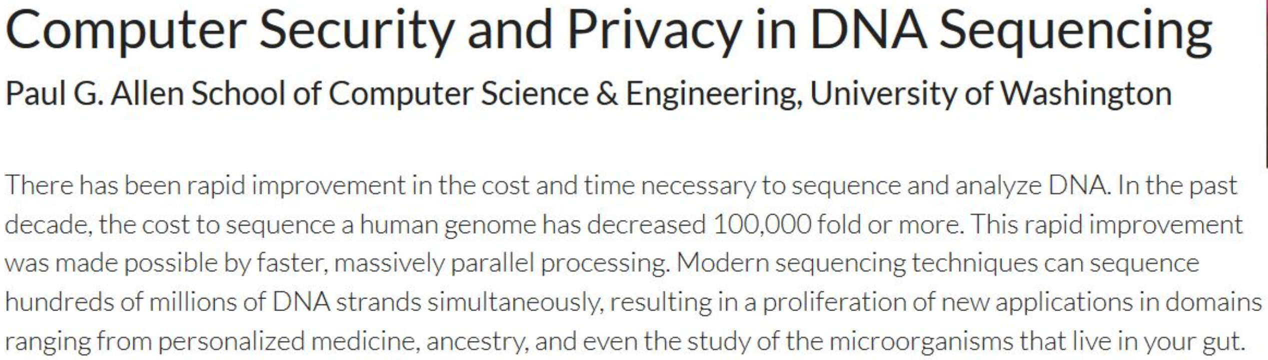
UW CSE research from 2010 demonstrated wirelessly hacking a car using buffer overflow

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.

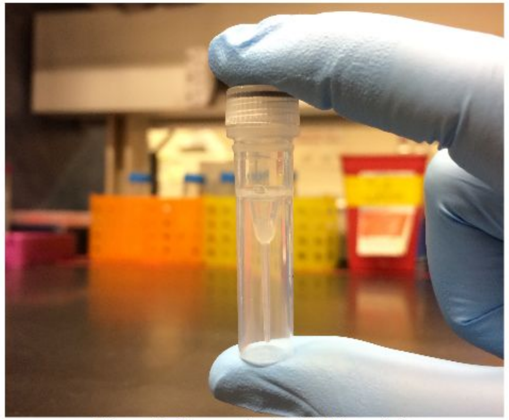


Figure 1: Our synthesized DNA exploit

Change return to last frame

```
void bufferplay (int a, int b, int c) {
  char buffer1[5];
  uintptr_t ret;  // holds an address

  // calculate the return address
  // change to be address of return
  ret = (uintptr_t) buffer1 + 0;

  // treat that number like a pointer,
  // and change the value in it
  *((uintptr_t*)ret) += 0;
}

int main(int argc, char** argv) {
  int x = 0;
  bufferplay (1,2,3);
  x = 1;  // want to skip this line
}
```

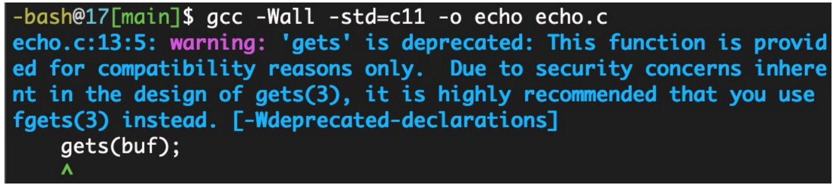
Use GDB:

```
break bufferplay
x buffer1       // prints the location of buffer1
info frame      // Look at "rip" to get the
    // location of the return address
print <rip-location> - <buffer1-location>
    // prints distance from buffer1 to return
    //  address.

disassemble main  // shows the machine
    // code  and how many bytes each
    // instruction takes up.
```

Dealing with Buffer Overflow Attacks

Avoid vulnerabilities in the first place

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



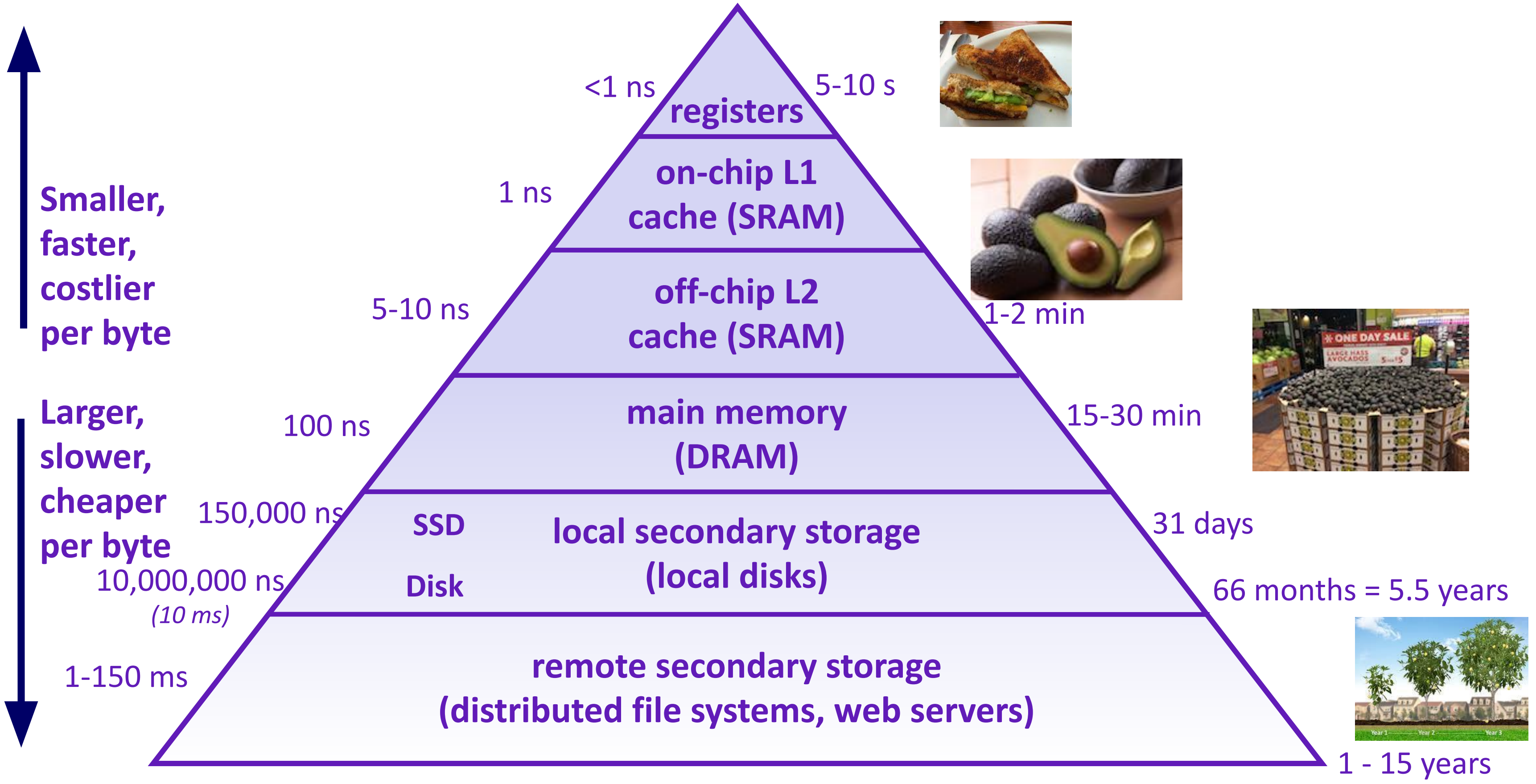
```
-bash@17[main]$ gcc -Wall -std=c11 -o echo echo.c
echo.c:13:5: warning: 'gets' is deprecated: This function is provid
ed for compatibility reasons only.  Due to security concerns inhere
nt in the design of gets(3), it is highly recommended that you use
fgets(3) instead. [-Wdeprecated-declarations]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

System-level protections

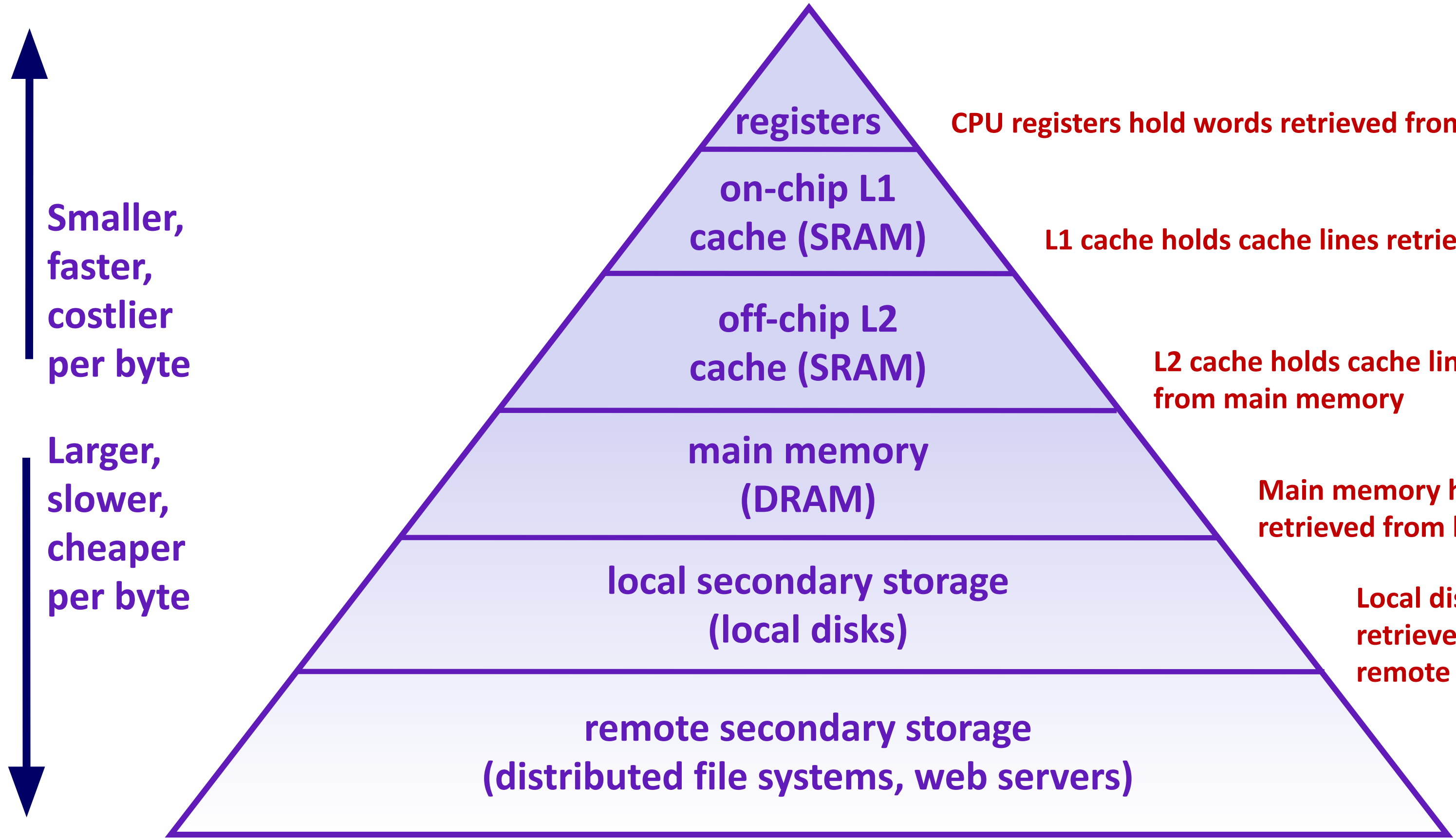
- Make stack non-executable
- Have compiler insert "stack canaries" (just like a networking header checksum)
- Put a special value between buffer and return address
- Check for corruption before leaving function
- Randomized Stack offsets

Using Memory Wisely

An Example Memory Hierarchy



An Example Memory Hierarchy



CPU registers hold words retrieved from L1 cache

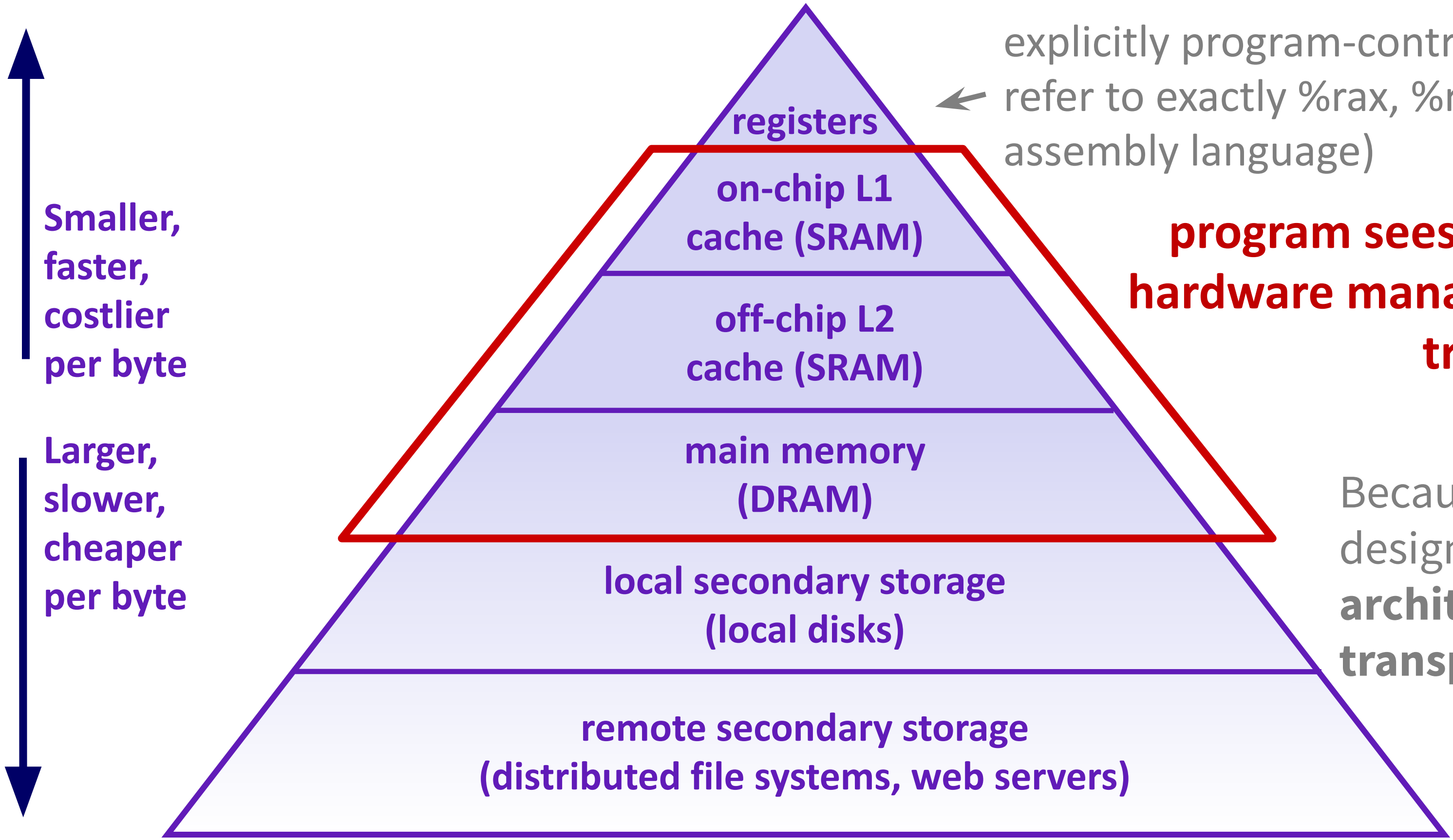
L1 cache holds cache lines retrieved from L2 cache

L2 cache holds cache lines retrieved from main memory

Main memory holds disk blocks retrieved from local disks

Local disks hold files retrieved from disks on remote network servers

Why haven't we seen caches before?



Smaller, faster, costlier per byte
Larger, slower, cheaper per byte
registers
on-chip L1 cache (SRAM)
off-chip L2 cache (SRAM)
main memory (DRAM)
local secondary storage (local disks)
remote secondary storage (distributed file systems, web servers)

explicitly program-controlled (e.g. refer to exactly %rax, %rbx in assembly language)

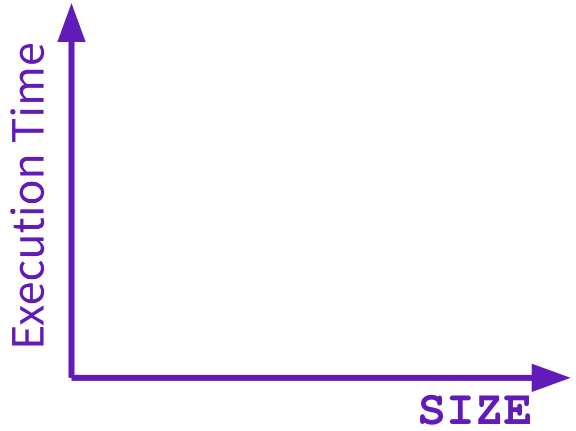
program sees "memory"; hardware manages caching transparently

Because they're designed to be **architecturally transparent**!

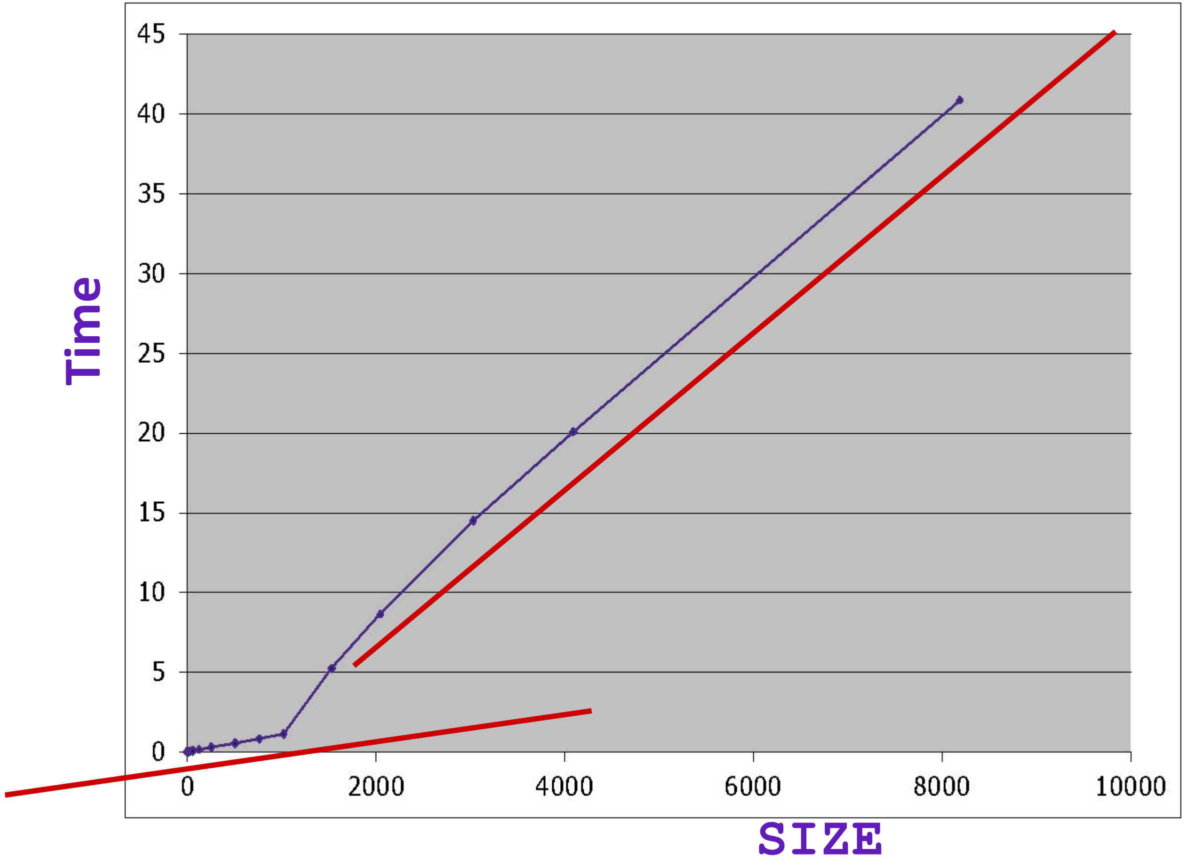
How does execution time grow with SIZE?

```
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

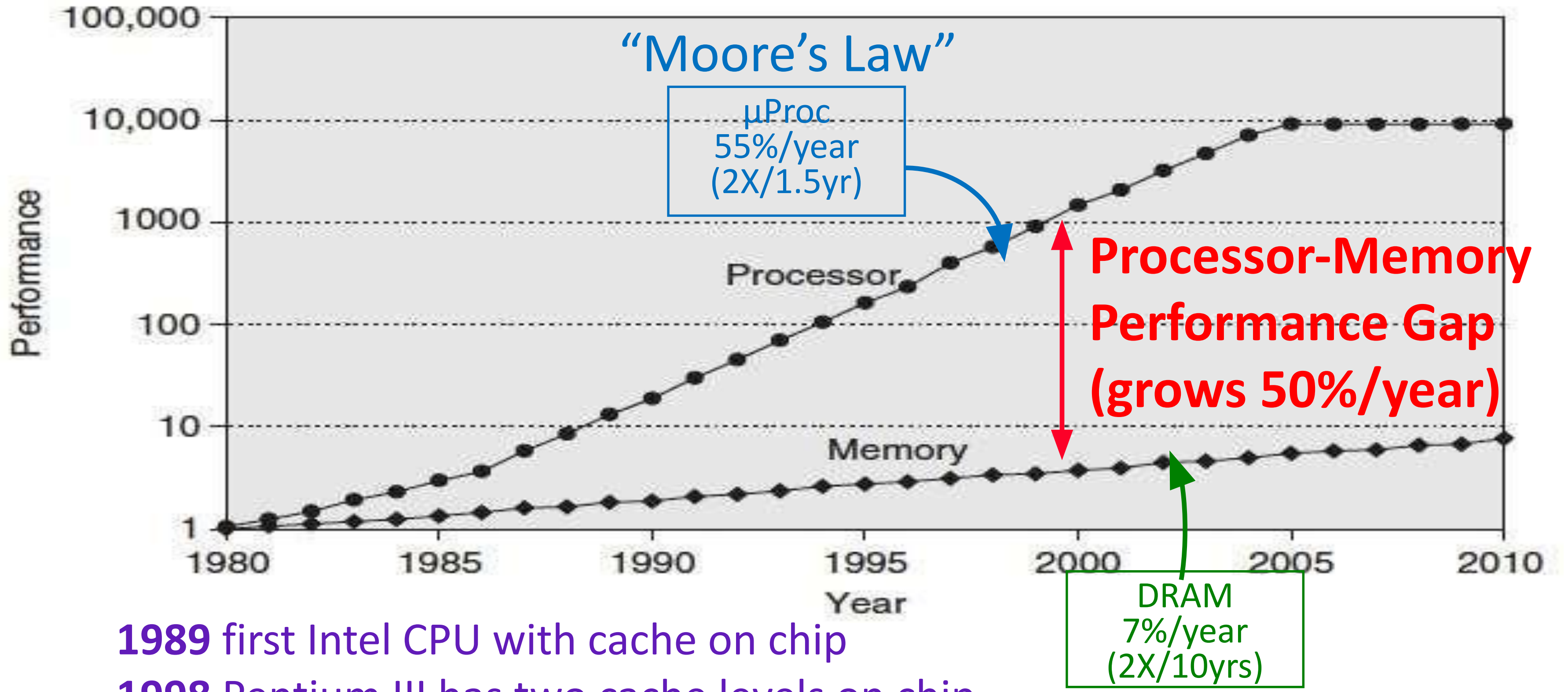
Accessing memory is a quick and constant-time operation

Lies!

Sometimes accessing memory is cheaper and easier than at other times

Sometimes accessing memory is very slow

Processor-Memory Gap



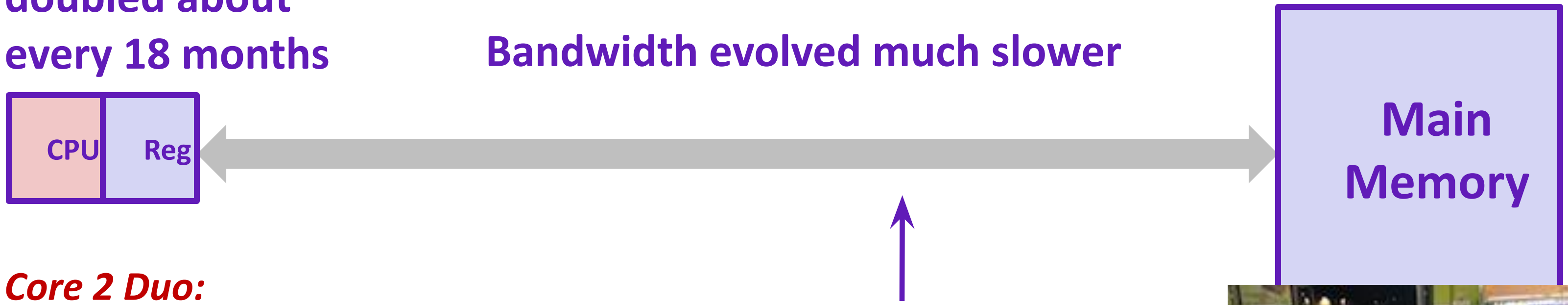
1989 first Intel CPU with cache on chip

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)

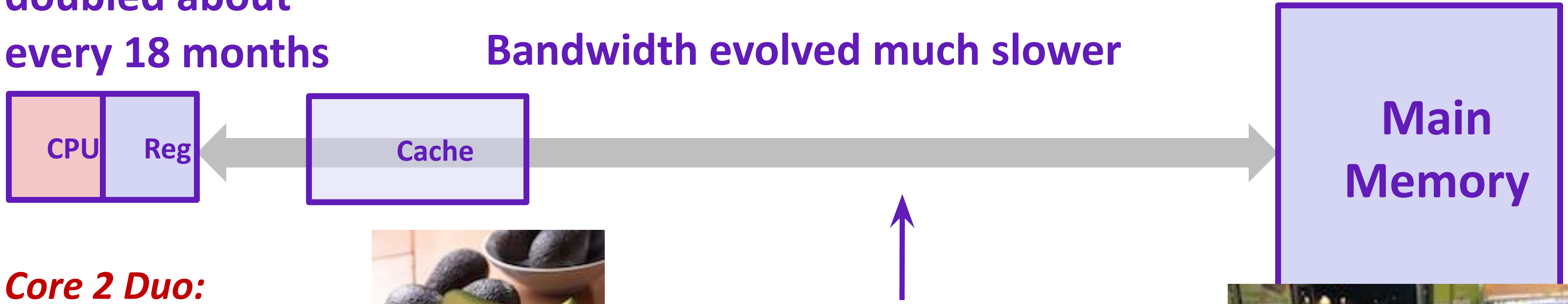
Problem: lots of waiting on memory

***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”

- We abbreviate this as “$”

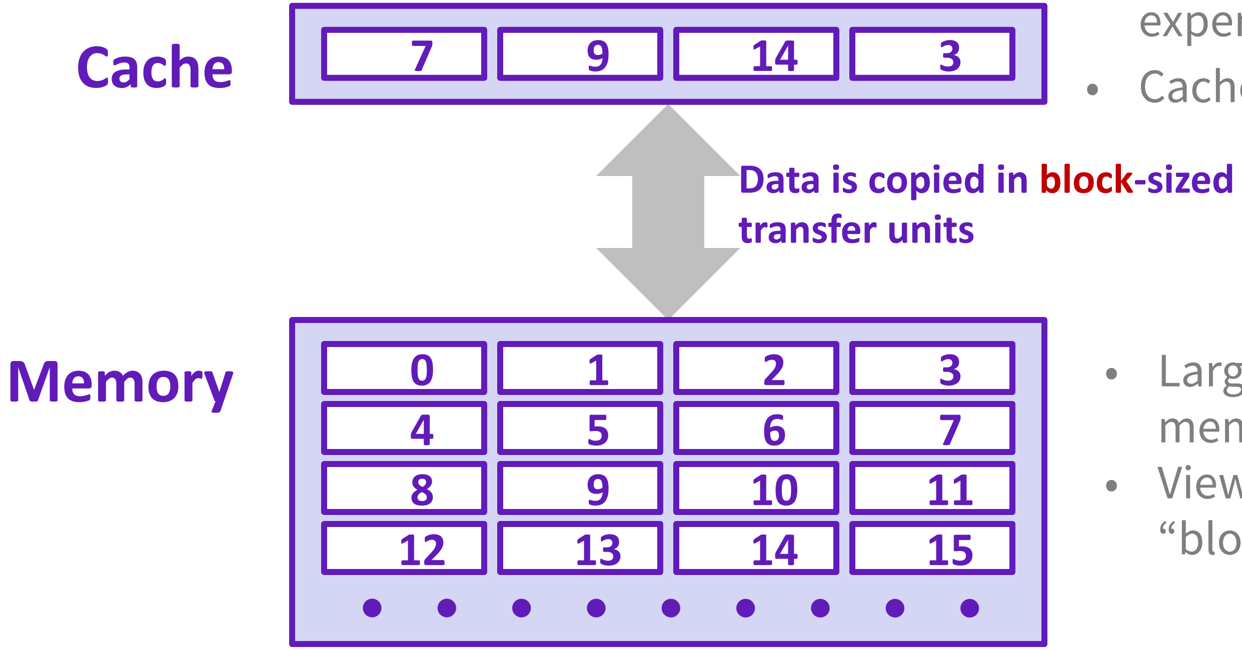
English: A hidden storage space

- 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.)

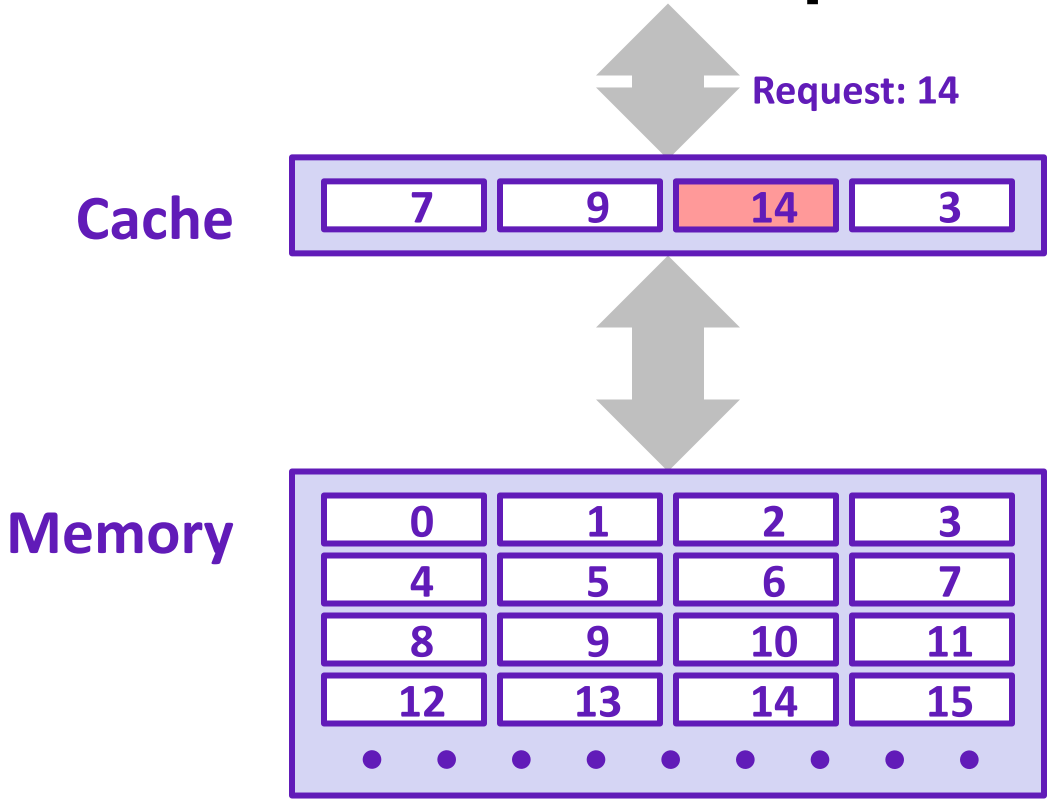
General Cache Mechanics



- Smaller, faster, more expensive memory
- Caches a subset of the blocks

- Larger, slower, cheaper memory.
- Viewed as partitioned into "blocks"

General Cache Concepts: Hit

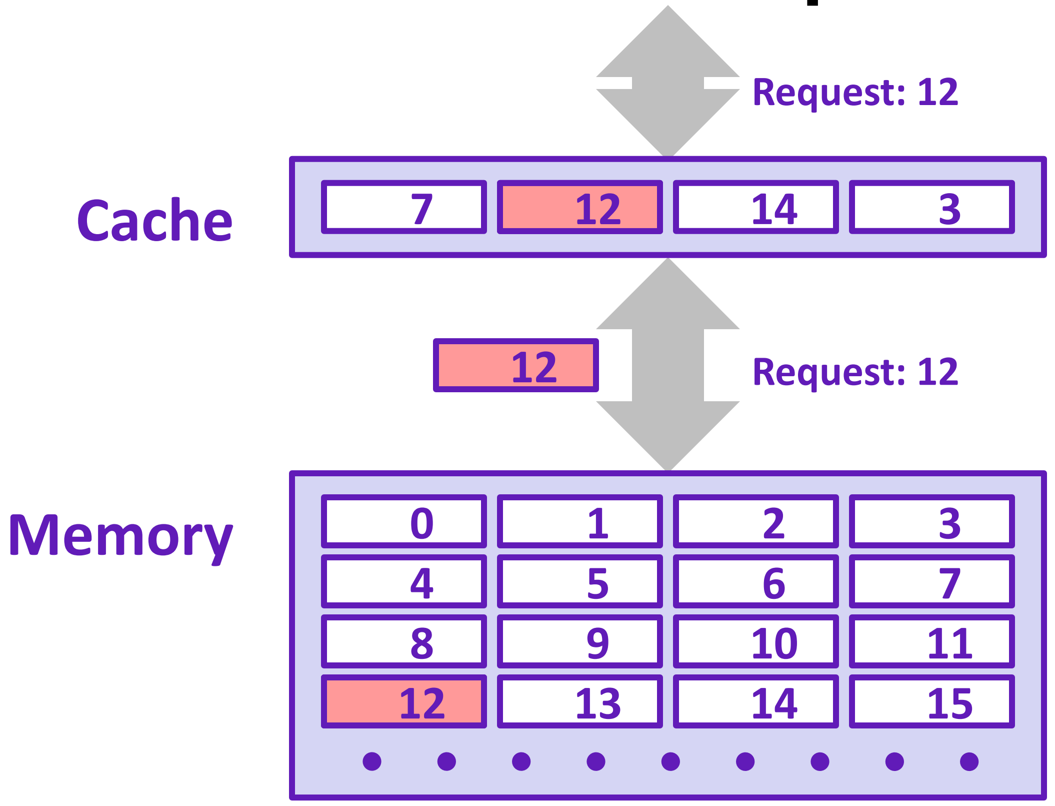


Data in block b is needed

Block b is in cache:
Hit!

Data is returned to CPU

General Cache Concepts: Miss



Data in block b is needed

Block b is not in cache:
Miss!

Block b is fetched from memory

Block b is stored in cache

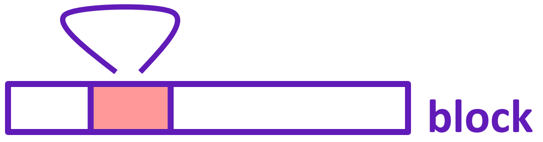
- Placement policy: determines where b goes
- Replacement policy: determines which block gets evicted (victim)

Data is returned to CPU

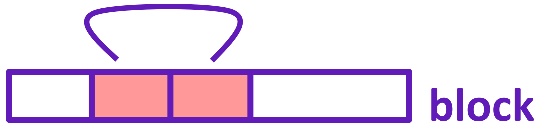
Why Caches Work

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: took a bite of sandwich, probably going to take a bite out of other half of sandwich (as opposed to a new sandwich)

How do caches take advantage of this?

Example: Any Locality?

```
sum = 0;
for (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;
}
```

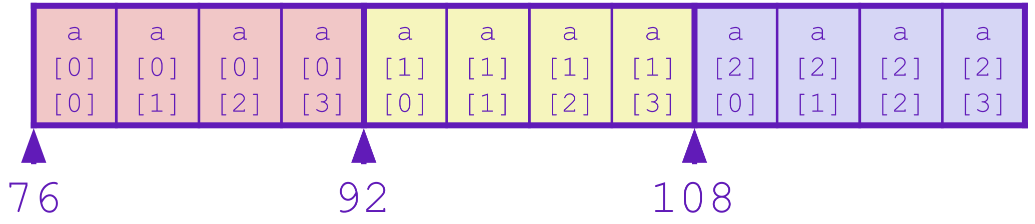
M = 3, N=4

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]
```

Layout in Memory



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

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;
}
```

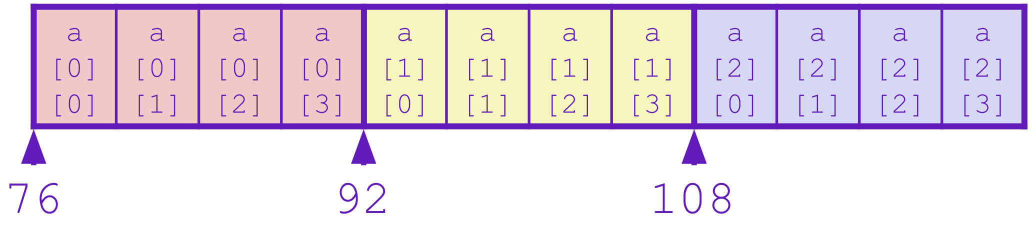
M = 3, N=4

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]

Layout in Memory



Compute Points on a Line (Original)

```c
// 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!

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 improve 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

Memory Allocator

HW Memory

In C: `malloc` and `free` are wrappers to system calls that reserve space in memory, or cancel the reservation.

(System calls deal with memory management, I/O stream management, access files, access the network.)

But `malloc` and `free` are more user friendly than the essential system calls.

Implement equivalents:

```
// acts like 'malloc' and returns address in memory
void* getmem(uintptr_t size)
// acts like 'free' and releases memory
void freemem(void* p)
```

Note:

`Uintptr_t` is an integer type that holds a pointer.

`void*` is a pointer to an unspecified type

HW6: Approach

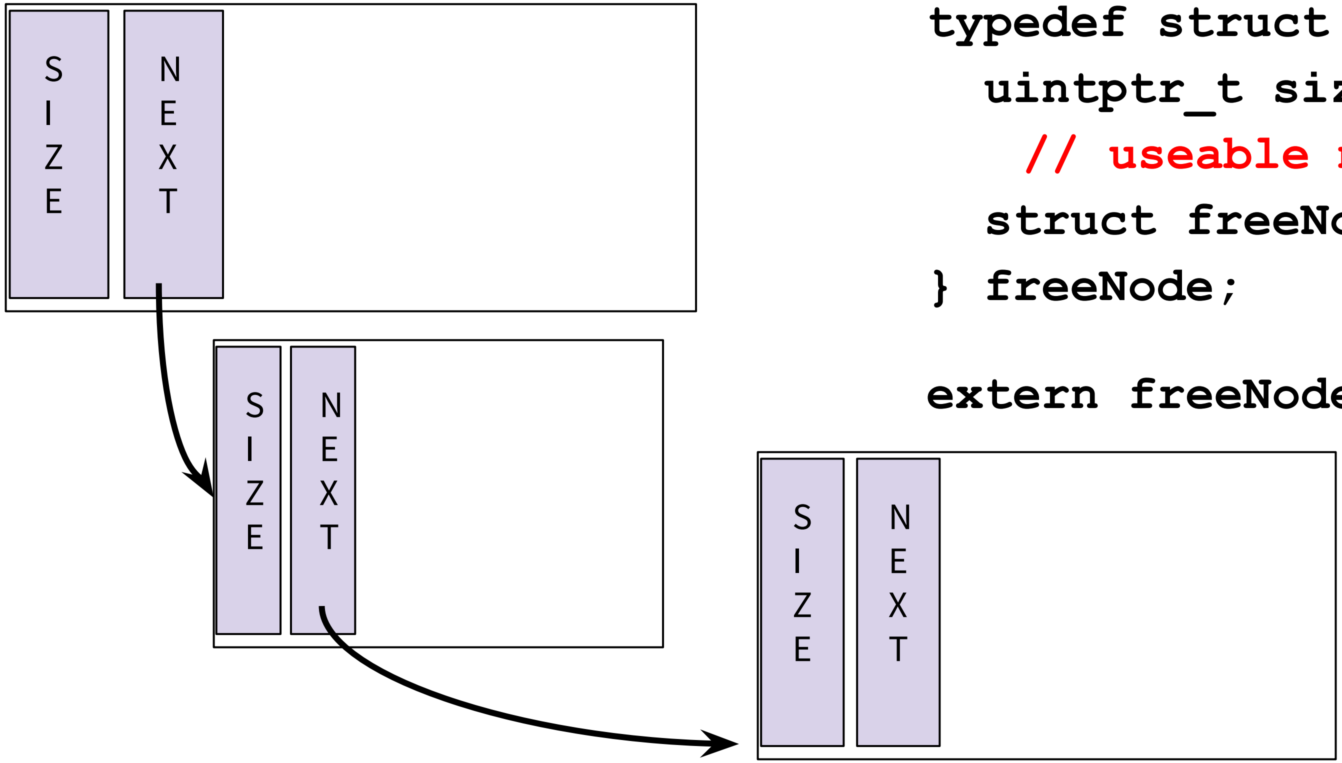
1. We use a system call (aka malloc) to get a big chunk of memory - like 4k-10k bytes.
2. We then parcel out pieces of this chunk to individual calls to getmem and mark them as reserved.
3. When someone calls freemem, we return the chunks to the set of free chunks.
4. How do we keep track of all of the available chunks vs reserved chunks?
   a. Use something called a "free list", which is a linked list of nodes that store information about available chunks.
   b. Shared by both getmem and freemem.
   c. Each block on the free list starts with an uintptr_t integer that gives its size followed by a pointer to the next block on the free list.
   d. To help keep data in dynamically allocated blocks properly aligned, we require that all of the blocks be a multiple of 16 bytes in size, and that their addresses also be a multiple of 16 (this is the same way that the built-in malloc works).

Approach, Cont.

Getmem request? Scan the free list looking for a block of storage that is at least as large as the amount requested, delete that block from the free list, and return a pointer to it to the caller.

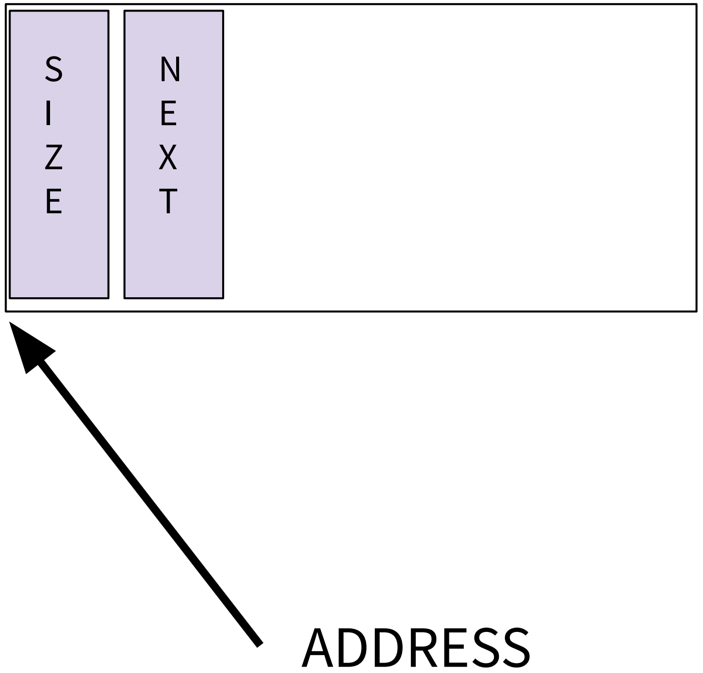
Freemem: return the given block to the free list, combining it with any adjacent free blocks if possible to create a single, larger block instead of several smaller ones.

What is a memory frame?



```
typedef struct freeNode {
  uintptr_t size;
   // useable memory
  struct freeNode* next;
} freeNode;

extern freeNode* freelist;
```

Addresses



What is the address?

- An integer pointing to the correct byte (`uintptr_t`)
- A pointer to a memory object (`void*`)

What can you do with it?

- Math - add or subtract an integer to go forward or backwards
- Cast between integer and (T*)
- If cast to (freeNode*) - access data of that type `freeNode->size, freeNode->next`

Approach: getting memory blocks

If, a large enough block exists, ‘getmem’ splits the block into an appropriate sized chunk and pointer to the rest of the block

Else, getmem needs to

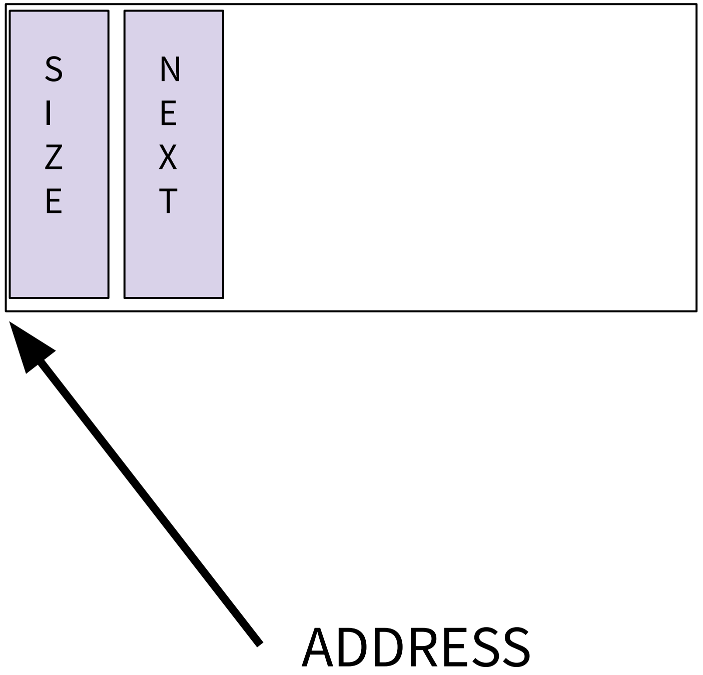
Get a good-sized block of storage from the underlying system.

Add it to the free list

Split it up, yielding a block that will satisfy the request (‘**if**’ condition)

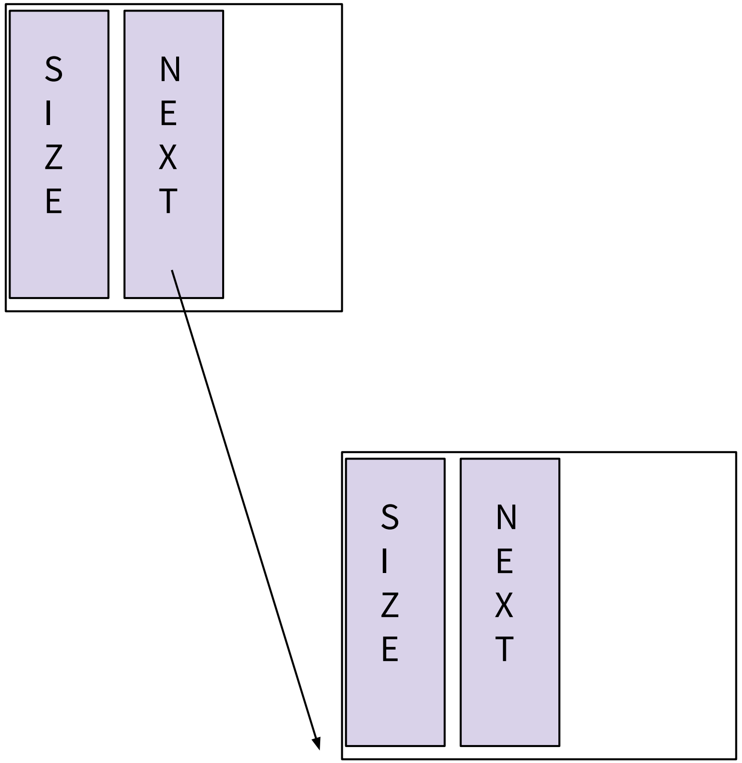
Note, Initial call to getmem finds it with no memory, and results in ‘**else**’ condition.

getmem



```
get_block (uintptr_t size) {

freeNode* currentNode = freelist;
while(currentNode) {
        if(currentNode->size >= minsize)
        ...
    return(uintptr_t)currentNode;
}

return((void*) block+NODESIZE);
// offset for user's purposes
```

getmem



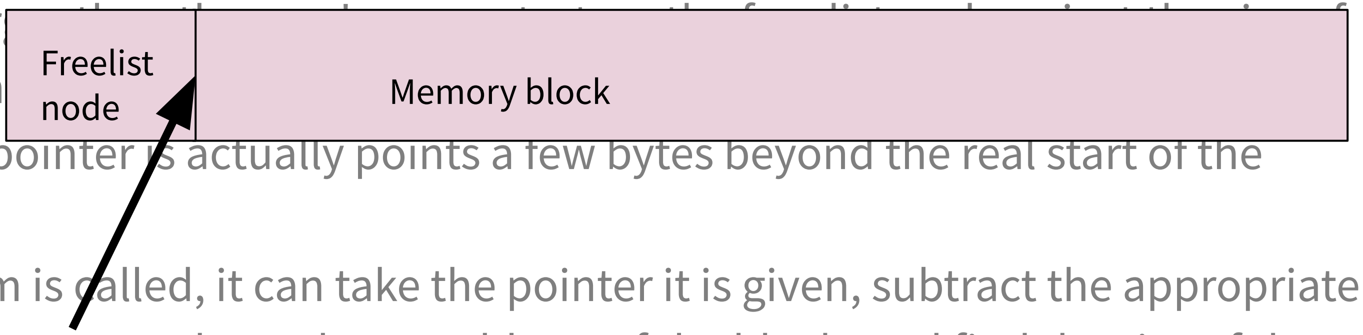
```
void split_node(freeNode* n, uintptr_t size) {
        freeNode* newNode =
                (freeNode*)((uintptr_t)(n) + size+NODESIZE);

 newNode->size = n->size - size - NODESIZE;
newNode->next = n->next;

 n->size = size;
 n->next = newNode;

….
```

Approach: returning memory

- Freemem gets a pointer to a block of storage and adds it to the free list, combining it with adjacent blocks on the list.
- Freemem isn't told is how big the block is and must find the size of the block.
- The usual way this is done is to have getmem actually allocate a block of memory that is a bit larger than the user's request, store the free list node or just the size of the block at the beginning of that block.
- The returned pointer is actually points a few bytes beyond the real start of the block.
- When freemem is called, it can take the pointer it is given, subtract the appropriate number of bytes to get the real start address of the block, and find the size of the block there.

Approach: returning memory

- Freemem gets a pointer to a block of storage and adds it to the free list, combining it with adjacent blocks on the list.
- Freemem isn't told is how big the block is and must find the size of the block.
- The usual way this is done is to have getmem actually allocate a block of memory that is a bit lar the block at th
- The returned pointer is actually points a few bytes beyond the real start of the block.
- When freemem is called, it can take the pointer it is given, subtract the appropriate number of bytes to get the real start address of the block, and find the size of the block there. `p-sizeof(freelist_node)->size`



Use ‘assert’ in C: `void check_heap ();`

Check for possible problems with the free list data structure. This function should use `assert`s to verify that:

- Blocks are ordered with increasing memory addresses
- Block sizes are positive numbers and no smaller than whatever minimum size you are using
- Blocks do not overlap (the start + length of a block is not an address in the middle of a later block on the list)
- Blocks are not touching (the start + length of a block should not be the address of the next block on the list)

If no errors are detected, this function should return silently after performing these tests. If an error is detected, then an `assert` should fail and cause the program to terminate at that point.

```
void check_heap() {

  if (!freelist) return;
  freeNode* currNode = freelist;
  uintptr_t mins= \
currNode->size;

  < …….>
  assert (mins >= MINSIZE);
}
```

Next up: C++ *(Want to read ahead?)*

Best place to start: C++ Primer, Lippman, Lajoie, Moo, 5th ed., Addison-Wesley, 2013

Every serious C++ programmer should also read: Effective C++, Meyers, 3rd ed., Addison-Wesley, 2005

Best practices for standard C++

Effective Modern C++, Meyers, O'Reilly, 2014

Additional "best practices" for C++11/C++14

Good online source: cplusplus.com

What is C++ ?

A big language - much bigger than C

Conveniences in addition to C (new/delete, function overloading, bigger std library)

Namespaces - similar to Java

Extras (casts, exceptions, templates, lambda functions)

Object Oriented - has classes and objects similar to Java

Why C++ ?

- C++ is C-like in
  - User-managed memory
  - Header files
  - Still use pointers
- C++ is Java like in
  - Object Oriented
  - Modern additions to language
- Knowing C++ may help understand both C & Java better