CSE 29
Lecture 15 Summary

February 24, 2026

📣 Logistical Things

- Exam Notes: Joe will make a piazza post about this.
- There are no Assignment 5 Resubmissions
  - Design Questions, PA, and PSet will all be due Friday Week 10

Review Questions

Answers in the next slide!

Array Notation	equivalent to...	Pointer Notation	
l-values `a[0]`	`(double)`	`*a`	l-values
l-values `a[i]`	`(double)`	`*(a+i)`	l-values
`&a[i]`	`(double*)`	`a+i`	
`a[i] = v`		`*(a+i) = v`	
`double* a;`	`a += 1`	`a = a+1`	

```
void* allocate_at(uint64_t* start, size_t size) {
  size_t current_size = block_size(start[0]);
  if(current_size > size) {
    uint64_t remaining = current_size - size - SLOT_SIZE;
    int next_block_index = (size / SLOT_SIZE) + 1;
    start[next_block_index] = remaining; // even, free
  }
  start[0] = size | 1; // busy
  return &start[1];
}
```

```
void* allocate_at(uint64_t* start, size_t size) {

  size_t current_size = block_size(                    );


  if(current_size > size) {
    uint64_t remaining = current_size - size - SLOT_SIZE;
    int next_block_index = (size / SLOT_SIZE) + 1;


                         = remaining; // even, free
  }
  start[0] = size | 1;


  return                 ;
}
```

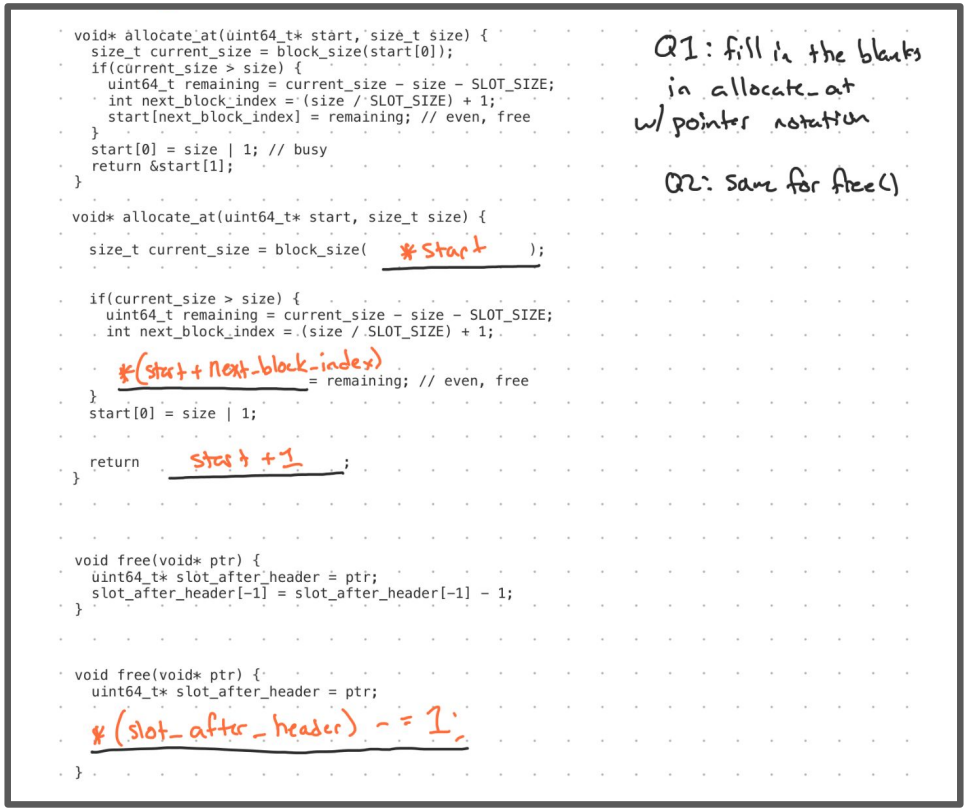
Q1: Fill in the blanks to use pointer notation

```
void free(void* ptr) {
  uint64_t* slot_after_header = ptr;
  slot_after_header[-1] = slot_after_header[-1] - 1;
}
```

```
void free(void* ptr) {
  uint64_t* slot_after_header = ptr;


}
```

Q2 do the same in free()

RQ Answers



```
void* allocate_at(uint64_t* start, size_t size) {
  size_t current_size = block_size(start[0]);
  if(current_size > size) {
    uint64_t remaining = current_size - size - SLOT_SIZE;
    int next_block_index = (size / SLOT_SIZE) + 1;
    start[next_block_index] = remaining; // even, free
  }
  start[0] = size | 1; // busy
  return &start[1];
}
```

```
void* allocate_at(uint64_t* start, size_t size) {

  size_t current_size = block_size(    *start     );

  if(current_size > size) {
    uint64_t remaining = current_size - size - SLOT_SIZE;
    int next_block_index = (size / SLOT_SIZE) + 1;

    *(start + next_block_index)
                        = remaining; // even, free
  }
  start[0] = size | 1;

  return     start + 1     ;
}
```

Q1: fill in the blanks in allocate_at w/ pointer notation

Q2: same for free()

```
void free(void* ptr) {
  uint64_t* slot_after_header = ptr;
  slot_after_header[-1] = slot_after_header[-1] - 1;
}
```

```
void free(void* ptr) {
  uint64_t* slot_after_header = ptr;
  *(slot_after_header) -= 1;
}
```

🤔 Questions

- What if you tried to do pointer arithmetic with a void pointer?
  - You would get an error, it's not allowed
- Is there something you can do with array notation that you can't do in pointer notation?
  - No, most likely not
- Is there a reason both of them exist?
  - There's a category of pointers that are arrays, and there's a category of pointers that is just pointing at a single value
- When we add 1 to a pointer, does it's value increase by 1? Does it move 1 byte?
  - No and no. While increasing its value by 1 would move it one byte, the pointer will move the amount of bytes that is the size of what it is pointing to. (8 for pointer, 4 for int etc.)
    - double * a; // pointer to 8 byte double | if it was 0xfff7123400
    - a += 1; // moves our pointer 8 bytes. | it is now 0xfff7123408

Virtual Memory

Let’s look at the handout

```
#include <unistd.h>
#include <stdio.h>
#include <sys/wait.h>

int main() {
  int value = 17;
  int pid = fork();
  if(pid == 0) {
    printf("In child before assignment: %p => %d\n", &value, value);
    value = 999;
    while(1) {
      printf("In child:\t%p => %d\n", &value, value);
    }
  }
  else {
    printf("In parent before assignment: %p => %d\n", &value, value);
    value = 3333;
    while(1) {
      printf("In parent:\t%p => %d\n", &value, value);
    }
  }
}
```

- There’s a `fork()` to make a parent and child process
- We use the same variable `value` and change it to a different value
- We print out the address of value in the parent and in the child process

Think about these questions

```
#include <unistd.h>
#include <stdio.h>
#include <sys/wait.h>

int main() {
  int value = 17;
  int pid = fork();
  if(pid == 0) {
    printf("In child before assignment: %p => %d\n", &value, value);
    value = 999;
    while(1) {
      printf("In child:\t%p => %d\n", &value, value);
    }
  }
  else {
    printf("In parent before assignment: %p => %d\n", &value, value);
    value = 3333;
    while(1) {
      printf("In parent:\t%p => %d\n", &value, value);
    }
  }
}
```

What prints?

How many different addresses print?

always print 999?

always print 3333?

Outline for answer

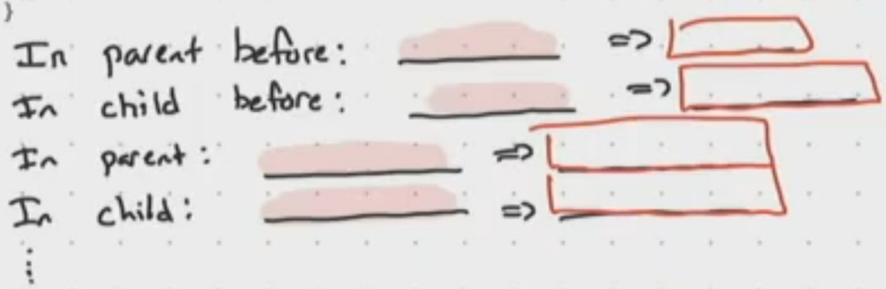
```c
#include <unistd.h>
#include <stdio.h>
#include <sys/wait.h>

int main() {
  int value = 17;
  int pid = fork();
  if(pid == 0) {
    printf("In child before assignment: %p => %d\n", &value, value);
    value = 999;
    while(1) {
      printf("In child:\t%p => %d\n", &value, value);
    }
  }
  else {
    printf("In parent before assignment: %p => %d\n", &value, value);
    value = 3333;
    while(1) {
      printf("In parent:\t%p => %d\n", &value, value);
    }
  }
}
```

What prints overall?

How many different addresses print?

Does this always print 999?

Does this always print 3333?

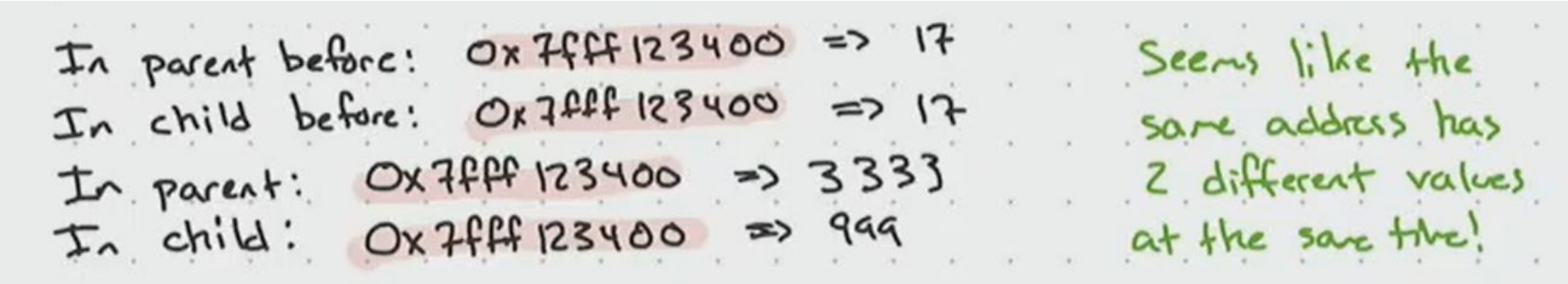


Guesses from students

- The two `value`s are at two different addresses with different values
- The two `value`s are at the same address and somehow share the address
- There is one address that is shared at the same address and one assignment of the addresses will win

What actually happens

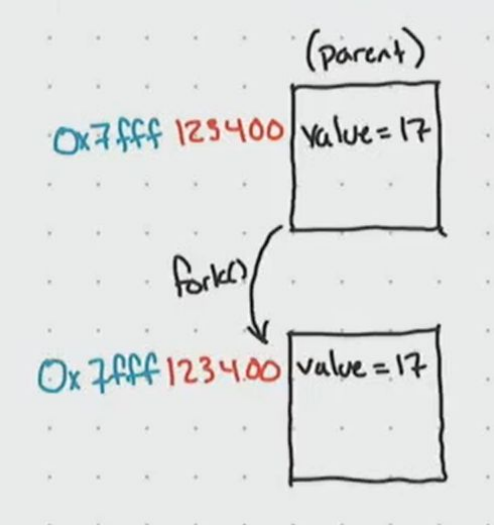
When the program is ran, this is what is printed



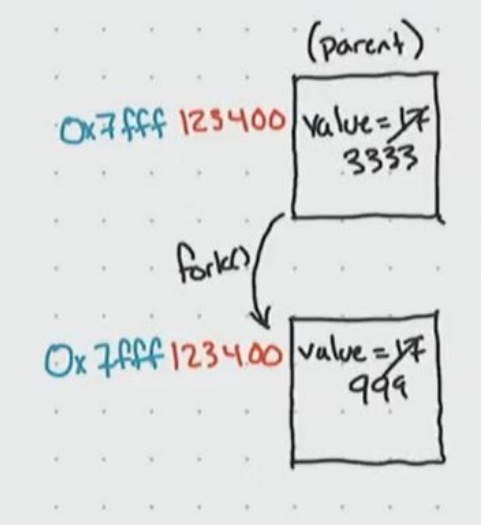
The same address is printed out *and* it prints 2 different values!

What the operating system is doing

Forking



Re-assigning `value`



Processes are a data structure stored in OS (Addresses are data structures too!)

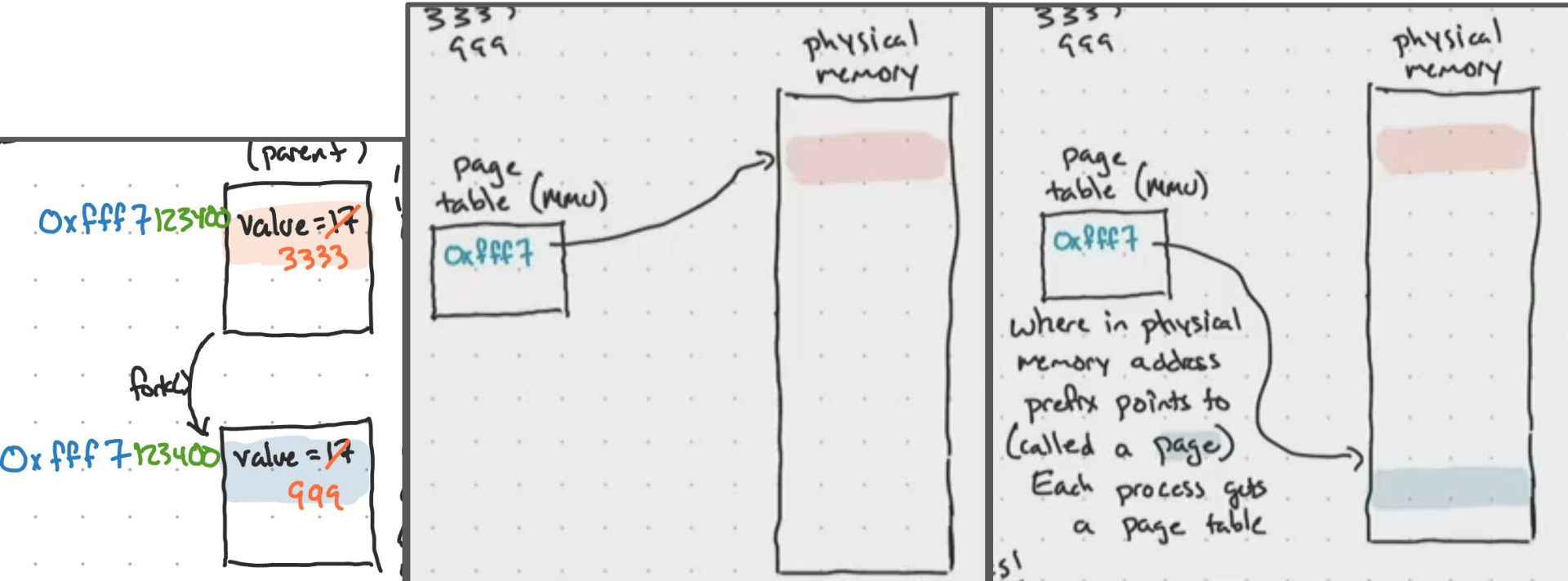
Memory Management Unit (MMU) & Page Table

- This is a piece of hardware in your computer
- Store a mapping from address prefixes to physical memory addresses
- We keep swapping page tables in and out of the MMU and
  - Each process holds its own page table (the parent and child each have one)
- The parent and child processes are running at different locations in physical memory
- A chunk of physical memory is a page
  - Pages are assigned to processes
- Both `mmap` and `sbrk` return pages

Restructure this to be make more sense

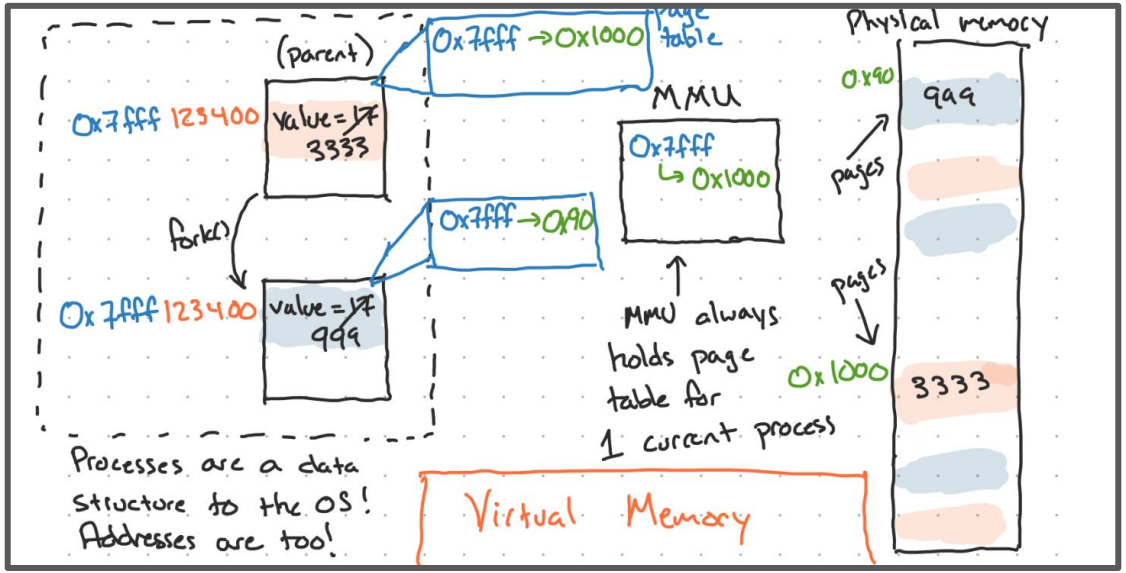
Page Table drawings

When the OS switches processes, this prefix switches what part of physical memory it points to



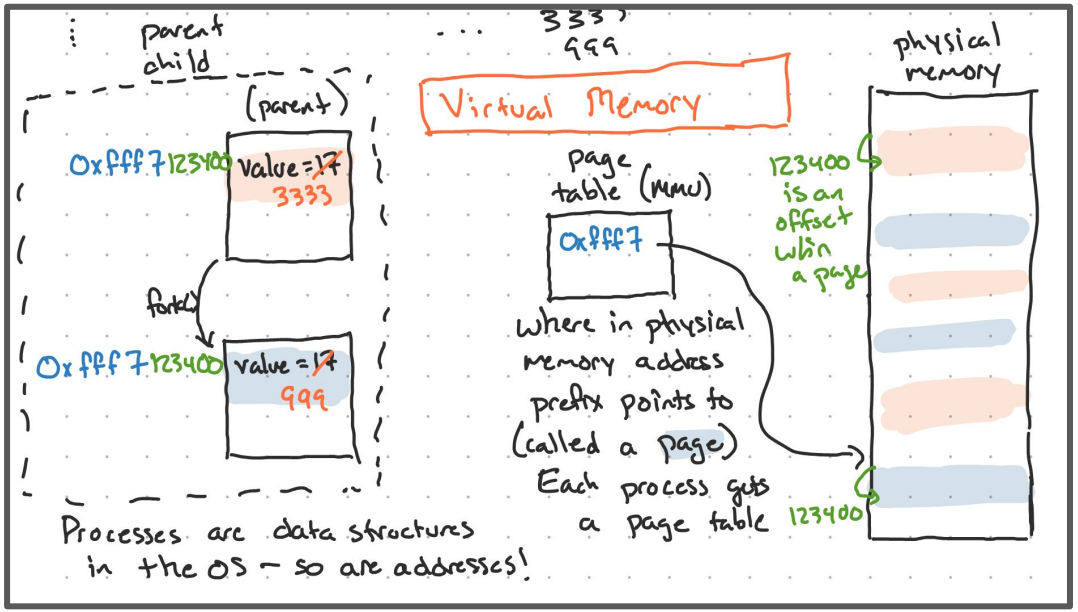
Virtual Memory

- This is how the OS maintains the illusion that each process has the same memory



Virtual Memory

- The virtual address is split into 2 parts
  - Prefix→ 0xfff7
    - This determines which page (red and orange highlighted parts in physical memory
  - Offset → 0x123400
    - This is how far the address is from the start of the page. (so value is 0x123400 bytes from the beginning of the page)

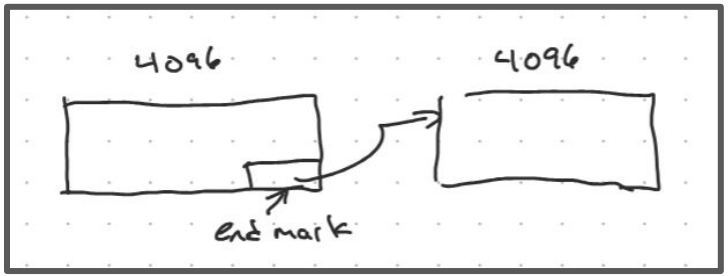


Questions

- Why did we decide to do virtual memory rather than just use physical memory addresses?
  - When a process starts, we don’t know if a process is large (Google Chrome) or small (ls)
  - It would be difficult to know how much memory we need from the start of a process
  - With virtual memory, we can ask for more memory and elastically grow if the process needs to get bigger
  - With virtual memory, we use multiple unconnected pages (chunks in physical memory) for a processes
- How does hibernate and sleep work in the OS?
  - The OS saves/copies all the pages onto your disk at sleep and then copies it back to memory when you wake your computer up again

Questions

- So does malloc and free ever interact with the page table?
  - When you call malloc, it will ask for new pages if it needs more space (`mmap`)
  - The endmark of the first block will just be mapped to another page, linking it together



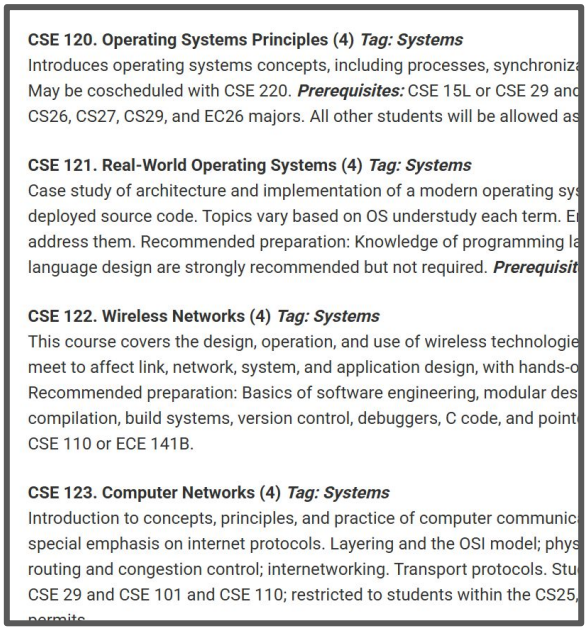
- Are heap blocks always 4096?
  - No, depends on your machine
- What's the difference between a program and a process
  - A program is a binary file
  - A process is a running program
    - Processes get a page table, programs don't need one

Questions

- The MMU is fixed size, so does that mean there are a fixed number of processes?
  - This is unrelated
  - The MMU only holds the page table for one process at a time
  - Although you can run out of pages you can map

Further Learning

- If you're interested in operating systems and hardware, we recommend exploring the 120 series, the 140 series, and CSE 130!

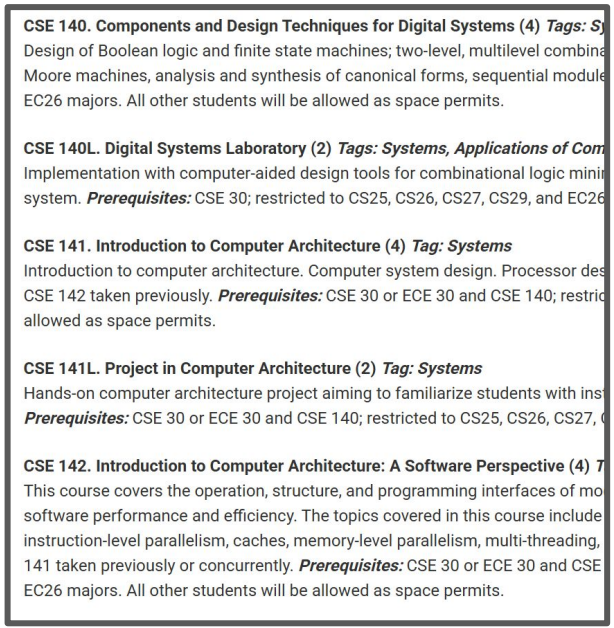


CSE 120. Operating Systems Principles (4) ***Tag: Systems***
Introduces operating systems concepts, including processes, synchroniza
May be coscheduled with CSE 220. ***Prerequisites:*** CSE 15L or CSE 29 and
CS26, CS27, CS29, and EC26 majors. All other students will be allowed as

CSE 121. Real-World Operating Systems (4) ***Tag: Systems***
Case study of architecture and implementation of a modern operating sys
deployed source code. Topics vary based on OS understudy each term. E
address them. Recommended preparation: Knowledge of programming la
language design are strongly recommended but not required. ***Prerequisit***

CSE 122. Wireless Networks (4) ***Tag: Systems***
This course covers the design, operation, and use of wireless technologie
meet to affect link, network, system, and application design, with hands-o
Recommended preparation: Basics of software engineering, modular des
compilation, build systems, version control, debuggers, C code, and point
CSE 110 or ECE 141B.

CSE 123. Computer Networks (4) ***Tag: Systems***
Introduction to concepts, principles, and practice of computer communic
special emphasis on internet protocols. Layering and the OSI model; phys
routing and congestion control; internetworking. Transport protocols. Stu
CSE 29 and CSE 101 and CSE 110; restricted to students within the CS25,
permits.



CSE 140. Components and Design Techniques for Digital Systems (4) ***Tags: Sy***
Design of Boolean logic and finite state machines; two-level, multilevel combina
Moore machines, analysis and synthesis of canonical forms, sequential module
EC26 majors. All other students will be allowed as space permits.

CSE 140L. Digital Systems Laboratory (2) ***Tags: Systems, Applications of Com***
Implementation with computer-aided design tools for combinational logic mini
system. ***Prerequisites:*** CSE 30; restricted to CS25, CS26, CS27, CS29, and EC26

CSE 141. Introduction to Computer Architecture (4) ***Tag: Systems***
Introduction to computer architecture. Computer system design. Processor des
CSE 142 taken previously. ***Prerequisites:*** CSE 30 or ECE 30 and CSE 140; restric
allowed as space permits.

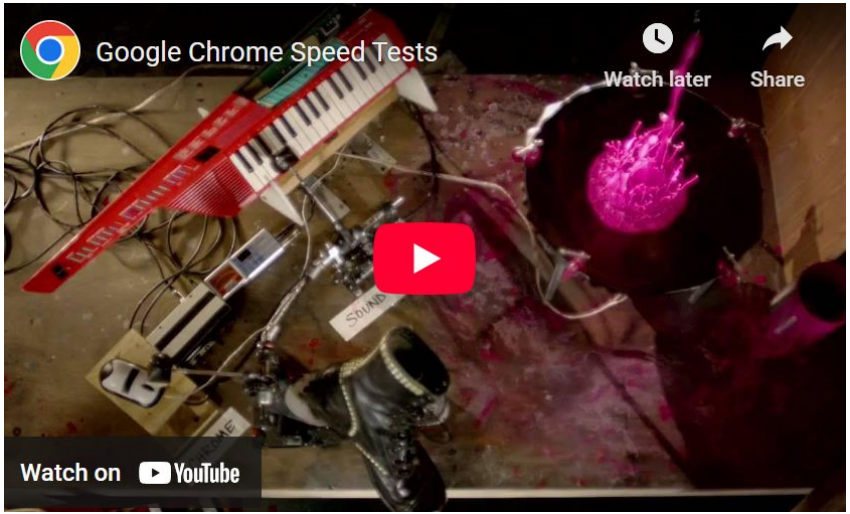
CSE 141L. Project in Computer Architecture (2) ***Tag: Systems***
Hands-on computer architecture project aiming to familiarize students with inst
Prerequisites: CSE 30 or ECE 30 and CSE 140; restricted to CS25, CS26, CS27,

CSE 142. Introduction to Computer Architecture: A Software Perspective (4) ***T***
This course covers the operation, structure, and programming interfaces of mo
software performance and efficiency. The topics covered in this course include
instruction-level parallelism, caches, memory-level parallelism, multi-threading,
141 taken previously or concurrently. ***Prerequisites:*** CSE 30 or ECE 30 and CSE
EC26 majors. All other students will be allowed as space permits.

Cool Video: Chrome Potato Gun Speed Test

Watch this speed test for the Chrome Browser!

[Google Chrome Blog: Potato gun, lightning, and sonic magic: Unconventional speed tests for the browser]



Security Vulnerabilities with free

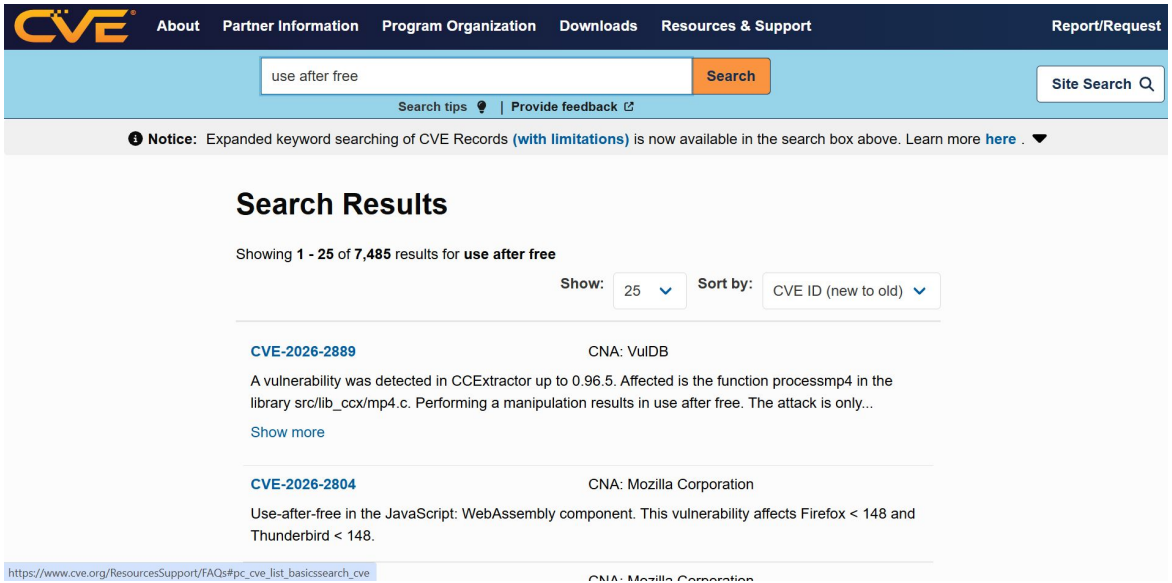
Use-After-Free Vulnerabilities

This is a vulnerability where you allocate something, you free it, and then you accidentally use that memory again

This memory could store sensitive information like passwords or some other critical information

Security Vulnerabilities with Use-After-Free

You can see real and recent vulnerabilities happening because of Use-After-Free



https://www.cve.org/CVERecord/SearchResults?query=use+after+free

The Takeaway

- Using `free` correctly and securely is hard
  - These Google engineers are also making mistakes
- C is a bad language to build applications connected to the internet
  - People are now rewriting browsers in languages that are memory safe

Joe's Notes (11am)

(Review Qs delayed - chill out :)

Week 10 Exam - In lab still 45m, still PL

"Final" (Make-ups) - Mon 12-4pm
Wed 10-4pm

Test length 1h40m
Can retake any or all of the exams

Makeup 1
Makeup 2
Makeup 3

	Array Notation	equivalent to...	Pointer Notation	
l-values	a[0]	(double)	*a	l-values
	a[i]	(double)	*(a+i)	
	&a[i]	(double*)	a+i	
	a[i] = v		*(a+i) = v	
	double* a;		a += 1	a = a+1

```
void* allocate_at(uint64_t* start, size_t size) {
  size_t current_size = block_size(start[0]);
  if(current_size > size) {
    uint64_t remaining = current_size - size - SLOT_SIZE;
    int next_block_index = (size / SLOT_SIZE) + 1;
    start[next_block_index] = remaining; // even, free
  }
  start[0] = size | 1; // busy
  return &start[1];
}
```

```
void* allocate_at(uint64_t* start, size_t size) {

  size_t current_size = block_size( *start );

  if(current_size > size) {
    uint64_t remaining = current_size - size - SLOT_SIZE;
    int next_block_index = (size / SLOT_SIZE) + 1;

    *(start + next_block_index) = remaining; // even, free
  }
  start[0] = size | 1;

  return start + 1;
}
```

```
void free(void* ptr) {
  uint64_t* slot_after_header = ptr;
  slot_after_header[-1] = slot_after_header[-1] - 1;
}
```

```
void free(void* ptr) {
  uint64_t* slot_after_header = ptr;
  *(slot_after_header) -= 1;
}
```

Q1: fill in the blanks in allocate_at w/ pointer notation

Q2: same for free()

```
#include <unistd.h>
#include <stdio.h>
#include <sys/wait.h>

int main() {
  int value = 17;
  int pid = fork();
  if(pid == 0) {
    printf("In child before assignment: %p => %d\n", &value, value);
    value = 999;
    while(1) {
      printf("In child:\t%p => %d\n", &value, value);
    }
  }
  else {
    printf("In parent before assignment: %p => %d\n", &value, value);
    value = 3333;
    while(1) {
      printf("In parent:\t%p => %d\n", &value, value);
    }
  }
}
```

What prints overall?

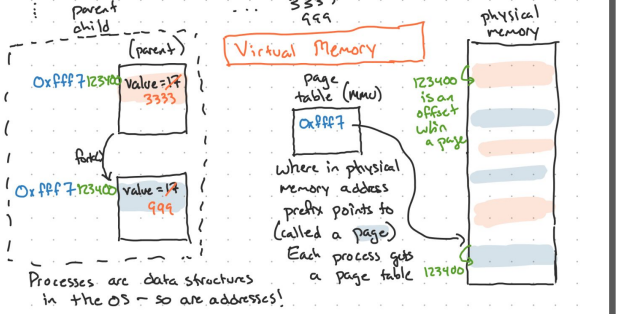
How many different addresses print? 1 — only 1 address prints

yes — Does this always print 999?

Does this always print 3333? yes

In parent before: 0xfff7123400 => 17
In child before: 0xfff7123400 => 17
In parent: 0xfff7123400 => 3333
In child: 0xfff7123400 => 999
⋮ ... 3333 999

Obs: 1 address is holding 2 diff values?



Processes are data structures in the OS - so are addresses!

Joe's Notes (12:30pm)