

Welcome + Threads + Processes

Problem Solving Lab for CS111

Stanford CS111ACE, Spring 2026

Attendance: tinyurl.com/cs111ace-section1-spr26

Hello there!

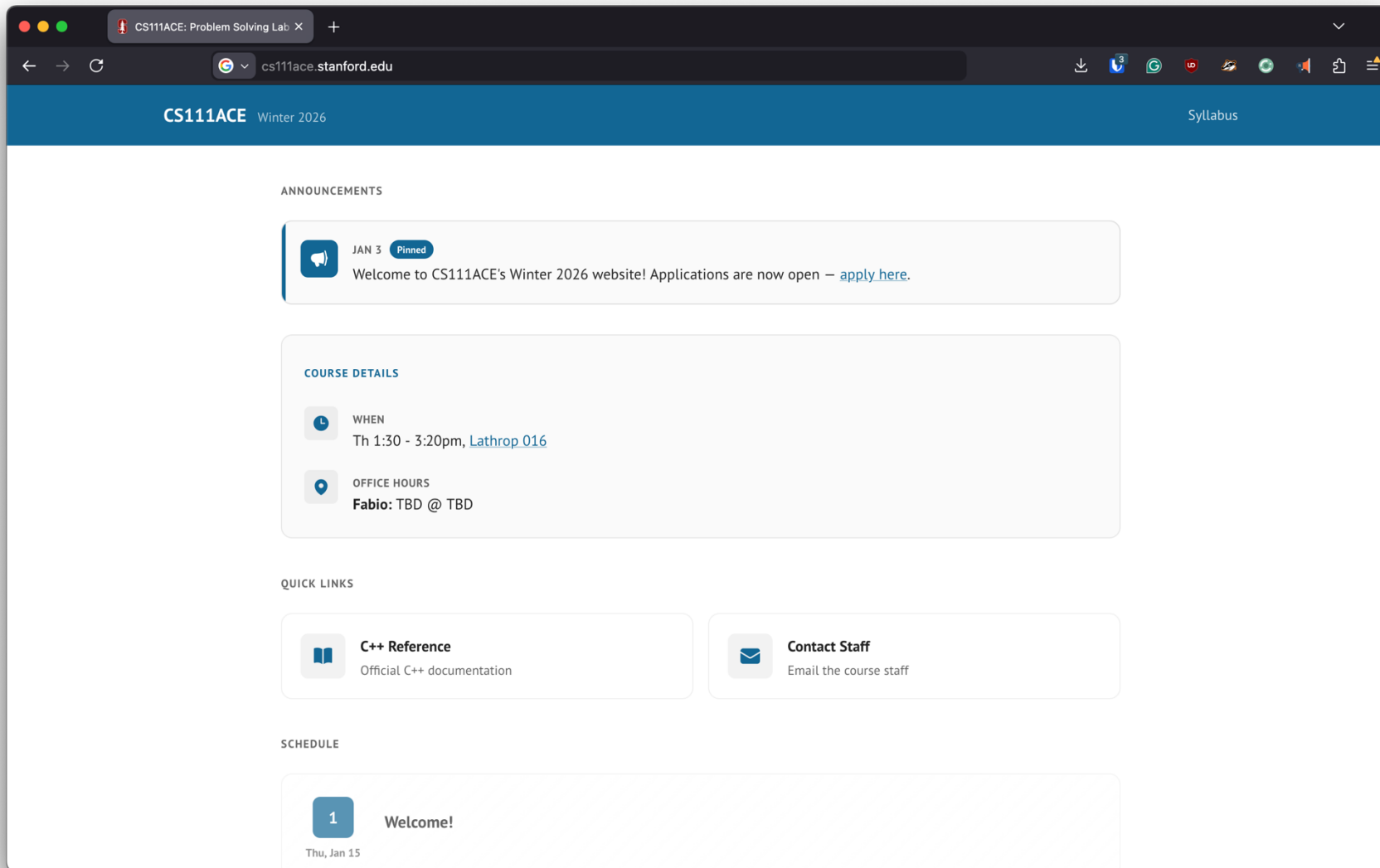
- MSCS (AI), BSCS (Systems)
- I enjoy teaching; section led (CS198) and taught CS106L in the past
- I spend my free time trying to cook, working out, and hanging out with friends!



Icebreaker

- What's your name
- Favorite hobby

What's CS111ACE about?



All course materials will be posted at cs111ace.stanford.edu

Syllabus

Adapted from the policies in previous CS111ACE offerings taught by Trip Master and Avery Wang.

CS111ACE – Problem Solving Lab for CS111

Section Time: Th 1:30 - 3:20pm, [Lathrop 016](#)

Course Assistant: Fabio Ibanez

Contact: fabioi@stanford.edu

Office Hours: TBD

Overview and Learning Goals

Additional Calculus for Engineers (ACE) is designed to provide the skills and solid foundation in mathematics, computational math in engineering, and computer science to undergraduate students interested in pursuing an engineering degree. The goal of ACE is to increase confidence and increase content knowledge through small group interactive sessions and the academic resources provided to students enrolled in the program.

CS 111 introduces the basic facilities provided by modern operating systems. The course is divided into three major sections:

- **File systems:** storage devices, disk management and scheduling, directories, and crash recovery.
- **Concurrency:** managing multiple tasks that execute at the same time and share resources. Topics include processes and threads, context switching, synchronization, scheduling, and deadlock.
- **Memory management:** dynamic memory allocation, dynamic address translation, virtual memory, and demand paging.

CS 111ACE is designed to (1) review the theoretical concepts from CS 111, and (2) serve as a collaborative lab to

Attendance

- As per the syllabus, you are allowed **1 unexcused absence**, no questions asked, and **1 excused absence**, where you must let me know in advance.
- Other absences beyond ***must*** be approved directly by the Lead ACE CAs, Bryant Perkins (bperk25) and Alka Panda (alkap)
- Being more than 15 minutes late or departing early without notifying me in advance will count as an absence

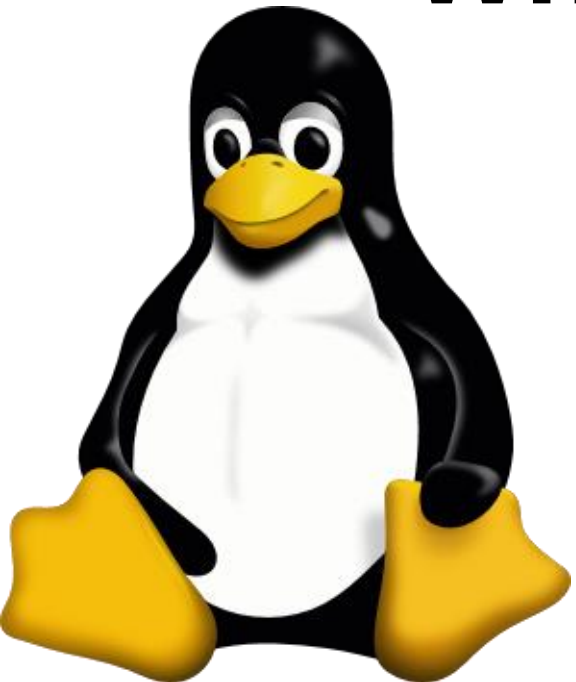
Honor Code

- CS111 ACE takes the Honor Code very seriously. Please see the updated Honor Code linked [here](#)
- As mentioned in the syllabus, the entire CS111 staff including me cannot look at your code directly.
- Please be doubly certain that you review CS111's collaboration policy [here](#)
- I was told to inform you that: filling out an attendance survey and not being in an ACE section is an honor code violation

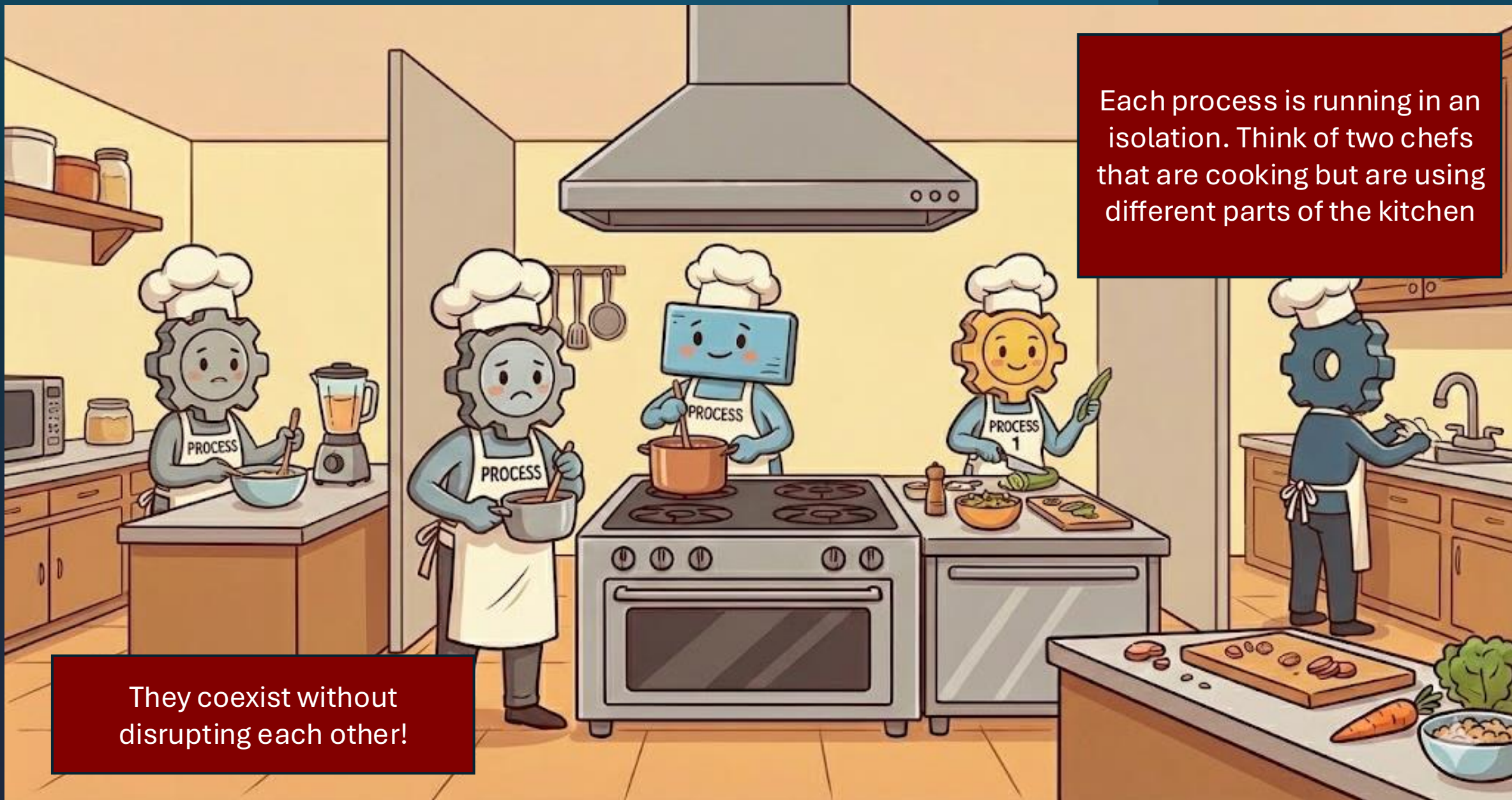
Class norms

- Take space, make space
- Avoid unrelated jargon & discussion during section– I'm happy to talk about things outside of the scope of the class offline 😊
- Please respect each other
- Normalize asking questions, and asking me to clarify things if unclear – I'm very happy to do this

What questions can I answer?



Multiprocessing



Each process is running in an isolation. Think of two chefs that are cooking but are using different parts of the kitchen

They coexist without disrupting each other!

How do we Create a Process?

```
void spawn_child()
{
    auto pidOrZero = fork();
    if (pidOrZero == 0)
    {
        printf("I am the child process\n");
    }
    else
    {
        printf("I am the parent process\n");
    }
}
```

- We can use the fork() system call
- fork returns either 0 if it's the child process, or the pid of the child if it's the parent

Caveat

```
void spawn_child()
{
    auto pidOrZero = fork();
    if (pidOrZero == 0)
    {
        printf("I am the child process\n");
    }
    else
    {
        printf("I am the parent process\n");
    }
}
```

- The parent must wait for the child process after it spawns it
- How does it do that?

pid_t waitpid(pid_t pid, int* status, int options)

- The waitpid function allows a parent to *clean up* the child process that it created
- Returns
 1. the pid of the child process that it *reaped*
 2. -1 if there was an issue cleaning up (i.e. there are no children to *reap/clean up*)

`pid_t waitpid(pid_t pid, int* status, int options)`

- The `waitpid` function allows a parent to *clean up* the child process that it created
- Returns
 1. the pid of the child process that it *reaped*
 2. -1 if there was an issue cleaning up (i.e. to *reap/clean up*)

You can pass in -1 as the pid if you want to reap any child



`pid_t waitpid(pid_t pid, int* status, int options)`

- The `waitpid` function allows a parent to *clean up* the child process that it created
- Returns
 1. the pid of the child process that it *reaped*
 2. -1 if there was an issue cleaning up (i.e. the child didn't *reap/clean up*)

Can tell us about whether or not the child gracefully exited or had an error



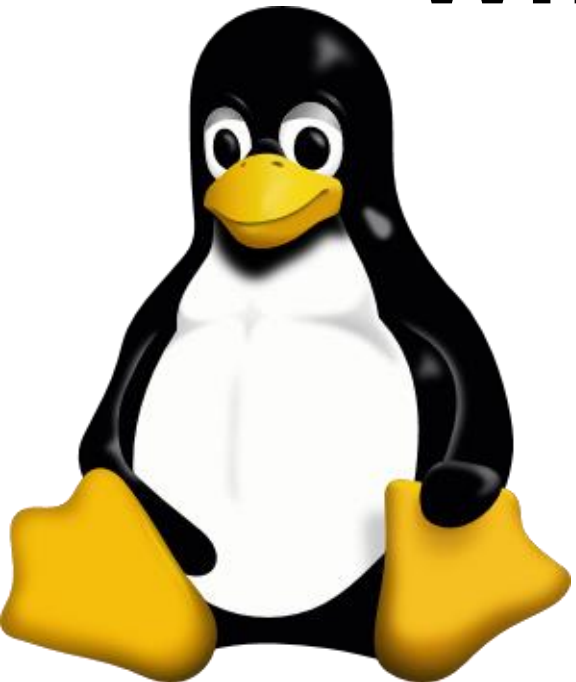
`pid_t waitpid(pid_t pid, int* status, int options)`

- The `waitpid` function allows a parent to *clean up* the child process that it created
- Returns
 1. the pid of the child process that it *reaped*
 2. -1 if there was an issue cleaning up (i.e. the child didn't *reap/clean up*)

Don't worry too much
about the options



What questions can I answer?



What happens (Think-pair-share)?

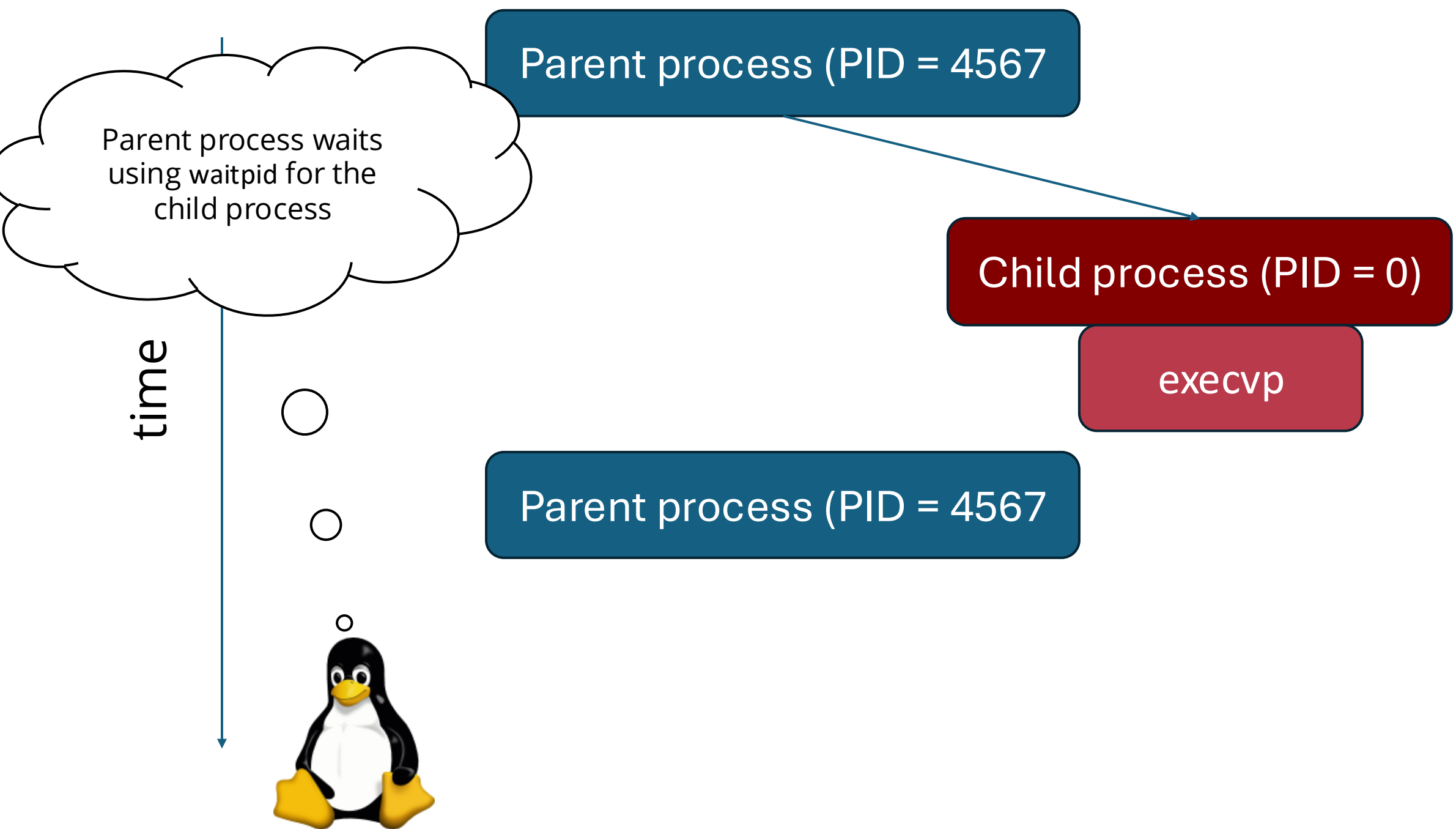
```
void spawn_and_wait_for_child()
{
    auto pidOrZero = fork();
    printf("Hi from the %s pid: %d\n", (pidOrZero == 0 ? "child" : "parent"), pidOrZero);

    int status;
    if (pidOrZero != 0) {
        waitpid(-1, &status, 0);
    }
}
```

- What order do the “Hello” statements print in?
- What happens if I remove the last line?
- What happens if I remove the **if** check on the penultimate line?

int execvp(const char * path, char *argv[])

- Another system call that allows you to run the program at the path path
- Passes argv as the arguments to that program
- The process in which execvp is called is completely consumed unless an error occurs



Extra practice

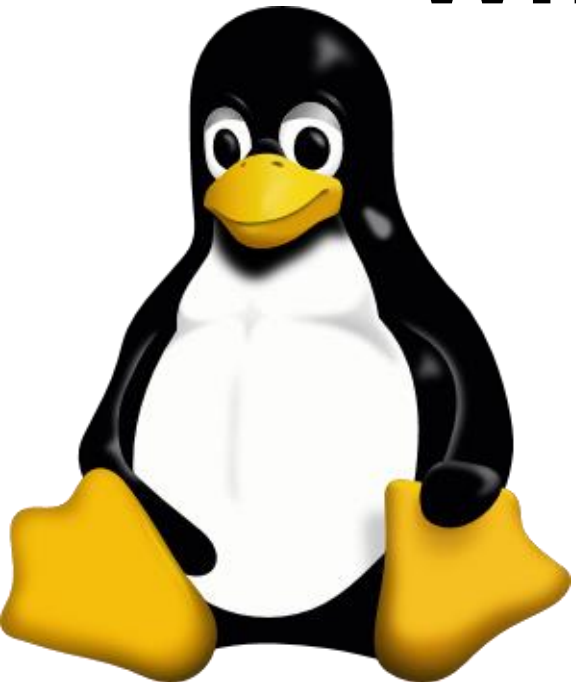
```
int main(void)
{
    int* stuff = malloc(sizeof(int)*1);
    *stuff = 5;
    pid_t pid = fork();
    printf("The last digit of pi is %d\n", *stuff);
    if (pid == 0)
        *stuff = 6
}
```

Extra practice

What will C print?

```
int main(void)
{
    char** argv = (char**) malloc(3*sizeof(char*));
    argv[0] = "/bin/ls";
    argv[1] = ".";
    argv[2] = NULL;
    for (int i = 0; i < 10; i++) {
        printf("%d\n", i);
        if (i == 3)
            execv("/bin/ls", argv);
    }
}
```

What questions can I answer?



Processes

- Isolated virtual addresses
- Can run external programs (fork-exec)
- Hard to coordinate multiple tasks within the same program

Threads

- Share virtual address space (threads can refer to the same memory)
- Can't run external programs as easily
- Can coordinate multiple tasks within one program

What is Multithreading?

How do we run code?

Source Code

```
#include <iostream>

void task(const std::string& name) {
    for (int i = 0; i < 3; ++i)
        std::cout << name << " says " << i << std::endl;
}

int main() {
    task("A");
    task("B");
    return 0;
}
```

Runs on



A multithreaded approach

```
#include <iostream>
#include <thread>

void task(const std::string& name) {
    for (int i = 0; i < 3; ++i)
        std::cout << name << " says " << i << std::endl;
}

int main() {
    std::thread t1(task, "A");
    std::thread t2(task, "B");

    t1.join();
    t2.join();

    return 0;
}
```

Thread 1

Thread 2





Multithreading

- Allows you to run multiple functions in *one* program concurrently
- Each thread runs within the same process, so they **share a virtual address space**.
 - A byproduct of this is that communication between threads is trivial
 - This also means that if two threads try to use the same resource (memory, file descriptors, streams, etc.), at the same time, it can have undefined behavior
- Each thread runs performs its function independently of the other

Revisiting our code

```
#include <iostream>
#include <thread>

void task(const std::string& name) {
    for (int i = 0; i < 3; ++i)
        std::cout << name << " says " << i << std::endl;
}

int main() {
    std::thread t1(task, "A");
    std::thread t2(task, "B");

    t1.join();
    t2.join();

    return 0;
}
```

Possible Orderings

A says 0
B says 0
A says 1
B says 1
A says 2
B says 2

A says 0
A says 1
A says 2
B says 0
B says 1
B says 2

A says 0
B says 0
B says 1
A says 1
B says 2
A says 2

B says 0
B says 1
B says 2
A says 0
A says 1
A says 2

Revisiting our code

```
#include <iostream>
#include <thread>

void task(const std::string& name) {
    for (int i = 0; i < 3; ++i)
        std::cout << name << " says " << i << std::endl;
}

int main() {
    std::thread t1(task, "A");
    std::thread t2(task, "B");

    t1.join();
    t2.join();

    return 0;
}
```

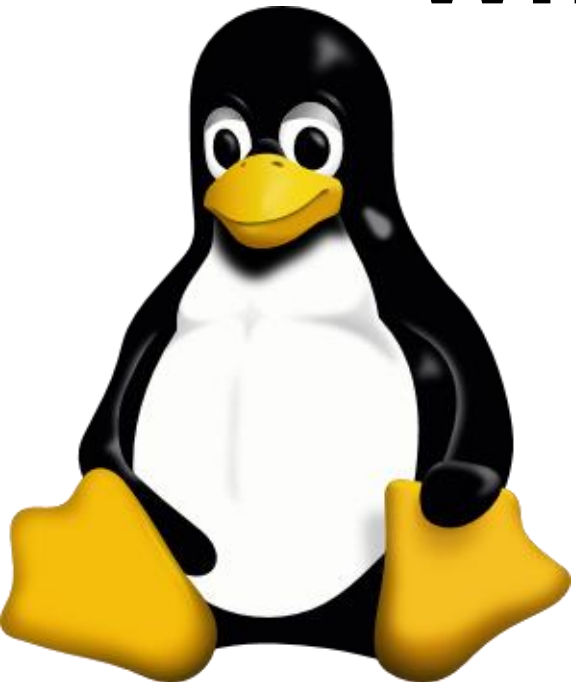
Another Possible Ordering

```
B says A says 0
B says 1
B says 2
0
A says 1
A says 2
```

What happened here? 🤔

If two threads try to use the same resource at the same time, it can have undefined behavior

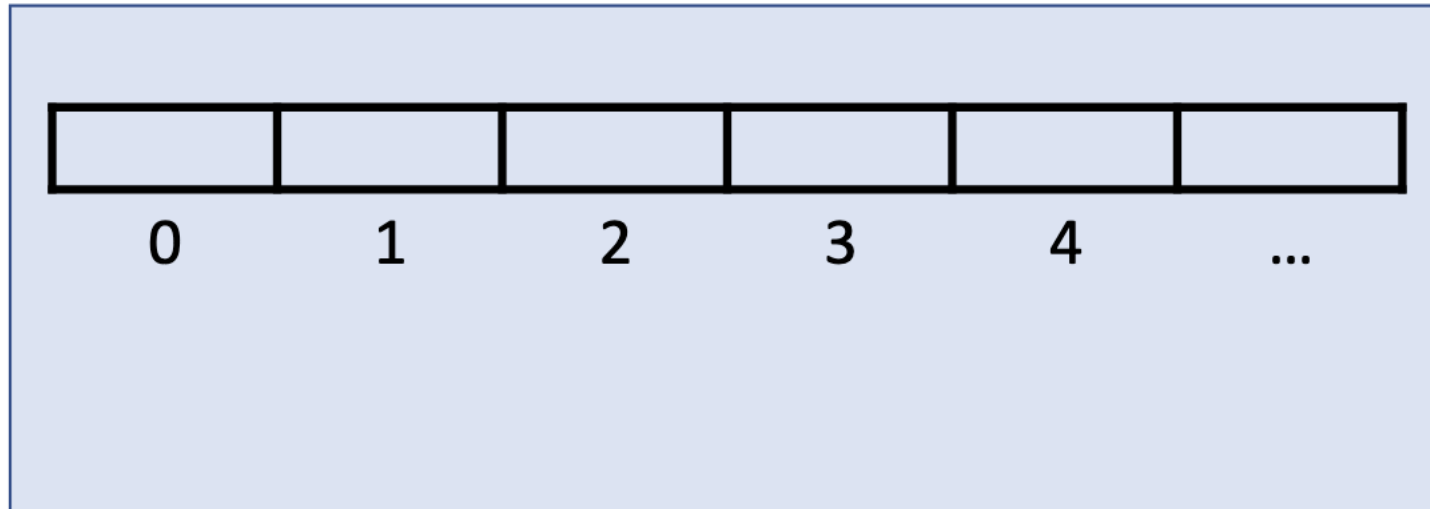
What questions can I answer?



**How do we switch between
running threads**

Process Control Block (PCB)

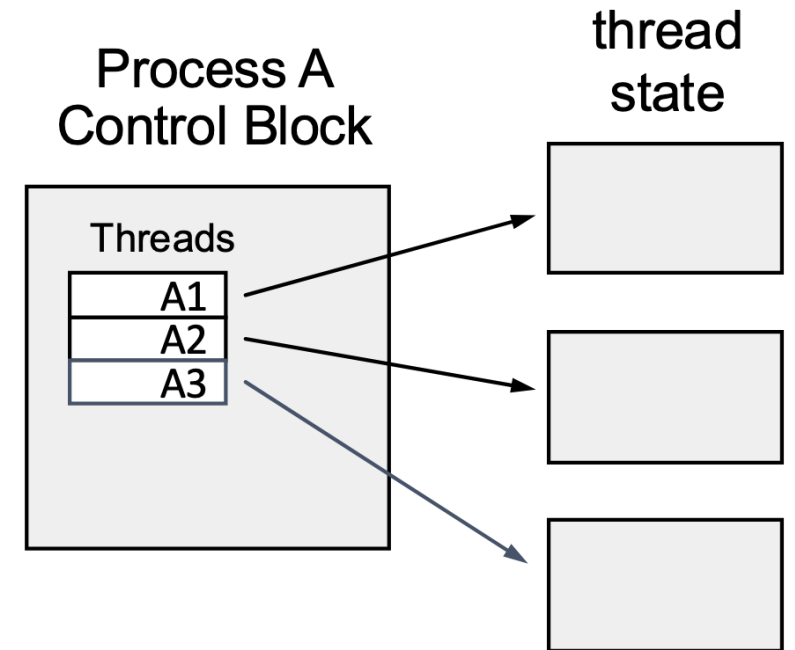
Process Control Block



Remember this thing that
we discussed a few weeks
back?

The Process Control Block (PCB)

- Contains information about the *state* of each thread that is running in a process
- We use this information to switch between running threads

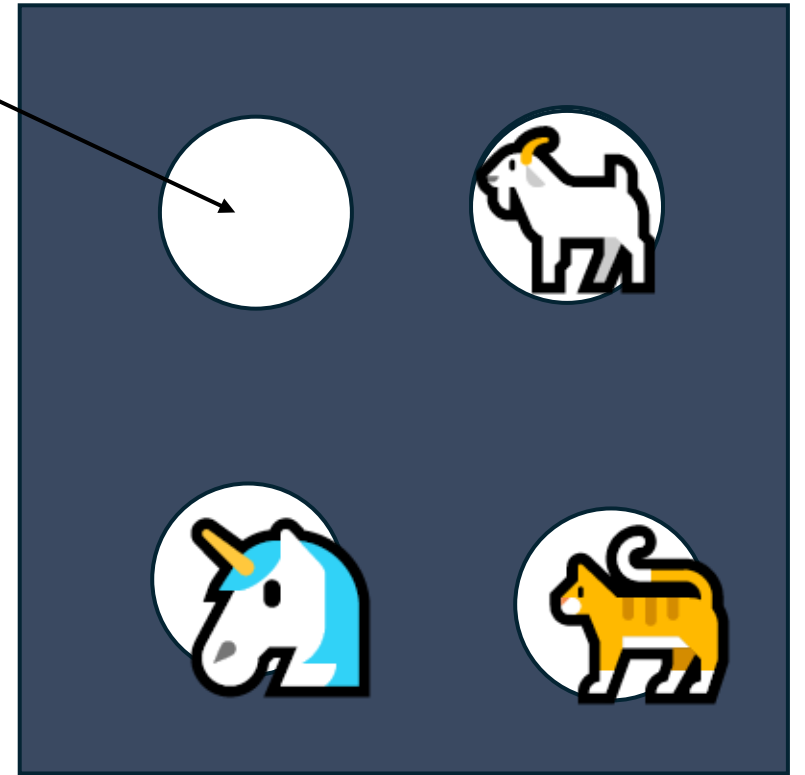
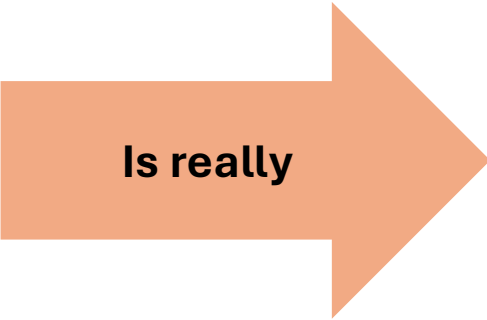
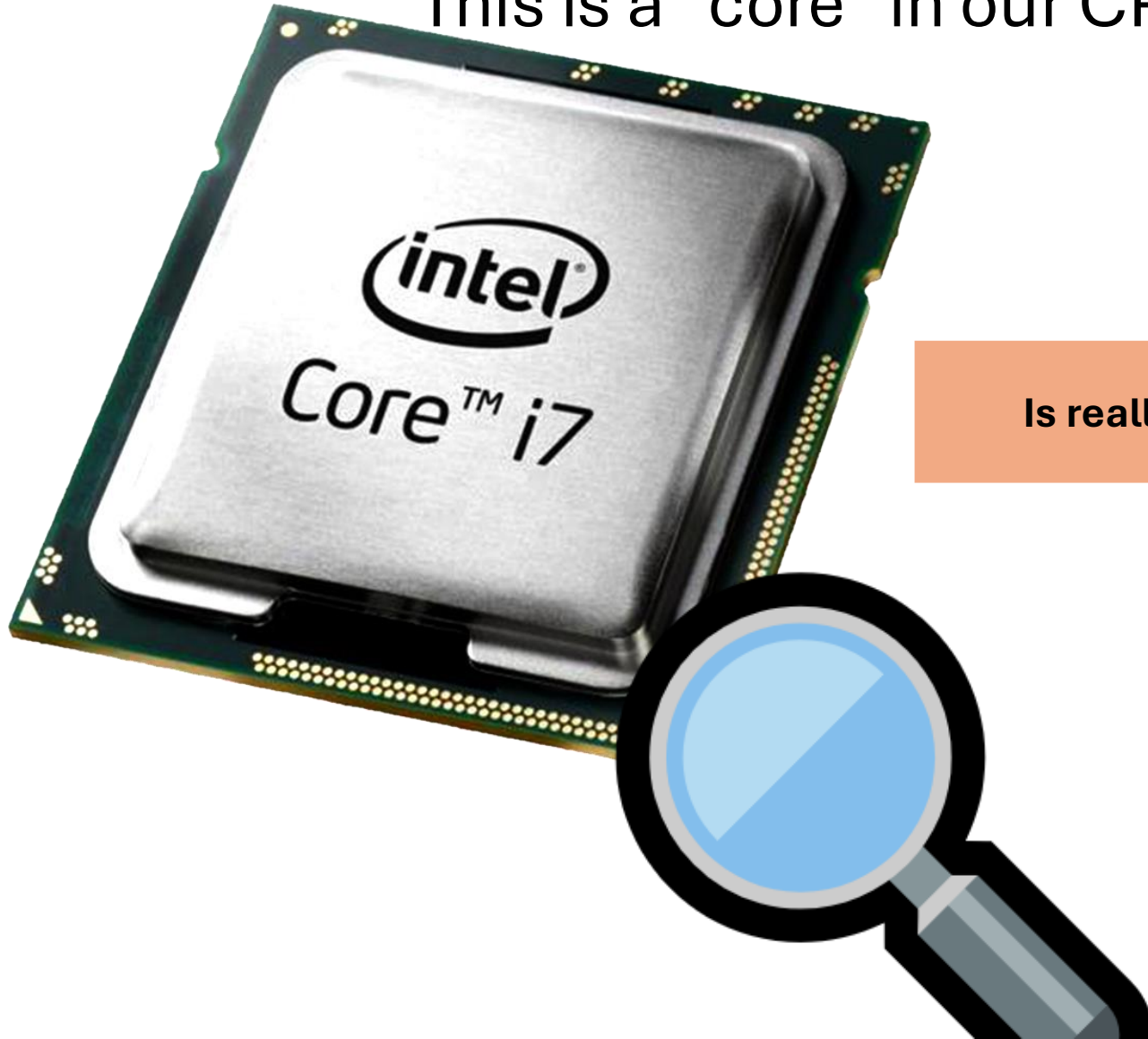




intel

Core™ i7

This is a “core” in our CPU that can run a thread



Each animal is a thread

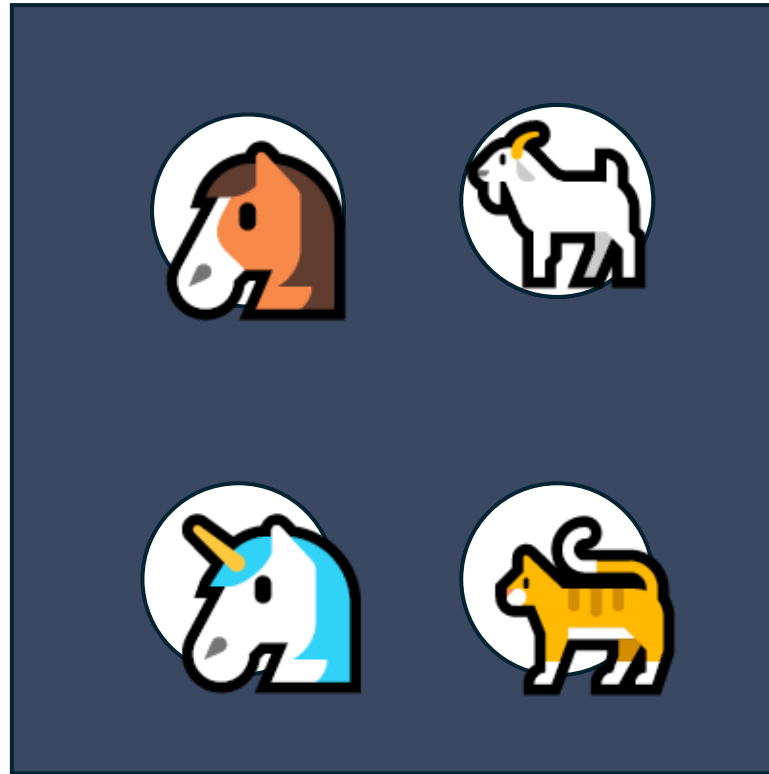
Context Switching



Context Switching



Dispatcher



Scheduler vs Dispatcher

A scheduler

- The part of the OS that decides what thread to run next
- It is different from a dispatcher which performs the context switch, which is all we've talked about up until now.

What are the reasons that a thread leaves a core?

1. The thread is done running
2. The thread “traps” into the OS
 - If a thread invokes something that only the OS can do, the OS *has* to step in
 - System Calls, Errors, or Page Faults
3. The thread is interrupted
 - Example: a timer, or a disk operation is completed
4. A thread yields to the core; i.e. the thread waits