How Programs Reproduce

12

CS/COE 0449
Introduction to Systems Software

wilkie

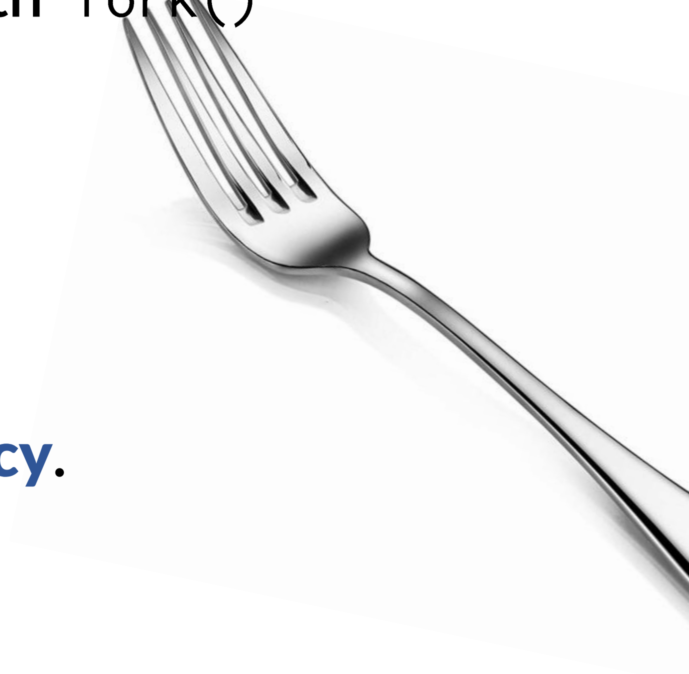
Spring 2019/2020

Creating Processes

Forks: what you stick in things that are done… and sometimes a system call.

This is a story about a system call…

- We are focusing several system calls starting with `fork()`

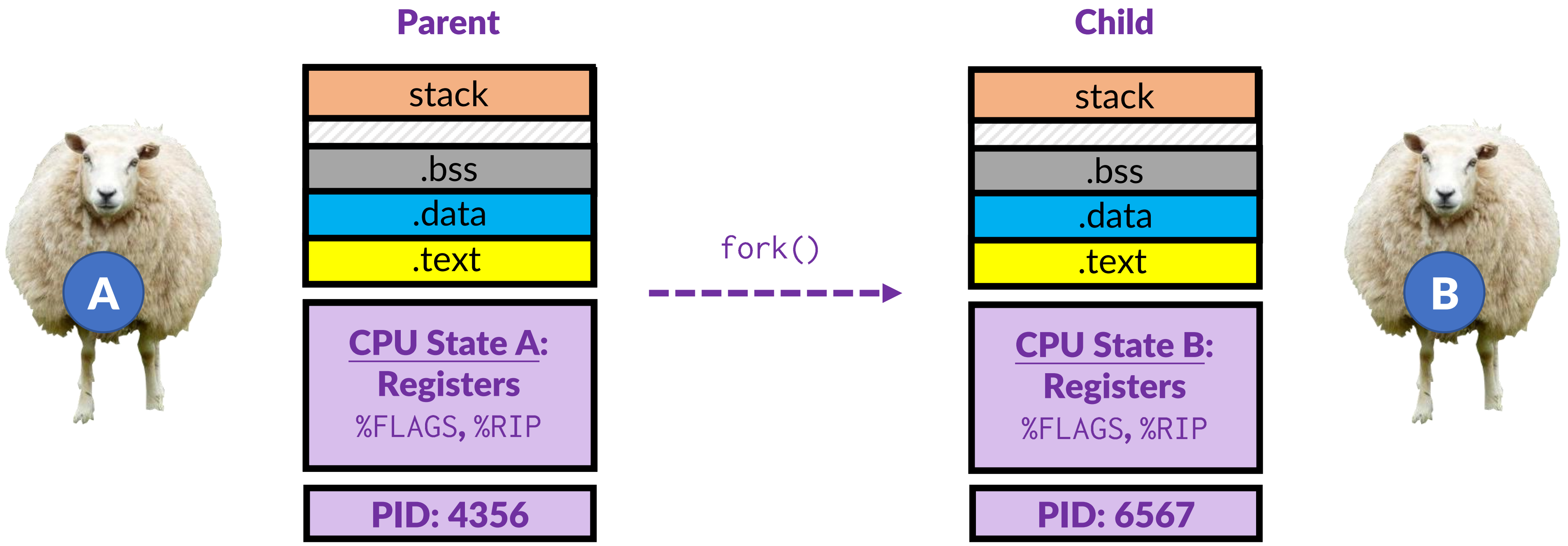


- This system call copies the current process.
  - This creates a "child" process that is a duplicate of the memory and state of its parent.
- This can be a convenient way to gain **concurrency**.
  - Copy the process and run each copy …
  - … those copies now run at the same time.
    - This is the origin of the term "fork" … a logical split in a program where there are now multiple paths.

  - We will see this idea in action soon.

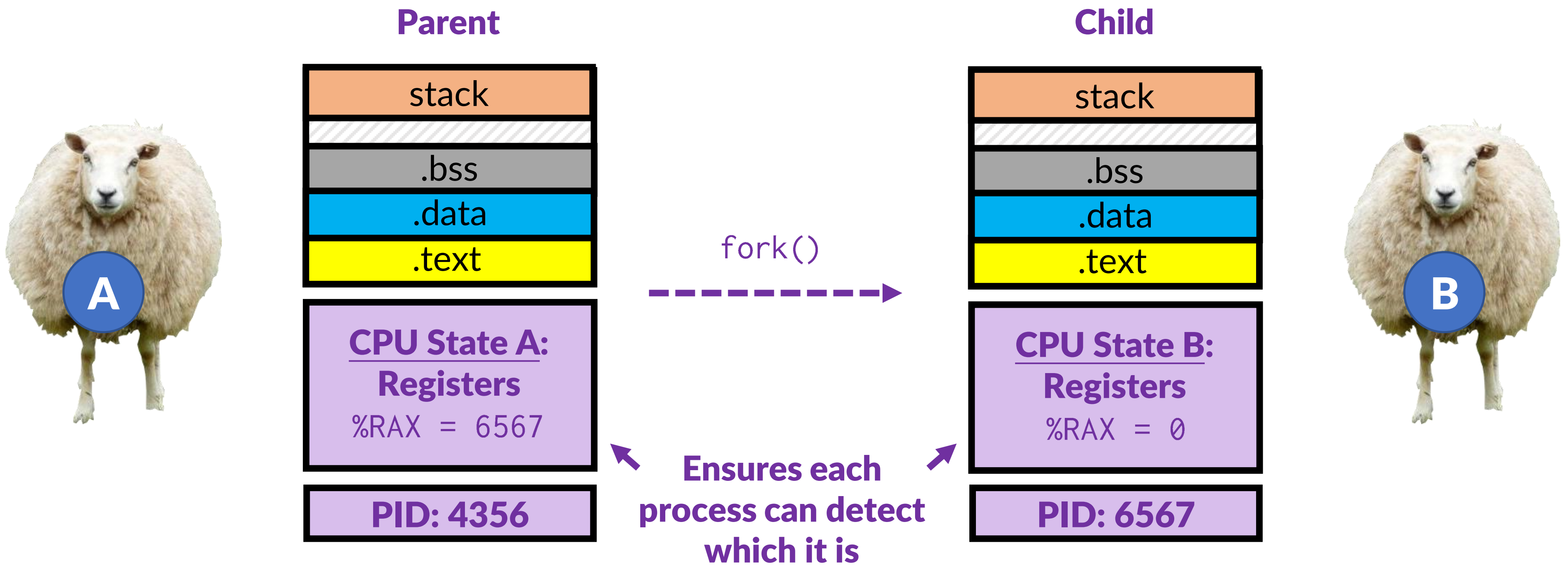
Here's Dolly

- The `fork()` system call in action:
  - Copies the memory layout.
  - Copies the process state. (but gives it a unique ID)



Here's Dolly's ID tag

- The `fork()` system call in action:
  - Updates the child's CPU state so that it returns `0`. (An invalid `pid`)
  - Updates parent's CPU state to return the child's process ID. (`pid`)



A small fork example... a... salad fork? example??

C

```c
#include <stdio.h>  // printf
#include <unistd.h> // pid_t

void spawn(void) {
  int x = 0;
  pid_t pid = fork();
  if (pid == 0) {
    x--;
    printf("child! %d\n", x);
  }
  else {
    x++;
    printf("parent! %d\n", x);
  }
}

int main(void) {
  spawn();
  return 0;
}
```

The x is copied, so it has different values in child and parent.

```
> ./fork-example
parent! 1
child! -1
```

- **There is only one process when `spawn()` is called.**
- **However, when `fork()` is called, the system call returns "twice"**
  - Once in the parent process
  - Once in the child process
- **This starts two concurrent executions within the same program.**
  - Via two processes.
- **What does this print out?**

Children first... OR NOT

C

```c
#include <stdio.h>  // printf
#include <unistd.h> // pid_t
```

Child **Parent**

```c
void spawn(void) {
  pid_t pid = fork();
  if (pid == 0) {
    printf("child!\n");
  }
  else {
    printf("parent!\n");
  }
}

int main(void) {
  spawn();
  return 0;
}
```

```
> ./fork-example
child!
parent!
> ./fork-example
parent!
child!
```

- If the child process goes first...
  - Then it will print the child text.
- Then the scheduler schedules the parent process once more.
  - Then it will print the parent text.
- However, that's not the only possible pattern.
- If the parent process goes first...
  - Then it prints the parent text ...
  - ... followed by the child.

Two roads diverged in a yellow wood, AND I TOOK BOTH (NOT SORRY)

C

```c
#include <stdio.h>  // printf
#include <unistd.h> // pid_t

void spawn(void) {
  pid_t pid = fork();
  while (1) {
    if (pid == 0) {
      printf("child!\n");
    }
    else {
      printf("parent!\n");
    }
  }
}

int main(void) {
  spawn();
  return 0;
}
```

- If I were to extend the code to make it loop infinitely…
  - The parent and child will constantly race to print out their respective text.

```
> ./fork-example
parent!
child!
parent!
parent!
child!
parent!
parent!
parent!
child!
child!
```

The good, the bad, and the unpredictable

- Adding *concurrency* to your program makes things… weird.
  - You cannot rely on the order processes will be scheduled.
  - Your operations will be **asynchronous** (not synchronized; no known order)
- If you need to synchronize processes, you can do so with `wait()`.
- `wait()` yields the process and returns only when a child process ends.
  - It returns when *any child process* exits.
  - Its return value is the pid of the child process that exited.
  - You can also use `waitpid(pid_t)` to specify a specific child process by its pid.

Waiting is such sweet sorrow… wait that's not right

```c
#include <stdio.h>  // printf
#include <unistd.h> // pid_t
#include <stdlib.h>  // exit
#include <sys/wait.h> // wait
```

Child **Parent**

```c
void spawn(void) {
  pid_t pid = fork();
  if (pid == 0) {
    printf("child!\n");
    exit(42);
  }
  else {
    pid_t exited_pid = 0;
    while (exited_pid != pid)
      exited_pid = wait(NULL);
    printf("parent!\n");
  }
}
int main(void) {
  spawn();
  return 0;
}
```

Always:

```
> ./fork-example
child!
parent!
```

- By using `wait()` the parent process only continues when the child process ends.
- Therefore, the output order is now known.
  - If the parent goes first…
  - It gets stuck at the `wait()` call.
  - Then the child goes until it hits `exit()`
  - `exit()` ends the process.
  - And then the parent continues.
- Nice and well-known behavior!

Notes on `exit()`

- **The `exit(int)` system call ends the current process.**
  - The given argument is the process return code also known as an **exit code**.
  - Normally your program yields an exit code at the end of `main()`
  - Exit ends your program exactly at the point of the call.
  - Therefore, it has its own means of giving the exit code.

- **However, we can have processes that are no longer running…**
  - Yet, not deallocated either.
  - The are not living…
  - And not dead!!

Zombies

- A terminated process still takes up space
  - All that process metadata sticks around
  - Until the parent tells the system it doesn't need it
- As long as the parent stays alive...
  - The corpse of the child process sticks around, too.
- These are called **zombie processes**.
  - They are processes that still exist and have an ID yet do not run and are no longer scheduled.

Dancing Zombie from Plants vs. Zombies
Copyright PopCap Games, a subsidiary of EA Games

The night of the living dead

C

```c
#include <stdio.h>  // printf
#include <unistd.h> // pid_t

void spawn(void) {
  pid_t pid = fork();
  if (pid == 0) {
    printf("child!\n");
  }
  else {
    printf("parent!\n");
    while(1) {} // Infinite Loop!
  }
}

int main(void) {
  spawn();
  return 0;
}
```

- If I added an infinite loop to the parent…
  - When the child ends…
  - And I list the active processes using the ps command.
  - I see a “defunct” child process. A ZOMBIE!

```
> ./fork-example
parent!
child!
```

```
> ps
  PID TTY          TIME CMD
 6569 pts/9   00:00:12 fork_example
 5435 pts/9   00:00:00 fork_example <defunct>
```

Just the normal kind of dead.

C

```
#include <stdio.h>  // printf
#include <unistd.h> // pid_t

void spawn(void) {
  pid_t pid = fork();
  if (pid == 0) {
    printf("child!\n");
    while(1) {} // Infinite Loop!
  }
  else {
    printf("parent!\n");
  }
}

int main(void) {
  spawn();
  return 0;
}
```

- However, if I added an infinite loop to the child…
  - When the parent ends… the program ends as well!
  - And I list the active processes using the `ps` command. I see only the child process

```
> ./fork-example
child!
parent!

> ps
  PID TTY          TIME CMD
 5435 pts/9    00:00:00 fork_example
```

No zombies here!

Just orphans…

How to run a different program?

- When you `fork()` a process, you are making an exact copy of that process.

- However, maybe you want to create a process to run a different program altogether?
  - This is very useful… instead of using a software library
  - You could just run the existing program.

- For this purpose, the `exec*()` family of system calls is used.
  - There are several different variations of exec calls…

Invoking the OS loader...

C

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

void main(void) {
  pid_t pid = fork();
  char* argv[] = {"/usr/bin/ls", "-a", NULL};
  if (pid == 0) { // Child
    printf("Child is calling exec!\n");
    execv("/usr/bin/ls", argv);
  }
  else {
    printf("Parent is waiting...\n");
    wait(NULL);
    printf("Done!\n");
  }
}
```

- Using the `execv()` system call.
  - The call takes the path to an executable
  - And an array of strings for the arguments.
    - Sentinel: must end in a `NULL`
  - The first argument to a C program is always its own path!

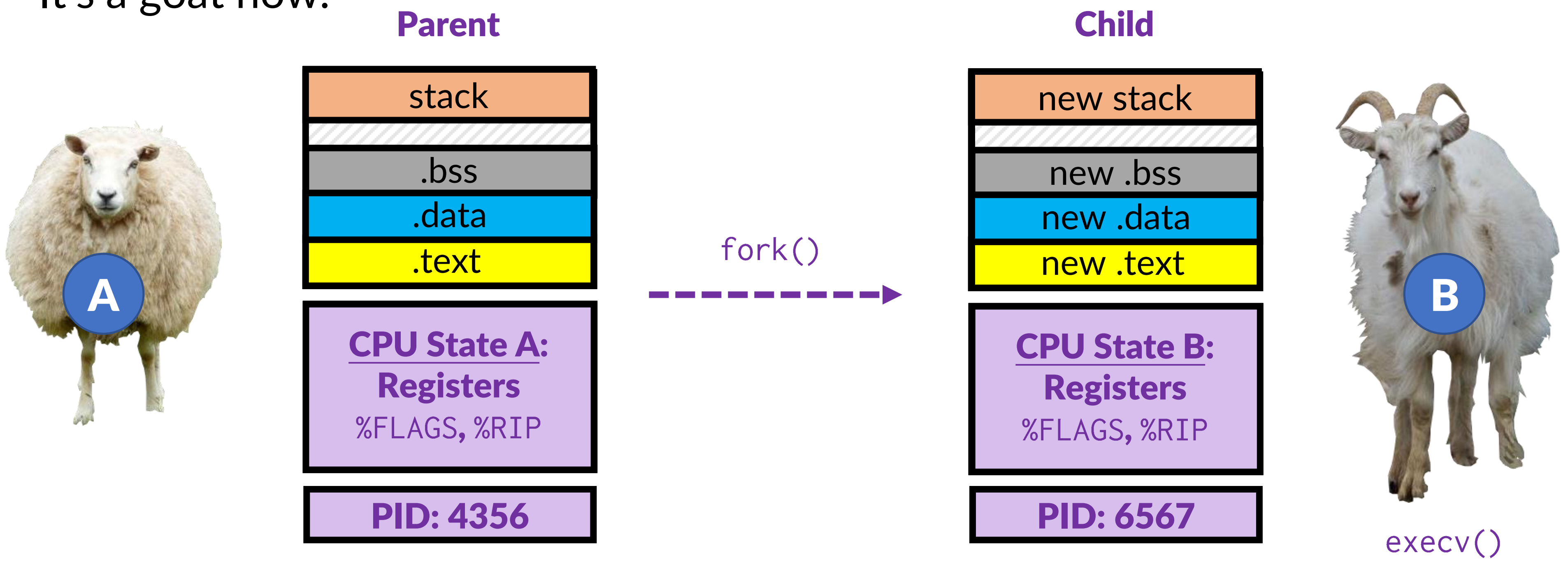
```
> ./fork-exec-example
Parent is waiting!
Child is calling exec!
.       fork-exec-example.c
..      fork-exec-example
Done!
```

We ran "`ls -a`"

Add then continued our own process.

Here's Dolly's brother Bobby. Bobby is a goat somehow. Don't ask.

- The `fork()` system call in action:
  - Copies the memory layout. Copies the process state. (but gives it a unique ID)
- The `execv()` system call in action:
  - It's a goat now.



Different forms of `exec`

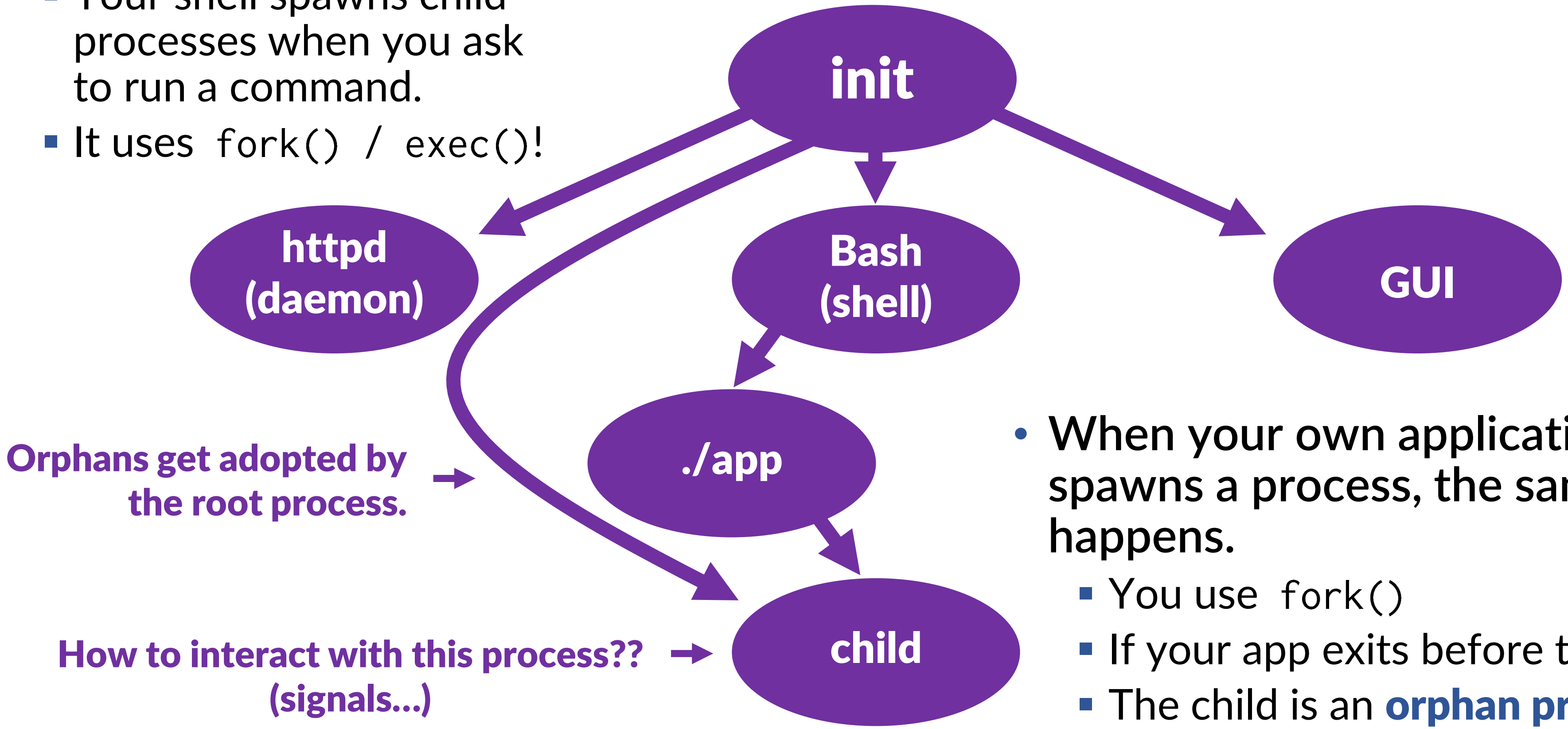
- You can look up the many different styles of exec
  - Each one has a different way of calling it.

- `execv` called with an array of strings terminated with a `NULL`
- `execve` same, but can use specific environment variables
- `execvp` searches the system paths for the executable
- `execvpe` combination of `execve` and `execvp`

- There are also `execl*` functions that use function arguments instead of an array of strings.
  - `execlp("ls", "ls", "-a", NULL);`

The common ancestor... and the orphan.

- UNIX/Linux has an interesting design: every application is a child process.
  - The root is the `init` task.
  - Your shell spawns child processes when you ask to run a command.
  - It uses `fork() / exec()`!



- When your own application spawns a process, the same thing happens.
  - You use `fork()`
  - If your app exits before the child...
  - The child is an **orphan process**.

An extreme attitude

- How do we interact with orphaned processes?

- How do we synchronize at a finer granularity?
  - Using `wait()` is rather inflexible.
  - It can only detect that a child process ends using `exit()` or via main

- What if you want to synchronize smaller events…
  - The child process does something… The parent responds…
  - But, keep the child process running longer.

- For this, we will need the parent and child to be able to communicate with one another.

Inter-Process Communication

IPC … not to be confused with ICP

What that last slide said...

- Passing data or messages from one process to another is called **inter-process communication**.

- This is a broad OS topic as there are many ways to do this.
  - Shared memory (we will talk about this a bit later)
  - Message passing (we will talk about this NOW)
    - Simple messages (signals, this lecture)
    - More complex (pipes, semaphores, etc, soon)
    - Most complex (network sockets, we will look at this later)

- Message passing is a fancy way of saying are using an API to send a small message to another process.
  - And also some means of listening for messages.

All aboard the train metaphor

- In UNIX/Linux, tiny messages sent between processes are called **signals**.

- They are typically used to send messages about events from the system. Here are a few:

Number	Name	Description	Default Behavior
2	`SIGINT`	Interruption – Somebody pressed CTRL+C	Terminate
9	`SIGKILL`	Kill – Somebody wants us gone… ☹	Terminate
11	`SIGSEGV`	Memory Violation – Oops! Seg-fault	Terminate
17	`SIGCHLD`	Child exited – Child process ended	Ignore
10	`SIGUSR1`	A signal that you can use for any purpose	Ignore

Talking to orphans

C

```c
#include <stdio.h>  // printf
#include <unistd.h> // pid_t

void spawn(void) {
  pid_t pid = fork();
  if (pid == 0) {
    printf("child!\n");
    while(1) {} // Infinite Loop!
  }
  else {
    printf("parent!\n");
  }
}

int main(void) {
  spawn();
  return 0;
}
```

- Recall the infinite looping child.
- Orphans run in the background.
- However, we can send a `SIGKILL` message (`9`) to the process by its id.

```
> ./fork-example
child!
parent!
> ps
  PID TTY          TIME CMD
 5435 pts/9    00:00:00 fork_example
> kill -9 5435
> ps
 PID TTY          TIME CMD
>
```

The parent ended.

But not the child.

We can send a `SIGKILL` message using the `kill` application..

And the child is gone!

Receiving Signals

C

```
#include <stdio.h>  // for printf
#include <unistd.h> // for pid_t
#include <signal.h> // for signal

void sigint_handler(int signum) {
}

int main(void) {
  signal(SIGINT, sigint_handler);
  while(1) {}
  return 0;
}
```

- **The `signal()` standard function will set up your application to listen for a particular signal.**
- **This example hooks the empty function sigint_handler to override the default behavior of the SIGINT signal.**
- **If you recall, that happens on a CTRL+C… which now does not terminate the foreground process!**
  - Needs to be killed using `SIGKILL` now.

Waiting for a signal...

```c
#include <stdio.h>  // for printf
#include <unistd.h> // for pid_t
#include <signal.h> // for signal

static int goods_count = 0;
static int wait_counter = 0;

void signal_handler(int signum) {
  wait_counter = 0;
}

void main(void) {
  signal(SIGUSR1, signal_handler);
  pid_t pid = fork();

  wait_counter = 1;
  while(goods_count < 5) {
    if (pid == 0) { // Child: Consumes data
      while(wait_counter == 1) {}
      printf("Consumed data!\n\n");
      wait_counter = 1;
      kill(getppid(), SIGUSR1);
    }
    else { // Parent: Produces data
      printf("Produced data!\n");
      wait_counter = 1;
      kill(pid, SIGUSR1);
      while(wait_counter == 1) {}
    }
    goods_count++;
  }
}
```

Both processes set `wait_counter` **to 0 on** `SIGUSR1`**.**

1. `wait_counter` **is initially** `1`

2. Which causes the child to wait...

3. Until the parent process signals it, after it prints its message.

4. Afterward, the parent process waits

5. Until child signals it back after printing its own message.

6. Repeat... for both

- Proper use of signals and waiting on the values of variables to change can create **synchronization**.

```
> ./signal-sync-example
Produced data!
Consumed data!

Produced data!
Consumed data!

Produced data!
Consumed data!

Produced data!
Consumed data!

Produced data!
Consumed data!
```

Let's look at that again. (animated)

```
#include <stdio.h>  // for printf
#include <unistd.h> // for pid_t
#include <signal.h> // for signal

static int goods_count = 0;
static int wait_counter = 0;

void signal_handler(int signum) {
  wait_counter = 0;
}
```

Child 0 **Parent 0**

```
void main(void) {
  signal(SIGUSR1, signal_handler);
  pid_t pid = fork();

  wait_counter = 1;
  while(goods_count < 5) {
    if (pid == 0) { // Child: Consumes data
→     while(wait_counter == 1) {}
      printf("Consumed data!\n\n");
      wait_counter = 1;
      kill(getppid(), SIGUSR1);
    }
    else { // Parent: Produces data
→     printf("Produced data!\n");
      wait_counter = 1;
      kill(pid, SIGUSR1);
      while(wait_counter == 1) {}
    }
    goods_count++;
  }
}
```

1. Child waits
2. Parent prints
   1. Sets its own wait variable
   2. Sends signal to child
   3. Waits
3. Child prints
   1. Sets its wait variable
   2. Sends signal to parent
   3. Waits
4. Parent prints
   1. Sets its own wait variable
   2. Sends signal to child
   3. Waits
5. Repeat...

```
> ./signal-sync-example
Produced data!
Consumed data!

Produced data!
Consumed data!

Produced data!
Consumed data!

Produced data!
Consumed data!

Produced data!
Consumed data!
```

If you are in a hurry... (animated)

```c
#include <stdio.h>  // for printf
#include <unistd.h> // for pid_t
#include <signal.h> // for signal

static int goods_count = 0;
static int wait_counter = 0;

void signal_handler(int signum) {
  wait_counter = 0;
}

void main(void) {
  signal(SIGUSR1, signal_handler);
  pid_t pid = fork();

  wait_counter = 1;
  while(goods_count < 5) {
    if (pid == 0) { // Child: Consumes data
      while(wait_counter == 1) {}
      printf("Consumed data!\n\n");
      wait_counter = 1;
      kill(getppid(), SIGUSR1);
    }
    else { // Parent: Produces data
      printf("Produced data!\n");
      wait_counter = 1;
      kill(pid, SIGUSR1);
      while(wait_counter == 1) {}
    }
    goods_count++;
  }
}
```

Child 0 **Parent 0**

OOPS

1. Child waits
2. Parent prints
   1. Sets its own wait variable
   2. Sends signal to child
   3. Waits
3. Child prints
   1. Sends signal to parent
   2. Sets its wait variable
   3. Waits
4. Parent prints
   1. Sets its own wait variable
   2. Sends signal to child
   3. Waits
5. OH NO!!!

OH NO!

oh no

Let's Mess Things Up!!

```
> ./signal-sync-example
Produced data!
Consumed data!

Produced data!
```

The race is on!

- When you have concurrent tasks, they may compete.

- A bug in a concurrent program where the logic breaks if one process out-paces another is called a **race condition**.
  - That is, if the correctness requires a strict order, but that order is not guaranteed.

- When you add *synchronization* you need to be careful that you ensure that each synchronized section (called a **critical section**) is logically sound.

```
while(wait_counter == 1) {}    // START (wait)
printf("Consumed data!\n\n");  // do work
wait_counter = 1;              // prepare to wait
kill(getppid(), SIGUSR1);      // END (signal)
```

← **We know we won't be interrupted between the while loop and the signal. (This is our critical section)**

Summary

- **Today we learned the birds and bees of programs.**
  - They start as processes (technically children of a shell or some root process)
  - They can spawn child processes using `fork()`
  - They can load executables over top of them using `exec*()` system calls
  - And if one process ends before the other, we either get zombies or orphans.

- **We also learned about inter-process communication in the form of signals.**
  - These are tiny messages sent using the `kill()` function; received via `signal()`.
  - We can use them to synchronize events between processes.
  - However, if we aren't careful, we may introduce a bug called a race condition.
  - This is when the program requires a logical order it cannot guarantee.