CS111ACE Section 2

Threads, Processes, Scheduling!

Attendance Form -> https://tinyurl.com/CS111A-pollen
PolIEV (For anonymous Q+A) -> pollev.com/tripmaster419
Game Plan

- Icebreaker
- Welcome + Syllabus
- Threads + Processes
Game Plan

- Icebreaker:
  - Name
  - Pronouns (if you’d like!)
  - Hometown
  - Name of the person to the left of you
  - Favorite ___
Who am I?

- 2nd year Coterm in CS (Systems)
- CS undergrad (Computer Engineering)
- Section Leader, Head TA (CS106B), Coordinator (CS198)
- I LOVE teaching!

Some stuff I used to think about!

This is Buster!
Syllabus Time!

Record now 😊
Classroom Norms

- We’re all here to learn! Please be respectful of each other.
- There are no bad questions! This material is difficult, and I’m always happy to clarify / rewind
- Please refrain from using jargon (defined here as overly-technical language that is unrelated to what we’re learning) in class.
  - Don’t be *that person*
2 other ways to think about a process…

(spoiler alert: neither of these things is actually a process)
2 ways to think about a process…

Processes are *kind of like* programs.
2 ways to think about a process…

Processes are *kind of* like programs.
2 ways to think about a process…

Processes are *kind of* like programs. They’re independent, fairly large pieces of software (like big standalone programs)

**Key Idea:** Processes have lots of features, and they are independent of one another.
2 ways to think about a process…

Another way to think about a process on your computer is like a sandbox. Each one coexists on your system without disrupting other sandbox environments!
2 ways to think about a process…

The sandbox, or execution state, includes things like a private address space and a private table that interacts with the file system!
2 ways to think about a process…

The sandbox, or execution state, includes things like a private address space and a private table that interacts with the file system!

What does it mean to have a private address space???
2 ways to think about a process…

The **sandbox**, or execution state, includes things like a **private address space** and a **private table** that interacts with the file system!

What does it mean to have a private address space???
Fork() is a system call that clones the current process. The parent and child processes resume concurrently from the line of code following the fork() call.

```c
pid_t pidOrZero = fork();
```
Fork()

- Fork is a **system call** that clones the current process. The parent and child processes resume **concurrently** from the line of code following the fork() call.
- Fork takes in no arguments. Woo!!

```c
pid_t pidOrZero = fork();
```
Fork()

- Fork is a **system call** that clones the current process. The parent and child processes resume **concurrently** from the line of code following the fork() call.
- Fork takes in no arguments. Woo!!
- Fork returns **different things to the parent and to the child.**
  - The parent gets the pid of the child
  - The child receives the value 0.

```c
pid_t pidOrZero = fork();
```

- `pid` = Process ID (unique identifier)
Whenever a process forks a child process, the parent must clean up after the child once it terminates.

```c
pid_t waitpid(pid_t pid, int *status, int options);
```
Whenever a process *forks* a child process, the parent *must* clean up after the child once it terminates.

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

*waitpid* returns the pid of the child it *reaped* (cleaned up). It returns -1 if there was some error (usually when all children have been *reaped*)
Whenever a process forks a child process, the parent must clean up after the child once it terminates.

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

- `waitpid` returns the pid of the child it reaped (cleaned up). It returns -1 if there was some error (usually when all children have been reaped).
- The argument `pid` is the pid of the child the parent would like to reap. You can also pass in -1 if you simply want to reap any child. (works well with the -1 return value if you want to waitpid until all children are reaped.)
Whenever a process forks a child process, the parent must clean up after the child once it terminates.

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

- `waitpid` returns the pid of the child it reaped (cleaned up). It returns -1 if there was some error (usually when all children have been reaped).
- The argument `pid` is the pid of the child the parent would like to reap. You can also pass in -1 if you simply want to reap any child. (works well with the -1 return value if you want to `waitpid` until all children are reaped.
- We won’t really use `options` this quarter. The `status` pointer is populated after a `waitpid()` returns, and it indicates how the child terminated (i.e. did it crash)
Code Review 1

- Take a look at the following program. What does it print?

```c
int main (int argc, char *argv[]) {
    int pid = fork();
    int status;
    printf ("Hello from process %s\n", (pid != 0)? "parent" : "child");
    if (pid > 0) waitpid(pid, &status, 0);
    return 0;
}
```
Code Review 1

- Take a look at the following program. What does it print?

This is called a ternary operator. This entire expression returns either string on the right hand side, depending on the conditional in parentheses. It’s equivalent to:

```c
if (pid != 0) return "parent";
else return "child";
```
Code Review 1 (Think Pair Share)

- Take a look at the following program. What does it print?
  - What order do the “Hello” statements print in?
  - What happens if I remove the penultimate line?
  - What happens if I remove the if check on the penultimate line?

This is called a **ternary operator**. This entire expression **returns** either string on the right hand side, depending on the conditional in parentheses. It’s equivalent to:

```c
if (pid != 0) return "parent";
else return "child";
```

```c
int main (int argc, char *argv[]) {
    int pid = fork();
    int status;
    printf ("Hello from process %s\n", (pid != 0)? "parent" : "child");
    if (pid)  waitpid(pid, &status, 0);
    return 0;
}
```

*waitpid()* is similar to the **delete** or **free** keywords that you used with heap memory.
Announcements

- We’ll have Office Hours Thursdays from 9-11 AM
  - Let’s see if we can do it in Old Union on the second floor (near the FLI office). There is an elevator entrance at the side of the building!
  - OH will be in-person to maximize group discussion
    - OH is available only to students enrolled for units (ACE Policy)
- Make sure you’ve joined the Slack if you haven’t already!
  - It’s a great way to have informal discussions, and it’s the easiest way to get in contact with me!!
2 ways to think about a process...

What did Trip mean when he said that processes have **private address spaces**???

Small Code Demo:  
fork-copy.c

(courtesy of Nick T.)
The Process Analogy

- The space for our user programs can be thought of as a playground of many sandboxes. These sandboxes are isolated from each other.
As we said before, each sandbox is an “execution context” for a user program.

A sandbox itself is not enough!
The Process Analogy

- As we said before, each sandbox is an “execution context” for a user program.
- A sandbox itself is not enough!
The Process Analogy

- The more complete view of the process is the sandbox (execution context), as well as the child (or children) playing in the sandbox.
- You should think of the child as the entity that “gets stuff done” within the execution context.
The Process Analogy

- What is the child?
Threads

- Like a “child” in the “sandbox,” a thread is the actual entity that *runs* your code.
Threads

- Like a “child” in the “sandbox,” a thread is the actual entity that *runs* your code.
- When a process is created, the OS spins up a single *thread* to run the code within the context of the process.
Threads

- Like a “child” in the “sandbox,” a thread is the actual entity that runs your code.
- When a process is created, the OS spins up a single thread to run the code within the context of the process.
- Threads are designed for parallelizing workflows. This is because threads don’t have private address spaces!
Processes vs Threads

**Process**
- Private address spaces
- Good for "dispatching" stuff that might get destroyed (fork-exec)

**Thread**
- Share the address space of their "container" process
- Good for sharing work (parallelizing workflow)

```c
#define foo(x) x

void func()
{
  int man() {
    thread t = thread(func);  
    t.join();
  }
}
```
Scheduling

- This is what your **Central Processing Unit (CPU)** looks like:
Scheduling

- This is what your **Central Processing Unit** (CPU) looks like:

![Diagram of CPU with blue lightning bolts representing threads and black circles representing "cores"]
Scheduling

- This is what your **Central Processing Unit** (CPU) looks like:
- (How many threads are running *in this instant*)?

Blue lightning bolts represent threads.

These larger black circles represent "cores"
Scheduling

- As you can see, the OS is responsible for moving threads on and off the CPU very quickly!

Blue lightning bolts represent threads.

These larger black circles represent “cores”
Scheduling

- As you can see, the OS is responsible for moving threads *on and off* the CPU very quickly!

**Key Terminology:**

**Scheduler:** Code that figures out which thread(s) to run next

**Dispatcher:** Code that actually performs the thread swapping (on and off cores)

*Important: These are *not* threads. Your thread may enter these areas by calling special functions*
Scheduling

- As you can see, the OS is responsible for moving threads on and off the CPU very quickly!
- Under what circumstances does the dispatcher get invoked (which switches a thread off of the CPU?)

Blue lightning bolts represent threads.

These larger black circles represent “cores”
Dispatching

The dispatcher may get invoked for the following reasons:
Dispatching

The dispatcher may get invoked for the following reasons:

1. The thread has finished its execution (func from thread(func) has completed)
Dispatching

The dispatcher may get invoked for the following reasons:

1. The thread has finished its execution (*func* from `thread(func)` has completed)
2. The thread can “Trap” into the OS
   a. Any time a thread does something that requires OS help (unknowingly), the OS will take over to help out.
Dispatching

The dispatcher may get invoked for the following reasons:

1. The thread has finished its execution (\texttt{func} from \texttt{thread(func)} has completed)

2. The thread can “Trap” into the OS
   a. Any time a thread does something that requires OS help (unknowingly), the OS will take over to help out.
      i. Errors (seg fault)
      ii. System calls (read, write, fork, etc.)
      iii. Page Faults (we’ll learn about these later)
Dispatching

The dispatcher may get invoked for the following reasons:

1. The thread has finished its execution (\texttt{func} from \texttt{thread(func)} has completed)
2. The thread can “Trap” into the OS
3. The thread can \textit{yield} the core willingly (when??)
Dispatching

The dispatcher may get invoked for the following reasons:

1. The thread has finished its execution (**func** from \( \text{thread}(\text{func}) \) has completed)
2. The thread can “Trap” into the OS
3. The thread can \textit{yield} the core willingly (**waiting**)
4. The thread can be \textit{interrupted} by an \textbf{outside event}.
   a. Some important I/O arrives (like mouse click) / \textbf{OS replaces thread}
Dispatching

The dispatcher may get invoked for the following reasons:

1. The thread has finished its execution (`func` from `thread(func)` has completed)
2. The thread can “Trap” into the OS
3. The thread can `yield` the core willingly (`waiting`)
4. The thread can be `interrupted` by an `outside event`.
   a. Some important I/O arrives (like mouse click) / `OS replaces thread`
   b. Some scheduling algorithms use a timer to preempt + switch threads every time the timer fires.

In order to ensure `fairness` and prevent `starvation`, the OS tries to schedule threads as evenly as possible. This usually involves the OS needing to remove running threads from the CPU to give other threads CPU time.
Dispatching

The dispatcher may get invoked for the following reasons:

1. The thread has finished its execution *(func from thread(func) has completed)*
2. The thread can “Trap” into the OS
3. The thread can *yield* the core willingly *(waiting)*
4. The thread can be *interrupted* by an outside event.
   a. Some important I/O arrives (like mouse click) / OS replaces thread
   b. Some scheduling algorithms use a timer to preempt + switch threads every time the timer fires.

In order to ensure *fairness* and prevent *starvation*, the OS tries to schedule threads as evenly as possible. This usually involves the OS needing to remove running threads from the CPU to give other threads CPU time.
Dispatching

All this is to say, there are many reasons why threads will switch on and off the processor cores!

In practice, this means that we can never assume the exact order in which our threads will run!

Thread 1
printf ("Let's have")

Thread 2
printf ("a great quarter")
Dispatching

All this is to say, there are many reasons why threads will switch on and off the processor cores!

In practice, this means that we can never assume the exact order in which our threads will run!

Thread 1

printf ("Let’s have")

Thread 2

printf ("a great quarter")

it hasn't been proven yet, but my theory is that Master Yoda’s dialogue was inspired by race conditions in CS111 student code...
Major Learning Takeaways

Here’s what I hope you have a stronger understanding of today!

- The difference between a thread and a process
- The difference between the scheduler and the dispatcher
- The intuition that threads are *constantly switching on and off the CPU*, which is why we cannot reason about the exact ordering of threads