Lecture 06 - Program Execution

ENSF461 - Applied Operating Systems

Instructor: Lorenzo De Carli, University of Calgary (*lorenzo.decarli@ucalgary.ca*)
Slides by Lorenzo De Carli, partly based on material by Robert Walls (WPI)

Review of previous lecture

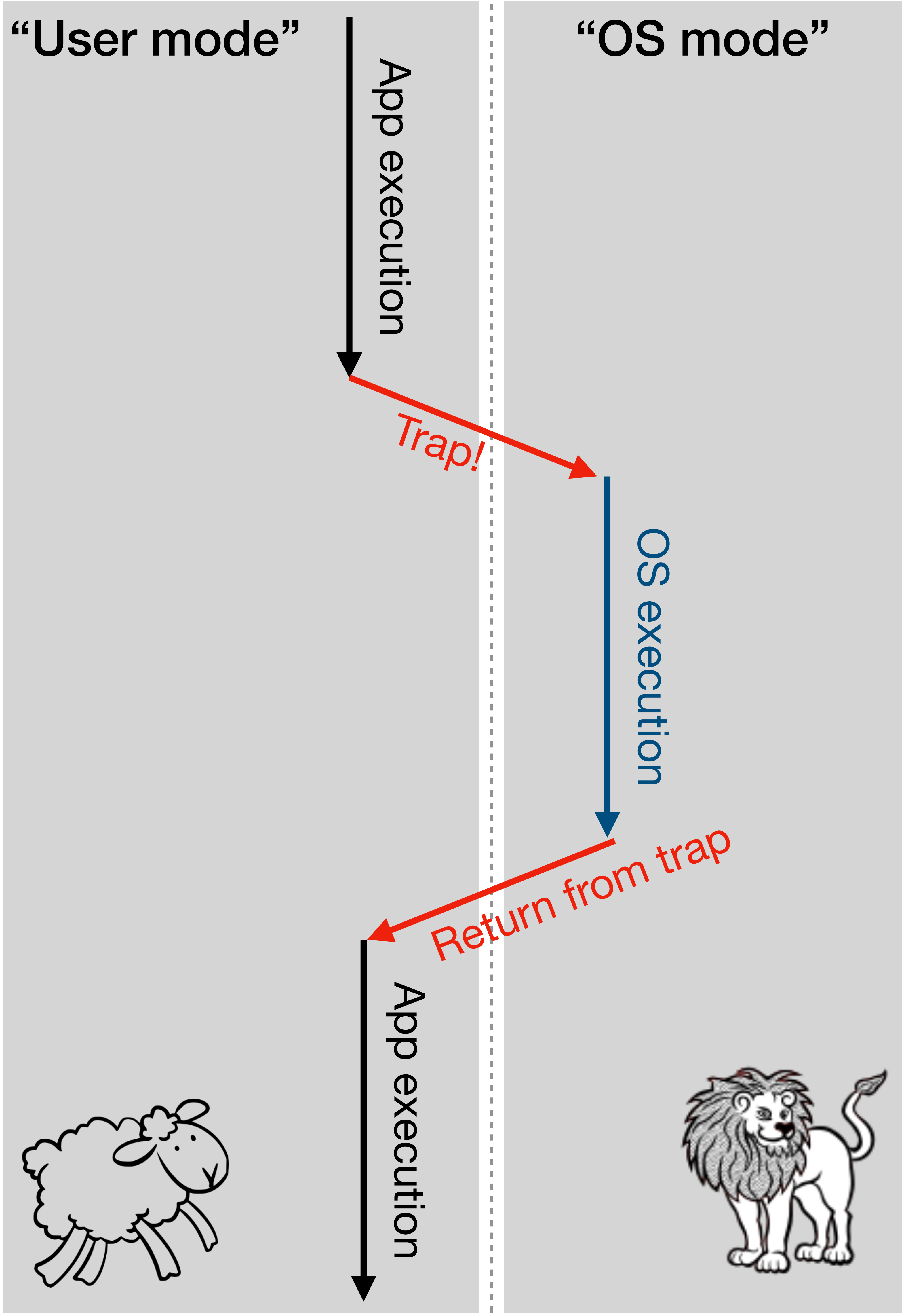
Separation

- Typically, the CPU architecture defines some type of **trap** - an instruction to switch into OS code, and back

- **To be clear:** this allows a process to “jump” into and continue execution into OS code

  - **It does not** allow a process to run its own code as OS

  - The app can pass **execution and parameters to OS**, but cannot define the code that the OS is going to execute



CPU virtualization

…or the art of “slicing” a CPU

- An OS should offer the ability to:
  - **Run** a specific program
  - **Stop** a program
- It should also:
  - Decide **which program should run** at any given time
  - **Divide CPU time** across programs “**fairly**”

Memory virtualization

What is this sorcery?

- **Virtualizing memory** is actually (suprisingly?) difficult
  - Arguably more than virtualizing CPU
- Memory virtualization give every process the same **virtual address space**
  - In other words, each process gets the illusion of having memory to itself (the **size** of the virtual address space depends on **OS/architecture**)
- The virtual address space is somehow **mapped** to the **physical space**

What is concurrency?

And how is it different from sharing CPU?

- **CPU virtualization** just means that multiple independent processes can share the CPU
- In many cases, you want **different parts** of your program **cooperate** to **complete a task**
    - The ability for different "parts" of the program to **communicate**
    - The ability to **synchronize** access to **shared data**

Quiz time!

D2L-> ENSF461 -> Assessment -> Quizzes -> Quiz 06

The process

The process

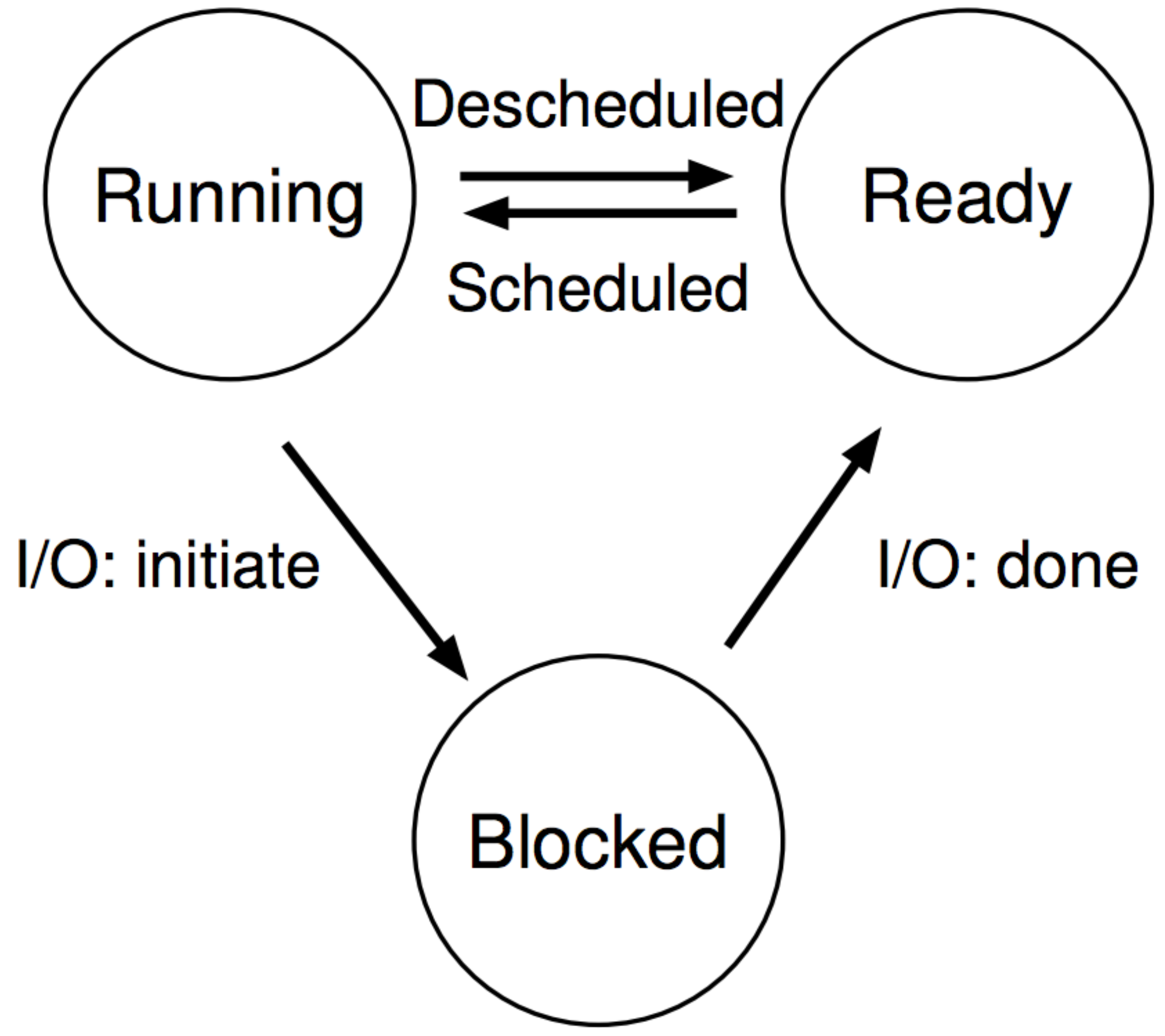
- The **process** is most fundamental abstraction of the operating system. Represents a single task the computer is doing.
- Unlike other abstractions, processes are **not tied** to a **specific hardware component**.
- **What hardware does the concept of a process help virtualize?**
- **Answer:** The CPU.
- What are the **elements** of a **process**?
  - one or more **threads**
  - an **address space** containing **instructions** and runtime **data structures**
  - zero or more open **file handles**

Process states (simplified)

- A process can be in one of three **execution states**:
- **running**: one of the process threads is currently executing on the CPU
- **ready**: the process is ready to run, but has yet to be scheduled by the CPU
- **blocked**: the process is waiting for something, e.g., a disk read, before it can continue

Scheduling

- Moving between running/ready is done by the OS. This is called **scheduling**.
- The underlying mechanism of saving (and restoring) the state of a process is called **context switching.**
- Issues: How does the OS decide which process to run next? What metrics might it try to optimize? The answers are defined by **policies**.
- More about scheduling and policies later. For now, let's look at the **details of processes.**



Managing processes (from the user's perspective)

- Users need to be able to **manage the processes** on their system.
- Focus on **UNIX-based operating systems**, though other OSes will have similar concepts.
- **How do we create a process in UNIX?**
  - One option: **use a shell**.

Before we move on

The process tree 🌳

> UNIX-like OS'es organize processes in **trees**

> Each process has a **parent** (process that created it) and can have 0 or more **children**

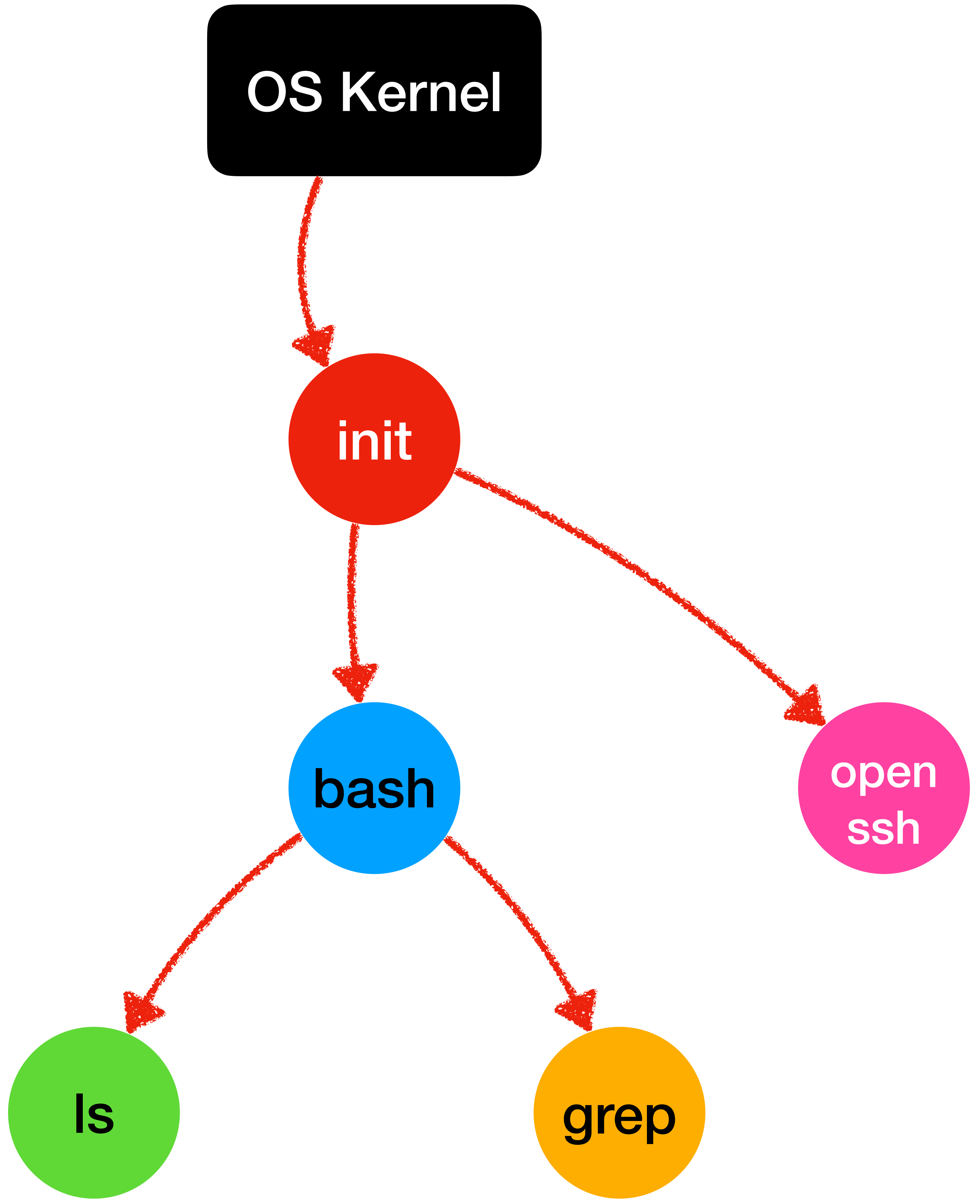
> When the parent terminates, **all children terminate**

> If a child terminates, the parent **continues running**

> Who starts the first process? Ad-hoc program executed by the kernel (**`init`** in **Linux**)

Example

- What happens if I **boot up the VM…**
- …wait for the **shell** to show up…
- and enter "**`ls -1 | grep .c`**"?



Each process has unique numerical identifier, called the PID (Process IDentifier)

One more thing

System calls, or sycalls

- We have seen a **process** can call **OS services** using a **trap**
- These “OS services” are called **system calls** (**syscalls** for friends)
- In practice system calls are **never issued manually**
- The C library (glibc) has **wrappers** that take care of issuing **syscalls**
- E.g.: to get the **PID** of a running process, you use **`getpid()`**
- To keep things simple, oftentimes we refer to **`getpid()`** (and similar) as system calls, although the actual system call gets issued inside **`getpid()`**

Creating our own shell

The shell

- What’s going on underneath it all? Could we **write our own shell**? Yes!

A simple shell

```c
#define TRUE 1

while (TRUE) {








}
```

The prompt

lorenzo — -bash — 61×18

```
Last login: Mon Sep 18 08:18:38 on console
[lorenzo@odin ~]$ echo "Meow" > milo.txt
[lorenzo@odin ~]$ cat milo.txt
Meow
[lorenzo@odin ~]$ sleep 5 &
[1] 4432
[lorenzo@odin ~]$ ps
  PID TTY           TIME CMD
 4417 ttys000    0:00.03 -bash
 4432 ttys000    0:00.01 sleep 5
[lorenzo@odin ~]$ whereis sleep
sleep: /bin/sleep /usr/share/man/man1/sleep.1
[1]+  Done                    sleep 5
[lorenzo@odin ~]$ 
```

a simple shell

```
#define TRUE 1

while (TRUE) {
  type_prompt();







}
```

a simple shell

```
#define TRUE 1

while (TRUE) {
  type_prompt();
  read_command(command, parameters);






}
```

a simple shell

```c
#define TRUE 1

while (TRUE) {
  type_prompt();
  read_command(command, parameters);


  /* Execute the command? */



}
```

How to draw an owl

1.

2.

1. Draw some circles

2. Draw the rest of the owl

Exec()

- **`exec()`** is a **system call** that starts a new process by **transforming** the current process.

A simple shell

```
#define TRUE 1

while (TRUE) {
  type_prompt();
  read_command(command, parameters);




    execve(???);

}
```

lorenzo — -bash — 81×33

```
[lorenzo@odin ~]$ man execve
```

lorenzo — less ◂ man execve — 81×33

```
EXECVE(2)                     System Calls Manual                     EXECVE(2)

NAME
     execve – execute a file

SYNOPSIS
     #include <unistd.h>

     int
     execve(const char *path, char *const argv[], char *const envp[]);

DESCRIPTION
     execve() transforms the calling process into a new process.  The new
     process is constructed from an ordinary file, whose name is pointed to by
     path, called the new process file.  This file is either an executable
     object file, or a file of data for an interpreter.  An executable object
     file consists of an identifying header, followed by pages of data
     representing the initial program (text) and initialized data pages.
     Additional pages may be specified by the header to be initialized with
     zero data;  see a.out(5).

     An interpreter file begins with a line of the form:

           #! interpreter [arg ...]

     When an interpreter file is execve()'d, the system runs the specified
     interpreter.  If any optional args are specified, they become the first
     (second, ...) argument to the interpreter. The name of the originally
     execve()'d file becomes the subsequent argument; otherwise, the name of
     the originally execve()'d file is the first argument.  The original
     arguments to the invocation of the interpreter are shifted over to become
     the final arguments.  The zeroth argument, normally the name of the
:
```

lorenzo — less ◂ man execve — 81×33

```
RETURN VALUES
     As the execve() function overlays the current process image  with a new
     process image, the successful call has no process to return to.  If
     execve() does return to the calling process, an error has occurred; the
     return value will be -1 and the global variable errno is set to indicate
     the error.

ERRORS
     execve() will fail and return to the calling process if:

     [E2BIG]            The number of bytes in the new process's argument list
                        is larger than the system-imposed limit.  This limit is
                        specified by the sysctl(3) MIB variable KERN_ARGMAX.

     [EACCES]           Search permission is denied for a component of the path
                        prefix.

     [EACCES]           The new process file is not an ordinary file.

     [EACCES]           The new process file mode denies execute permission.

     [EACCES]           The new process file is on a filesystem mounted with
                        execution disabled (MNT_NOEXEC in ⟨sys/mount.h⟩).

     [EFAULT]           The new process file is not as long as indicated by the
                        size values in its header.

     [EFAULT]           Path, argv, or envp point to an illegal address.

     [EIO]              An I/O error occurred while reading from the file
                        system.

:
```

Summary: exec()

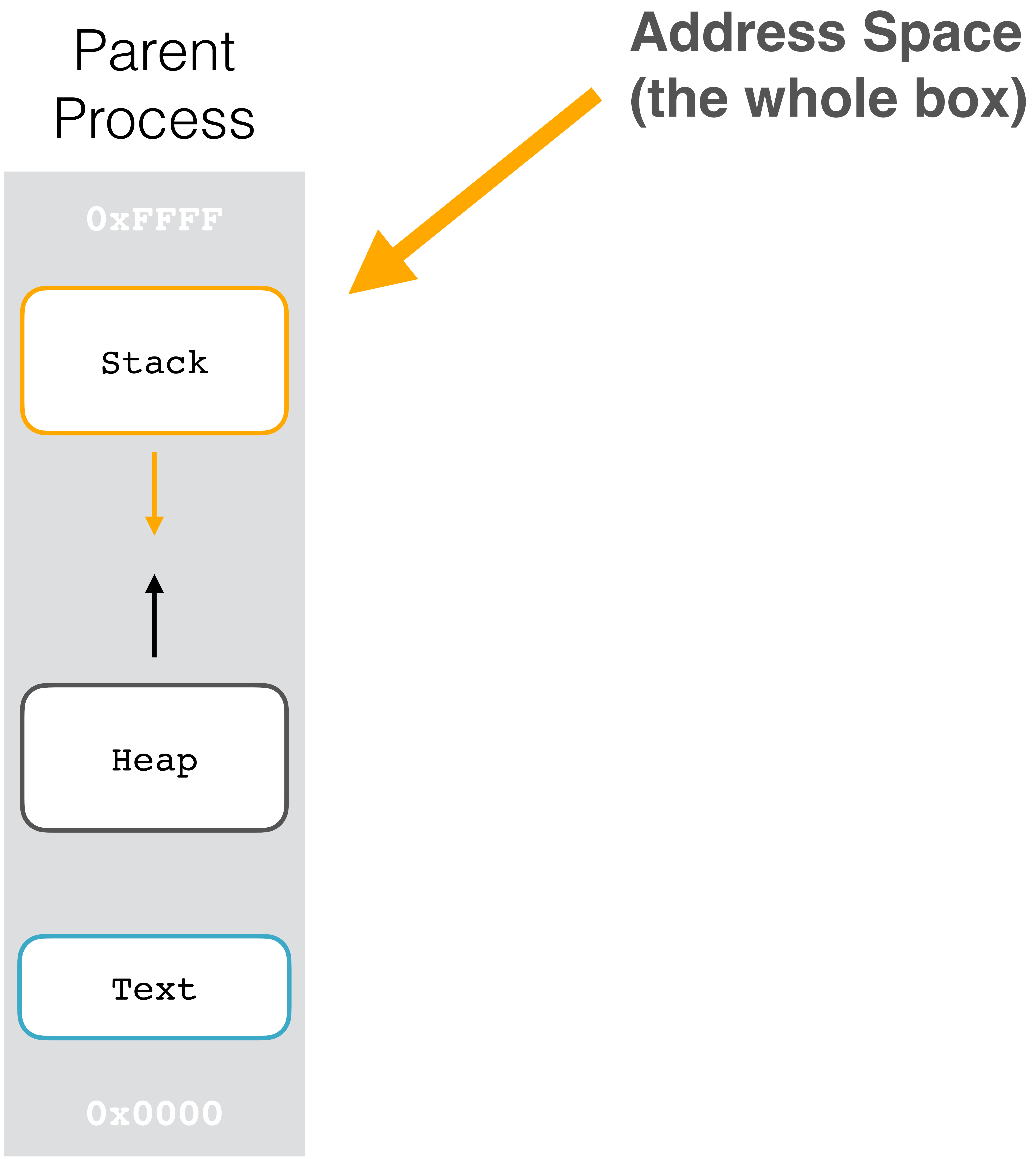
- **`exec()`** is a system call that starts a new process by transforming the current process.
- **`exec()`** duplicates the **address space**, then overwrites the text segment, and r**einitializes the stack and heap**.
- **Does not return if successful**, **returns -1 on failure**. Use **errno** to figure out what went wrong:
  - **`man 3 errno`**
- Many different **`exec()`** variants: `exec()`, `execl()`, `execle()`, `execlp()`, `execv()`, and `execvp()`.

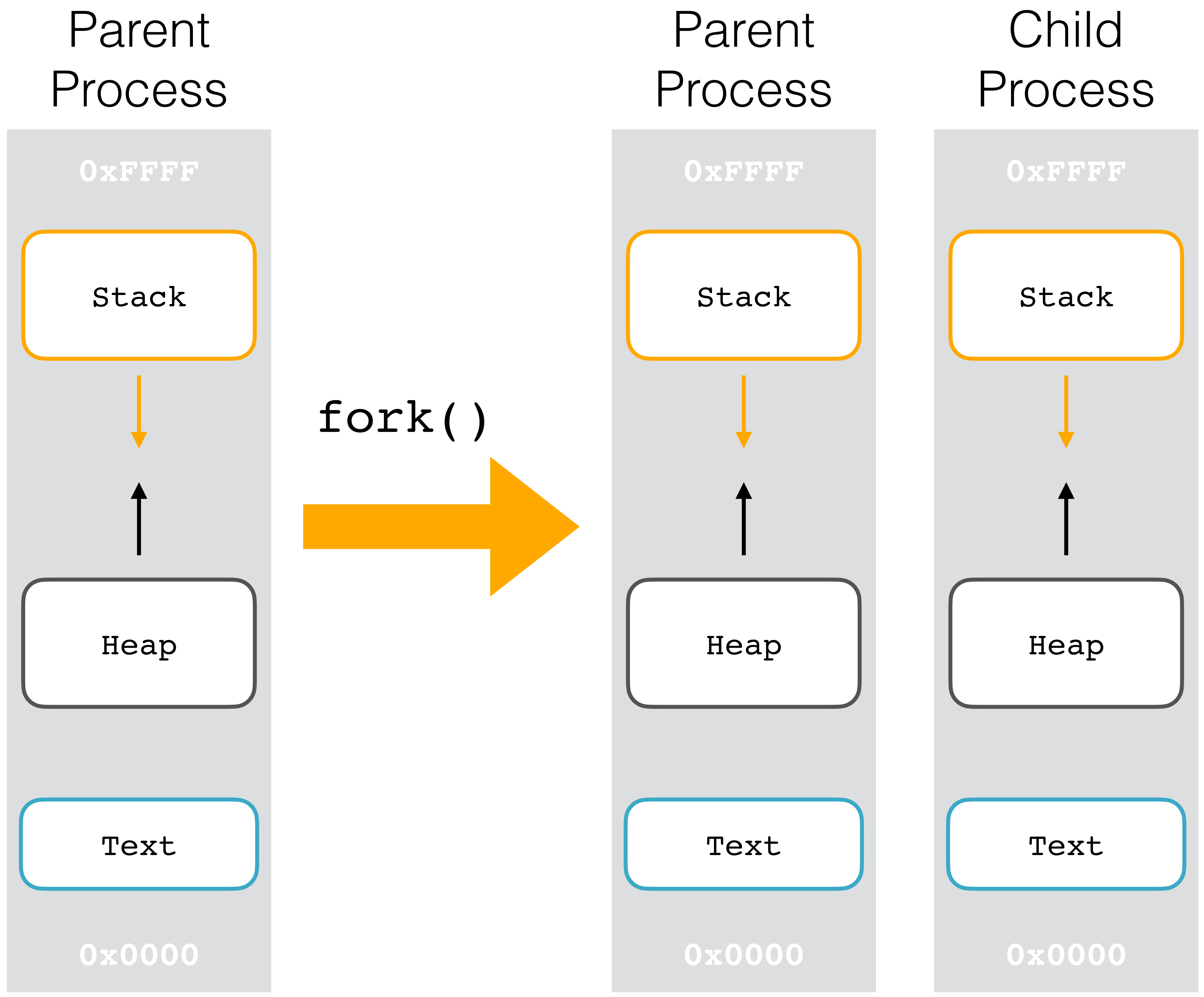
A simple shell

```c
#define TRUE 1

while (TRUE) {
  type_prompt();
  read_command(command, parameters);




    execve(command, parameters, 0);

}
```

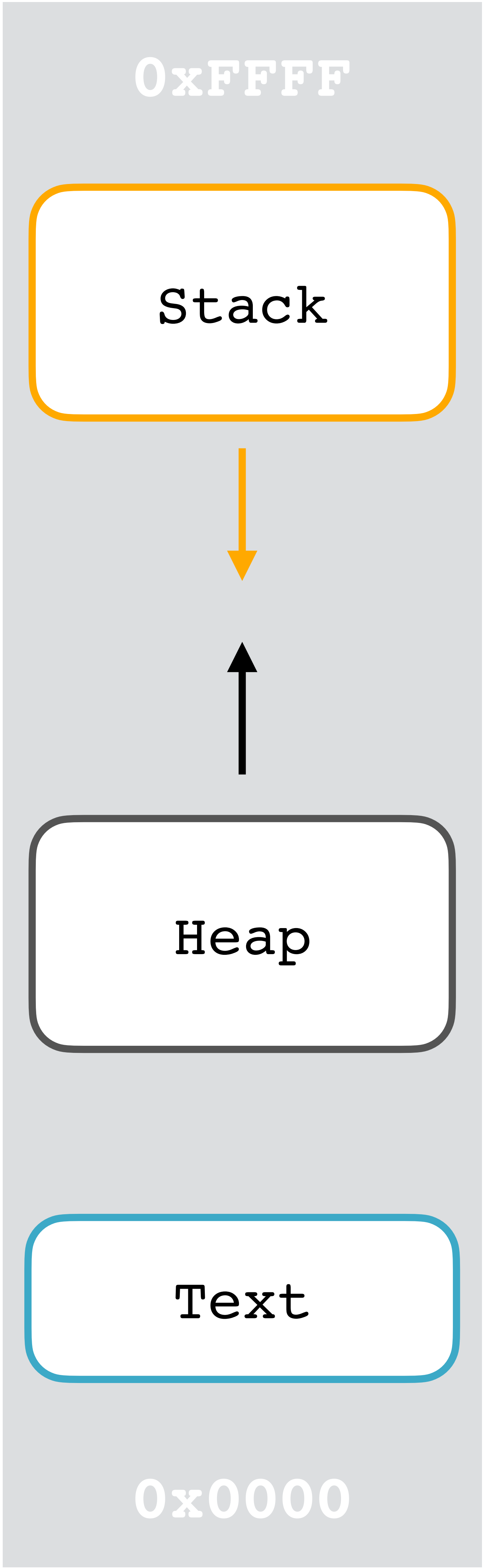
fork()

- `fork()` is a system call that creates an (almost) exact copy of the running process.The original process is called the **parent** and the new process the **child**.
- The child process starts running at the return from **`fork()`** and has its own copy of the **address space**, **registers**, any **open file handles**, etc.
- How does the forked process know it is the child? Fork returns a **PID** to the parent, but a **0** to the child.


Parent
Process
Address Space
(the whole box)
0xFFFF
Stack
Heap
Text
0x0000


Parent
Process
0xFFFF
Stack
Heap
Text
0x0000
fork()
Parent
Process
0xFFFF
Stack
Heap
Text
0x0000
Child
Process
0xFFFF
Stack
Heap
Text
0x0000

Virtual address space?



> The **virtual address space** is abstraction of the physical memory that makes memory simple for the process, e.g., a byte stream.

> Each byte in memory is associated with an **address**, allowing the process to access the memory location.

> We've divided the address space into three segments:

- **`stack:`** used to support function calls and local variables, grows and shrinks during execution.
- **`heap:`** used for dynamically-allocated, user-managed memory.
- **`text:`** the instructions of the program

fork()

- How does the forked process know it is the child? **Fork returns a PID to the parent, but a 0 to the child**.

a simple shell

```c
#define TRUE 1

while (TRUE) {
  type_prompt();
  read_command(command, parameters);

  if (fork() != 0) {

  } else {

  }
}
```

```c
execve(command, parameters, 0);
```

a simple shell

```
#define TRUE 1

while (TRUE) {
  type_prompt();
  read_command(command, parameters);

  if (fork() != 0) {
    /* Parent’s code */
  } else {
    execve(command, parameters, 0);
  }
}
```

wait()

- **`wait()`** is a system call that blocks the parent process until **one** of its children has finished executing.
- Performing a wait allows the OS to release the resources associated with the child, thereby avoiding **zombies** (technical term).

A simple shell

```c
#define TRUE 1

while (TRUE) {
  type_prompt();
  read_command(command, parameters);

  if (fork() != 0) {
    wait();
  } else {
    execve(command, parameters, 0);
  }
}
```

Quick recap

- **`exec()`** is a system call transforms the current process into a new process.
- **`fork()`** is a system call that creates a new process which is an (almost) exact copy of the running process.
- **`wait()`** is a system call that blocks a parent process until one of its children has finished executing.
- **Beware:** these descriptions are simplifications of what is actually happening. As always, check out the man pages for more details.

Other system calls

- The line between **library** and **system call** can get confusing as often you will call the **glibc wrapper** around the syscall rather than **the call itself** (see **`$ man syscalls`**)
  - e.g., glibc has a wrapper function **`truncate`** for the **`truncate`** system call
  - e.g., **`system`** is a wrapper around the **`execl`** and **`fork`** system calls. **Tip:** look at the top left corner of the man page, if the page is in **Section 2** then it's a **system call**. If the page is in **Section 3**, it's a **library function**.
  - e.g. “SYSTEM(3)” means the page for **`system`** is in **section 3**

Bonus: how does Linux do system calls?

System calls, the good old way

int 0x80 to the rescue

> The good ol' way to trigger a system call on x86 is the following:

- Write the system call code to the **%EAX register**
- Trigger **software interrupt 0x80**

> How do I need **which code** for w**hich system call**?

- There is a practical table: https://faculty.nps.edu/cseagle/assembly/sys_call.html
- (Observe that %EBX, %ECX, %EDX, %ESX, %EDI are used to pass **additional parameters**)

System calls, the newfangled way

- Triggering a software interrupt is **somewhat slow** for various reasons
- Modern x86_64 architectures use **specialized instructions**
  - **`syscall`**, **`sysret`**
  - Save **less register state** than **`int 0x80`**
  - **`int 0x80`** requires a lookup into **interrupt table**, **`syscall`** jumps directly to **syscall handler**
  - Parameters still passed via **registers**

That’s it for today!