

CS110 Spring 2021

Lecture 5: Introduction to Multiprocessing
(note: pre-lecture material included for your convenience)

Principles of Computer Systems

Stanford University, Dept. Of Computer Science

Lecturer: Roz Cyrus

Content adapted from material by Jerry Cain. Diagrams by Roz.

Lecture Overview

- Pre-lecture content: control flow, exceptions, & context switches
- Process life cycle
- The **fork** system call
- The **waitpid** system call

Reading Material

Bryant & O'Hallaron: Sections 1 - 4 of: Chapter 1 (reader) or 8 (full textbook)

Updates

- Congrats on finishing Assignment 1!
- Assignment 2 is out...get started EARLY (a.k.a. now)
- Office hours

Accessing Code Examples

- Today's lecture examples reside within:
`/usr/class/cs110/lecture-examples/processes`.
 - First **ssh** into a myth machine (ssh yourusername@myth.stanford.edu). When prompted for your password, it is normal for the text not to appear as you enter your password. Once logged onto a myth machine, **cd** into the above directory.
 - To get started, type:
git clone /usr/class/cs110/lecture-examples cs110-lecture-examples
at the command prompt to create a local copy of the master.
 - Each time I mention there are new examples (or whenever you think to), descend into your local copy and type **git pull**. Doing so will update your local copy to match whatever the master has become.

Pre-Lecture Slides

Processes Recap

- A **process** is an instance of a program in execution, versus a **program**, which is the code and data. Processes are active, whereas programs are passive.
- A program always runs in the **context** of some process. This context has state that the program needs in order to properly run.
- Processes provide two key abstractions to an application:
 - 1. A private address space that provides **the illusion that our program has exclusive use of memory** (discussed in the last lecture)
 - 2. An independent logical control flow that provides **the illusion that our program has exclusive use of the processor**

Introducing Control Flow

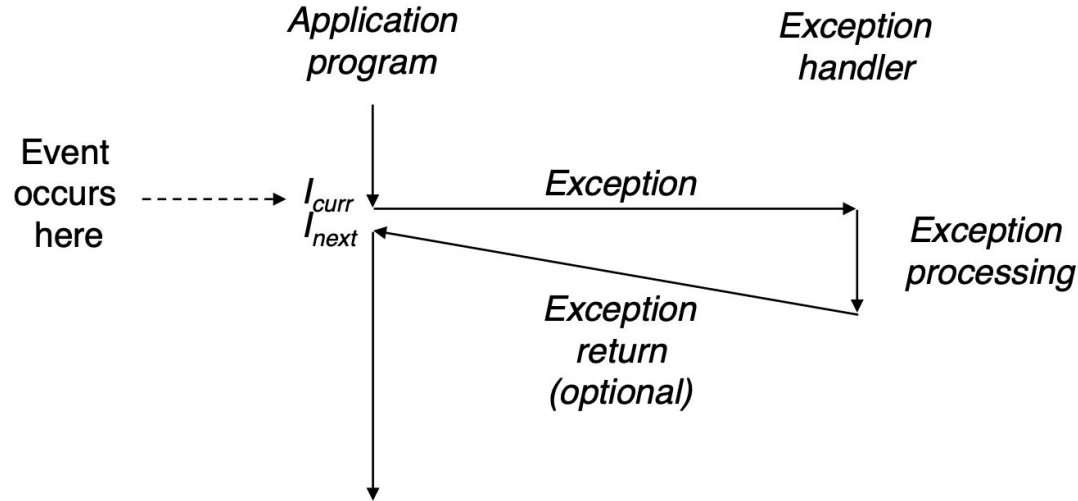
- The processor's program counter (which contains the next address to be executed) assumes a sequence of addresses that each correspond to an instruction.
- Each transition from one instruction to another is a **control transfer**, and a sequence of such control transfers is called the **flow of control (control flow)** of the processor.
 - A “smooth” control flow runs contiguous (adjacent in memory) instructions. Abrupt changes in this flow could occur from something program-related, such as jumps, function calls and returns.
 - Other abrupt changes to the system may not have to do with the running program at all. Examples:
 - A hardware timer going off
 - Network packets arrive and must be stored in memory
 - Another program is ready to receive data
 - Parent processes (explained later) are notified when their children processes terminate
 - These kinds of abrupt changes are called **exceptional control flow** (keyword: “exception”)

Control Flow (continued)

- Exceptional control flow occurs at all levels of a computer system. Examples:
 - **Hardware level:** events detected by hardware trigger abrupt control transfers to **exception handlers**
 - **OS level:** the kernel transfers control from one user process to another via **context switches** (explained in a later slide).
 - **Application level:** a process can send a **signal** to another process, which abruptly transfers control to a **signal handler** in the recipient.
 - A signal is a small message that notifies a process that an event of some type occurred. Signals are often sent by the kernel, but they can be sent from other processes as well. Signals and signal handlers will be explained in greater detail in the next lecture.

Exceptions

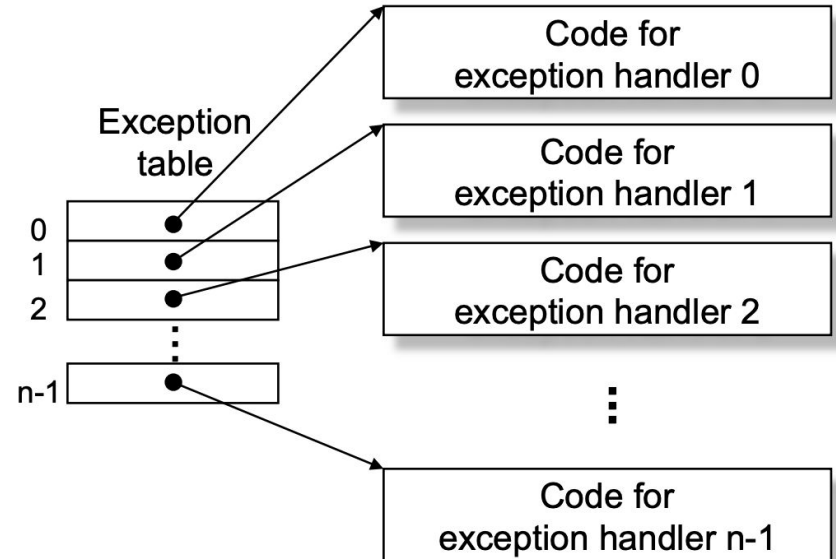
- An **exception** is an abrupt change in the control flow in response to some change in the processor's state.



- A change in the processor's state (an **event**) triggers an abrupt control transfer (an exception) from the program to an exception handler. After it finishes processing, the handler either returns control to the interrupted program (to either the same instruction or the next one) or aborts.

Exceptions (continued)

- Each type of possible exception in a system is assigned a unique nonnegative integer **exception number**. x86-64 systems: up to 256 exception types, numbered 0 - 255.
- Some numbers are assigned by designers of the processor. Examples:
 - Divide by zero (exception number 0)
 - Arithmetic overflows
 - Page faults (exception number 14)
 - Memory access violations
- Other numbers are assigned by the designers of the kernel. (e.g. system calls)
- An **exception table** stores the address of the handler code for each exception. It is initialized at boot time. Exception control handlers run in **kernel mode**, meaning they have complete access to all system resources. Once hardware triggers the exception, the handler runs in software and then optionally returns to the interrupted program.



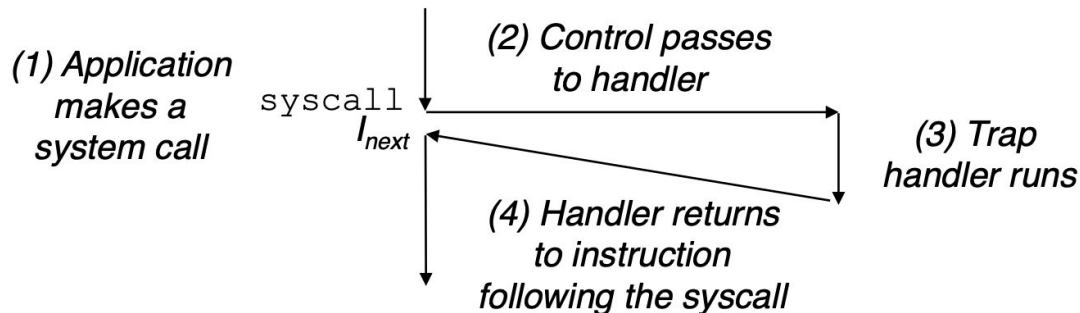
Exceptions (continued)

Types of exceptions

Class	Cause	Async/sync	Return behavior
Interrupt	Signal from I/O device that is external to the processor	Async	Always returns to next instruction
Trap	<u>Intentional</u> exception as a result of executing an instruction. Includes system calls	Sync	Always returns to next instruction
Fault	Potentially recoverable error (e.g. a page fault)	Sync	Might return to current instruction
Abort	Nonrecoverable error	Sync	Never returns

More About System Calls

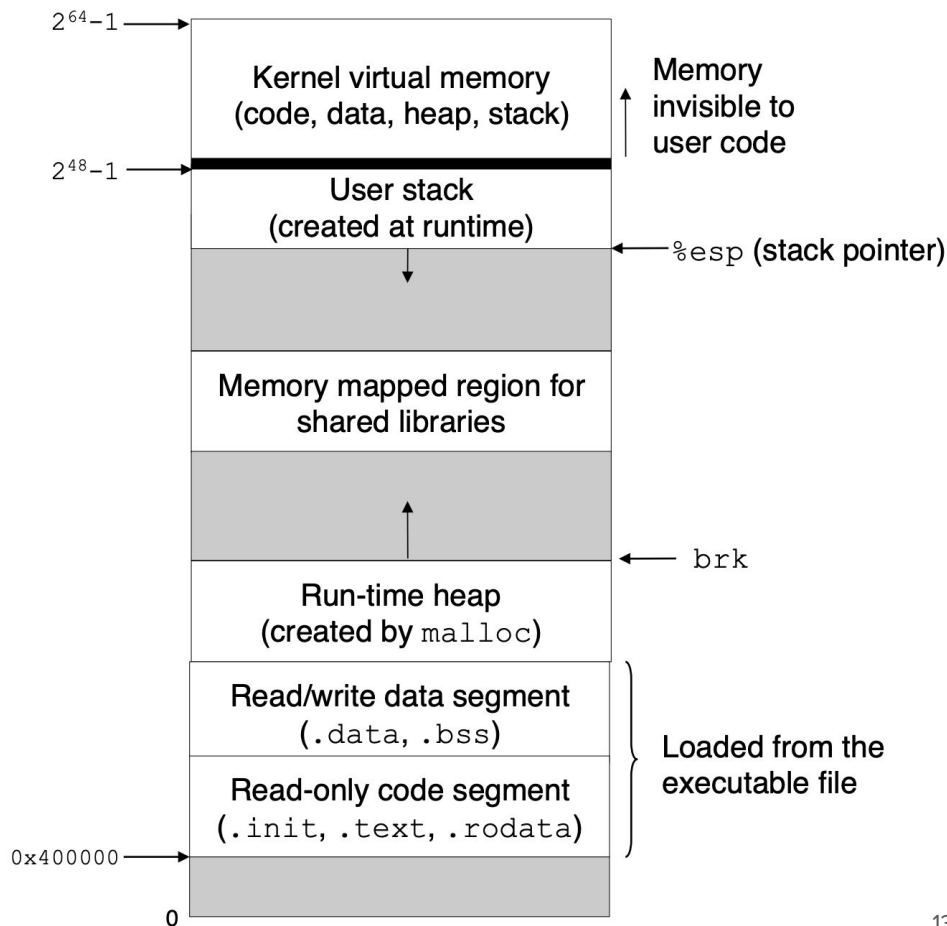
- The most important use of traps is to provide a procedure-like interface between user programs and the kernel, known as a **system call** (explained in the previous lecture). ([View a list](#) of Linux system calls)
 - Regular functions run in **user mode**, which restricts the types of instructions they can execute.
 - A system call runs in **kernel mode**, which allows it to execute privileged instructions and access a stack defined in the kernel.
- User programs often need to request services from the kernel such as handing a file (**read**, **write**, **open**, etc.) and creating a new process (**fork**).
- To allow controlled access to such kernel services, processors provide a special `syscall n` instruction that user programs can execute when they want to request service n .



Recap: "Exclusive" Memory Use

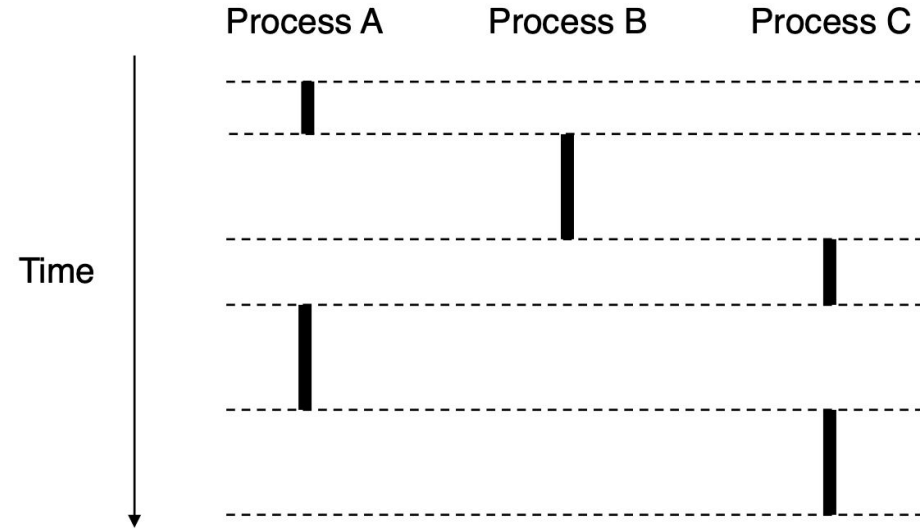
System Calls Summary (continued)

- A system call uses a software interrupt (a.k.a. a trap) to transfer control to the OS kernel. The user can only call a set of well-defined system calls, and there is little room for a security breach.
- Once the kernel is running a system call, it is in complete control of the system, and can access the necessary resources to fulfill the system call's needs.
- After a system call, the kernel returns control to the user program.



“Exclusive” Processor Use

- Processes provide each program with the illusion that it has exclusive use of the processor. Each vertical bar in the right image represents a portion of the logical control flow for a process. Note how the execution of the three logical flows is interleaved. Each process takes turns using the processor; each runs for a while then is temporarily suspended while others get their turn.
- Two flows run **concurrently** if their execution overlap in time. Multiple flows executing concurrently is **concurrency**. This is independent of # of cores. **Which processes are concurrent?**
- This notion of a process taking turns with other processes is also called **multitasking**.
- There may be tens, hundreds, or thousands of processes "running" at once, but on a single-core system, only one can run at a time, and this is coordinated by the OS. On multi-core machines (like most modern computers), multiple programs can literally run at the same time, one per core. **Parallel flows** are a subset of concurrent flows: flows running concurrently on different processor cores or computers. This is true **multiprocessing**.
- Each time period that a process executes a portion of its flow is called a **time slice (roughly 20 milliseconds)**

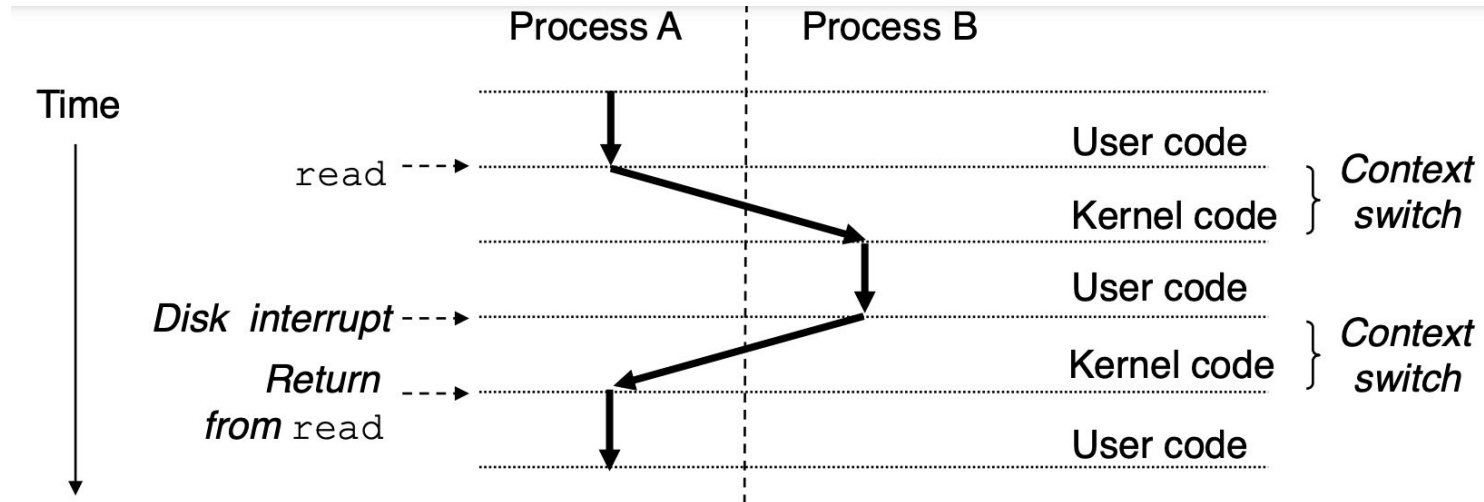


Context Switches

- The operating system kernel implements multitasking using a higher-level form of exceptional control flow known as a **context switch**. The decision to preempt the current process and restart a previously preempted process is called **scheduling**.
- A context switch (1) Saves the context of the current process, (2) restores the saved context of some previously preempted process, and (3) passes control to this newly restored process.
- Recall that the kernel maintains a **context** for each process, which is state that the kernel needs in order to restart a preempted process. The context is stored in a **process control block** (one per process) which stores a lot of process-related information, including:
 - Contents of general-purpose and floating-point registers
 - Program counter
 - User and kernel stack
 - Status registers
 - Code and in-memory data
 - A file descriptor table

Context Switches

In the below example, the kernel runs some code for process A in user mode. It then reaches the **read** syscall, which requests data from disk to be loaded into memory, but since this will take a while, the kernel context switches from A to B: the kernel first does work on A's behalf in kernel mode, then stays in kernel mode and does some work for B. The B's user code is run in user mode until the disk interrupts and says, "hey! I have data ready for process A!" Shortly after, the kernel will get tired of running B and will context switch to A (first wrapping up B in kernel mode, then some kernel mode work for A). It will then proceed with A's code in user mode, picking up right after the **read** call. A will continue happily along until the next exception occurs.



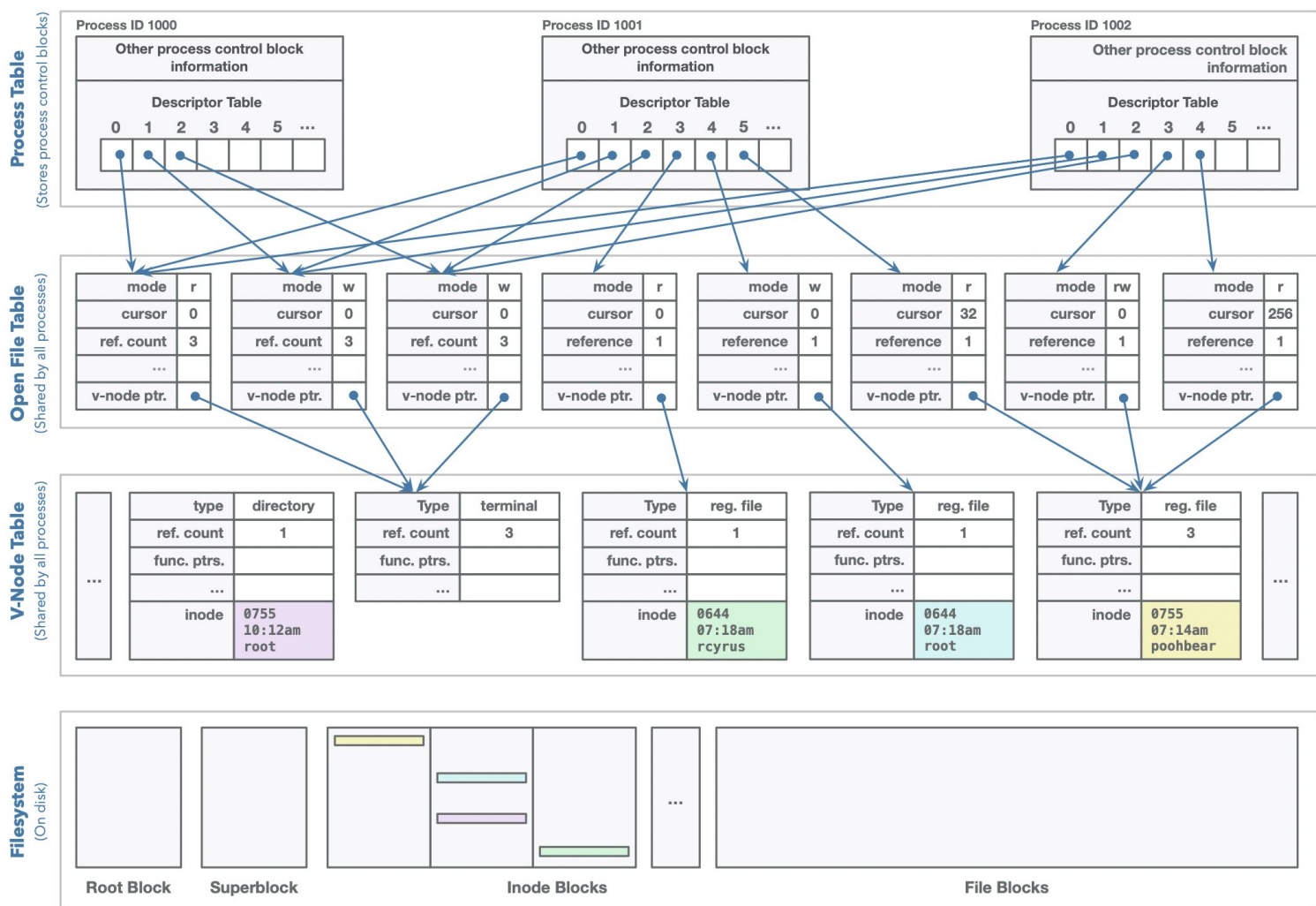
Let's talk about multiprocessing!

Multiprocessing Recap

- In the CS curriculum so far, your programs have operated in a single process, meaning, basically, that one program was running your code, line-for-line. The operating system made it look like your program was the only thing running, and that was that.
- Now, we are going to move into the realm of **multitasking**, where you control more than one process at a time with your programs. You will tell the OS, “do these things concurrently”, and it will.

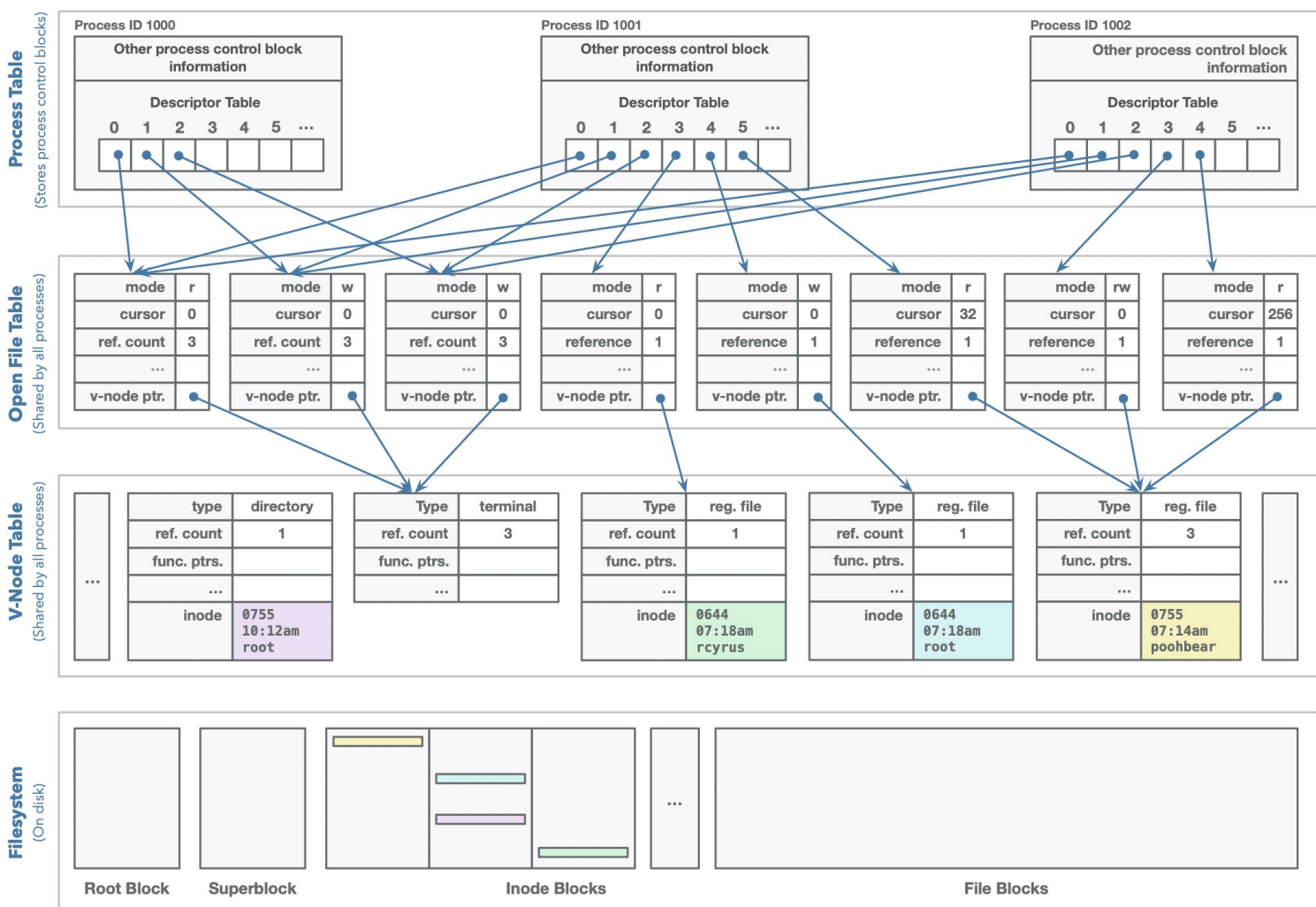
Kernel Data Structures Recap

The user program sees a **file descriptor** as the identifier needed to interact with a resource (most often a file) via **read**, **write**, and **close** calls. Internally, that descriptor is an index into the **descriptor table** of the process.



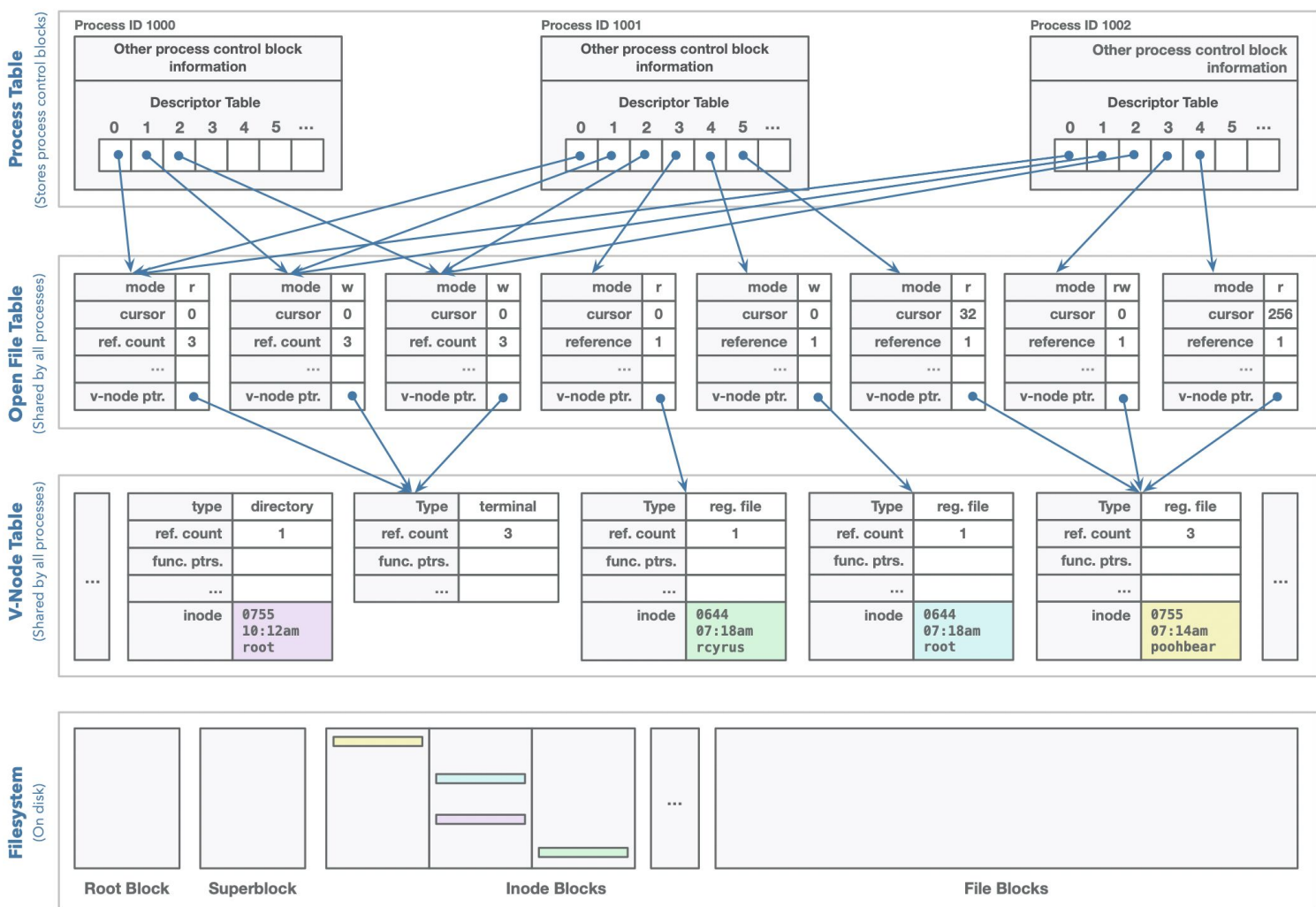
Kernel Data Structures Recap

Processes do not have direct access to files or inode tables. Each process must pass a file descriptor to the kernel via a **syscall**, which will handle the file for the process.



Kernel Data Structures Recap

The **process control block** tracks which descriptors are in use and which ones aren't. When allocating a new descriptor, the OS typically chooses the smallest available number.

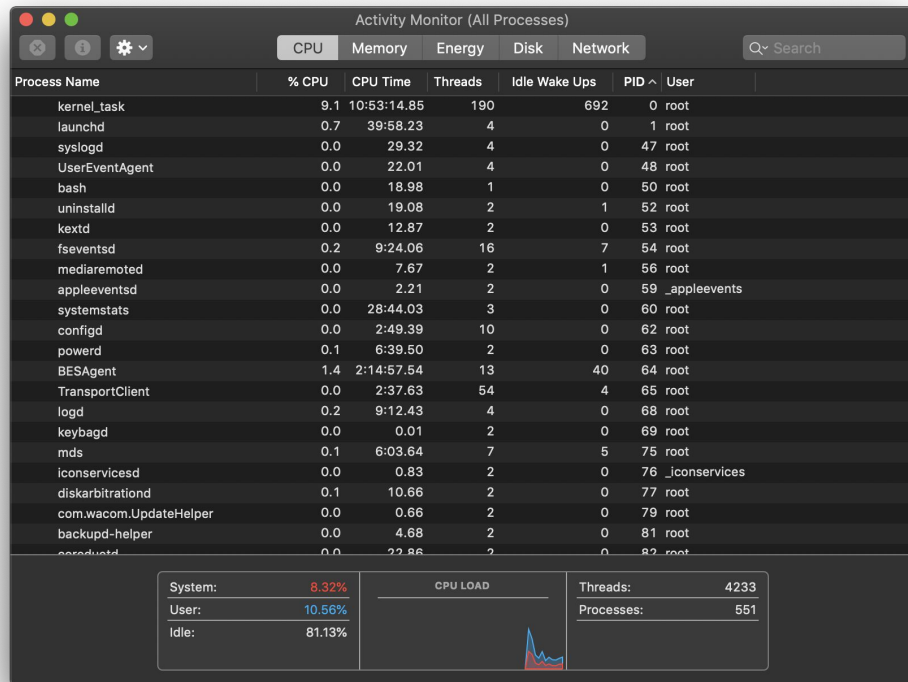


Kernel Data Structures Explained

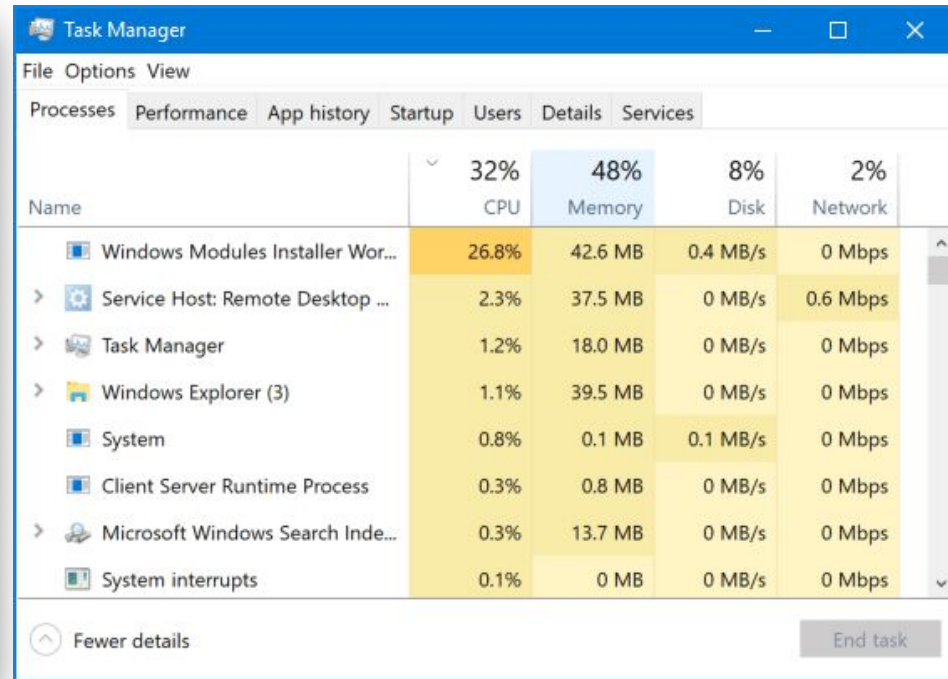
- **Process table:** stores information about all currently running processes (each entry is a **process control block**). Process control blocks store many things (the user who launched it, what time it was launched, CPU state, etc.). Among the many items it stores is the **descriptor table**.
- **Descriptor Table:** each process has its own separate descriptor table whose entries are indexed by the process's open file descriptors. Each open descriptor entry points to an entry in the **open file table**. Descriptors 0, 1, and 2 are understood to be treated as standard input, standard output, and standard error, but there are no predefined meanings for descriptors 3 and up.
- **Open file table:** contains info about all open files. It is shared by all processes. An open file table entry maintains information about an active session with a file (or something that behaves like a file, like the terminal or a network connection). Each file table entry contains pertinent file information, including (for our purposes):
 - The **file mode:** tracks whether we're reading, writing, or both.
 - The **current file position (cursor):** tracks a position within the file payload
 - A **reference count** of the number of descriptor entries (across all processes) that currently point to the entry.
 - Closing a descriptor decrements the reference count in the associated file table entry. The kernel will not delete the file table entry until its reference count is 0.
 - A pointer to an entry in the **v-node table**
- **V-node table:** Each entry contains most of the information in the **stat** structure that we previously explained (including **st_mode** and **st_size**). This is also shared by all processes. Each entry describes the actual underlying files. Vnodes (virtual file system entries) exist so that kernel functions don't need to care about which filesystem they are dealing with. Vnodes contain information about the file's inode.

Viewing Processes on Your Computer

Mac: Activity Monitor



Windows: Task Manager



More About Processes

- When you start a program, it runs in a single process. It has a **process id** (an integer) that the OS assigns. A program can find out its process id with the `getpid` system call:

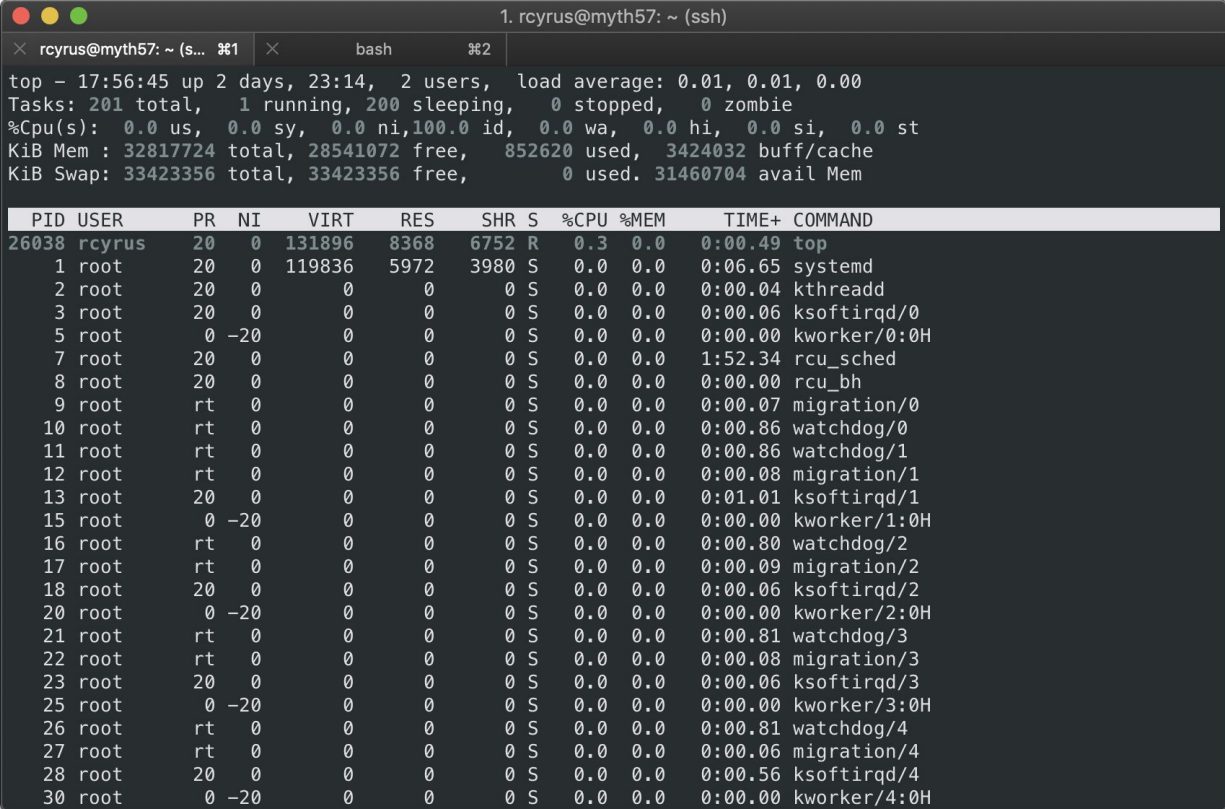
```
// file: getpidEx.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h> // for getpid
int main(int argc, char *argv[]) {
    pid_t pid = getpid();
    printf("My process id: %d\n", pid);
    return 0;
}
```

```
$ ./getpidEx
My process id: 1186537
```

- All programs are associated with one or more processes, and a process is scheduled to run its code by the OS.

Viewing Processes via a Linux shell

- To view processes running on a Linux machine like the **myth** machines, run **top**.
- You can also run **ps -A** to view a static list of all running processes.
- You can run **pstree** to see the process tree



```
1. rcyrus@myth57: ~ (ssh)
rcyrus@myth57: ~ (s... %1 | bash %2
top - 17:56:45 up 2 days, 23:14, 2 users, load average: 0.01, 0.01, 0.00
Tasks: 201 total, 1 running, 200 sleeping, 0 stopped, 0 zombie
%Cpu(s):  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem : 32817724 total, 28541072 free,  852620 used,  3424032 buff/cache
KiB Swap: 33423356 total, 33423356 free,  0 used. 31460704 avail Mem

  PID USER      PR  NI   VIRT   RES   SHR  S  %CPU  %MEM     TIME+ COMMAND
 26038 rcyrus    20   0 131896  8368  6752 R   0.3   0.0   0:00.49 top
     1 root      20   0 119836  5972  3980 S   0.0   0.0   0:06.65 systemd
     2 root      20   0         0         0         0 S   0.0   0.0   0:00.04 kthreadd
     3 root      20   0         0         0         0 S   0.0   0.0   0:00.06 ksoftirqd/0
     5 root       0 -20         0         0         0 S   0.0   0.0   0:00.00 kworker/0:0H
     7 root      20   0         0         0         0 S   0.0   0.0   1:52.34 rcu_sched
     8 root      20   0         0         0         0 S   0.0   0.0   0:00.00 rcu_bh
     9 root      rt    0         0         0         0 S   0.0   0.0   0:00.07 migration/0
    10 root      rt    0         0         0         0 S   0.0   0.0   0:00.86 watchdog/0
    11 root      rt    0         0         0         0 S   0.0   0.0   0:00.86 watchdog/1
    12 root      rt    0         0         0         0 S   0.0   0.0   0:00.08 migration/1
    13 root      20   0         0         0         0 S   0.0   0.0   0:01.01 ksoftirqd/1
    15 root       0 -20         0         0         0 S   0.0   0.0   0:00.00 kworker/1:0H
    16 root      rt    0         0         0         0 S   0.0   0.0   0:00.80 watchdog/2
    17 root      rt    0         0         0         0 S   0.0   0.0   0:00.09 migration/2
    18 root      20   0         0         0         0 S   0.0   0.0   0:00.06 ksoftirqd/2
    20 root       0 -20         0         0         0 S   0.0   0.0   0:00.00 kworker/2:0H
    21 root      rt    0         0         0         0 S   0.0   0.0   0:00.81 watchdog/3
    22 root      rt    0         0         0         0 S   0.0   0.0   0:00.08 migration/3
    23 root      20   0         0         0         0 S   0.0   0.0   0:00.06 ksoftirqd/3
    25 root       0 -20         0         0         0 S   0.0   0.0   0:00.00 kworker/3:0H
    26 root      rt    0         0         0         0 S   0.0   0.0   0:00.81 watchdog/4
    27 root      rt    0         0         0         0 S   0.0   0.0   0:00.06 migration/4
    28 root      20   0         0         0         0 S   0.0   0.0   0:00.56 ksoftirqd/4
    30 root       0 -20         0         0         0 S   0.0   0.0   0:00.00 kworker/4:0H
```

But Wait...What is a shell?

- A **shell** is a command-line interpreter (CLI) that runs programs on behalf of the user.
- It is called a shell because it is a “wrapper” around the kernel.
- To find out which shell you are running:

```
myth54$ ps -p $$
  PID TTY          TIME CMD
 28740 pts/8    00:00:00 bash
```

- The first **\$** denotes a shell variable, and the second **\$** gets the process ID of the current process, which is the shell. Another way to get PID of the shell: **echo \$\$**
- Columns:
 - PID is the process ID
 - TTY is the terminal (device file) associated with the process
 - TIME is the total CPU usage
 - CMD is the command (name of the process)
- Differences between shell, terminal, etc.: read [this](#)

Bonus: Useful Linux Commands

- To view a list of file descriptors in the shell's descriptor table (28470 below represents the PID of the shell):

```
myth54$ ls /proc/28740/fd/  
0 1 2 255 3 4
```

- To see which files these descriptors point to in the open file table (**ls**of = “list of open files”):

```
myth54$ ls -a -p 28740  
COMMAND  PID  USER  FD  TYPE  DEVICE  SIZE/OFF  NODE  NAME  
[...]  
bash     28740 rcyrus 0u   CHR   136,8    0t0      11    /dev/pts/8  
bash     28740 rcyrus 1u   CHR   136,8    0t0      11    /dev/pts/8  
bash     28740 rcyrus 2u   CHR   136,8    0t0      11    /dev/pts/8  
bash     28740 rcyrus 255u CHR   136,8    0t0      11    /dev/pts/8
```

- /dev/pts/8** is the terminal, which you can verify with the **tty** command:

```
myth54$ tty  
/dev/pts/8
```

- Thus file descriptors 0, 1, 2, and 255 are connected to the terminal for reading and writing. 255 is like a backup file descriptor that bash uses for terminal access. Note that FD is followed by one of these characters, describing the mode under which the file is open: **r** for read access; **w** for write access; **u** for read and write access.

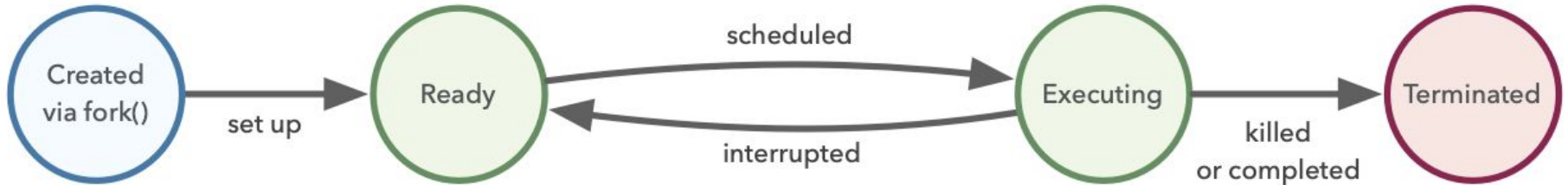
Scheduling

In order to provide the illusion of running many processes simultaneously, the operating system scheduler does the following:

- A process is allowed to run on a particular CPU core. After its time slice, the OS pauses the process, copies the necessary data into a *process control block*, and adds that data structure to a queue of ready-to-run (“runnable”) processes called the **ready queue**.
- The OS then selects another process from the ready queue, restores its state, and resumes execution of that process as if it had never stopped.
- Occasionally, a process needs to wait for something (e.g. it calls **sleep** or it waits for a network request to come in). In this case, the process is removed from the ready queue and is instead moved to the **blocked set**. (Eventually it is moved back to the ready queue when the thing it was waiting for is ready.)
- Note that the ready queue isn’t a simple ordered queue; we may have high-priority processes that should get more CPU time. The scheduler employs a sophisticated algorithm to balance process needs. You are never given any guarantees about process scheduling.

Process Life Cycle

Here is a naive view of the life cycle for each newly created process. Think of this as the scheduler's view of the process.



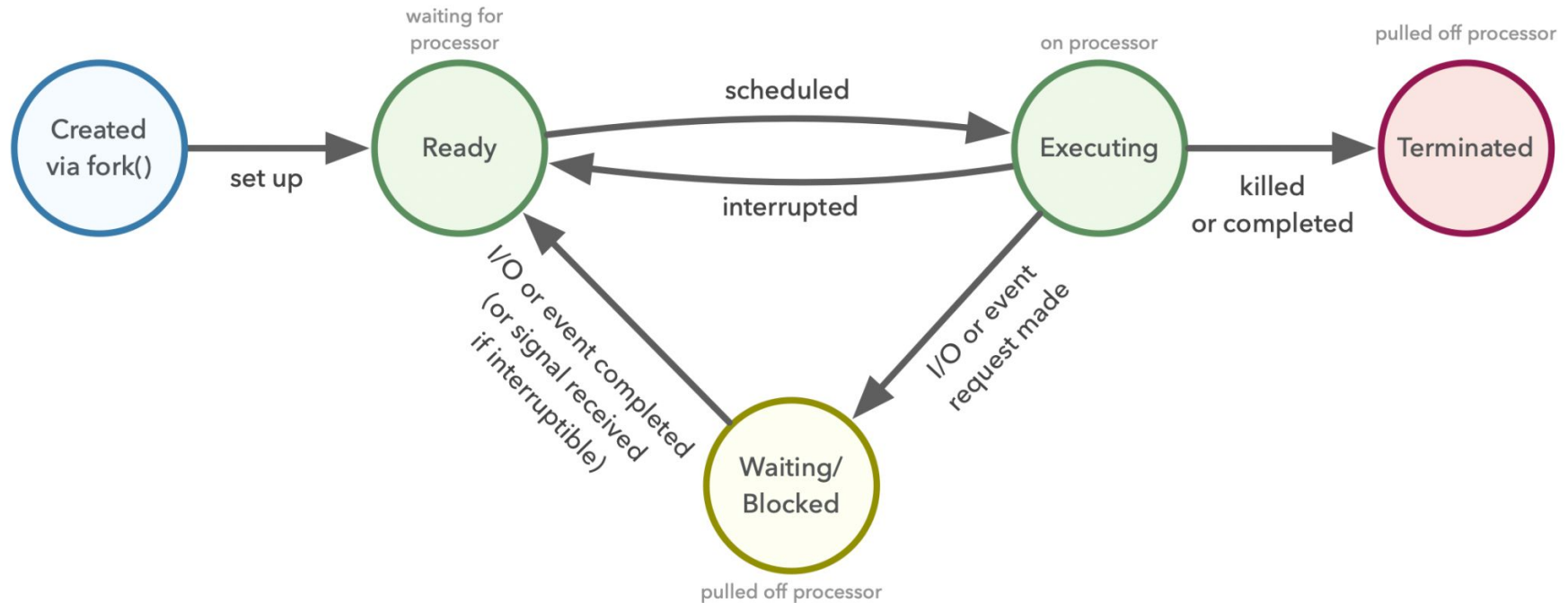
- **Created:** a brand new process that the kernel sets up to prep for execution. Processes are created via the `fork()` system call (more on this in a few slides).
- **Ready:** all set up and ready to be assigned to a processor and scheduled for execution
- **Executing:** currently executing on the processor (only one process can execute on a processor at a time!)
- **Terminated:** stopped permanently, for one of three reasons: (1) receiving a signal that terminates the process, (2) returning from `main`, (3) calling `exit` (note: `exit` terminates process with the given status code; another way to set the exit status is to return an integer from the `main` routine)

Process Life Cycle

What happens if the process needs to wait for a resource?

Process Life Cycle

Here's a 5-state process life cycle with the waiting state introduced.



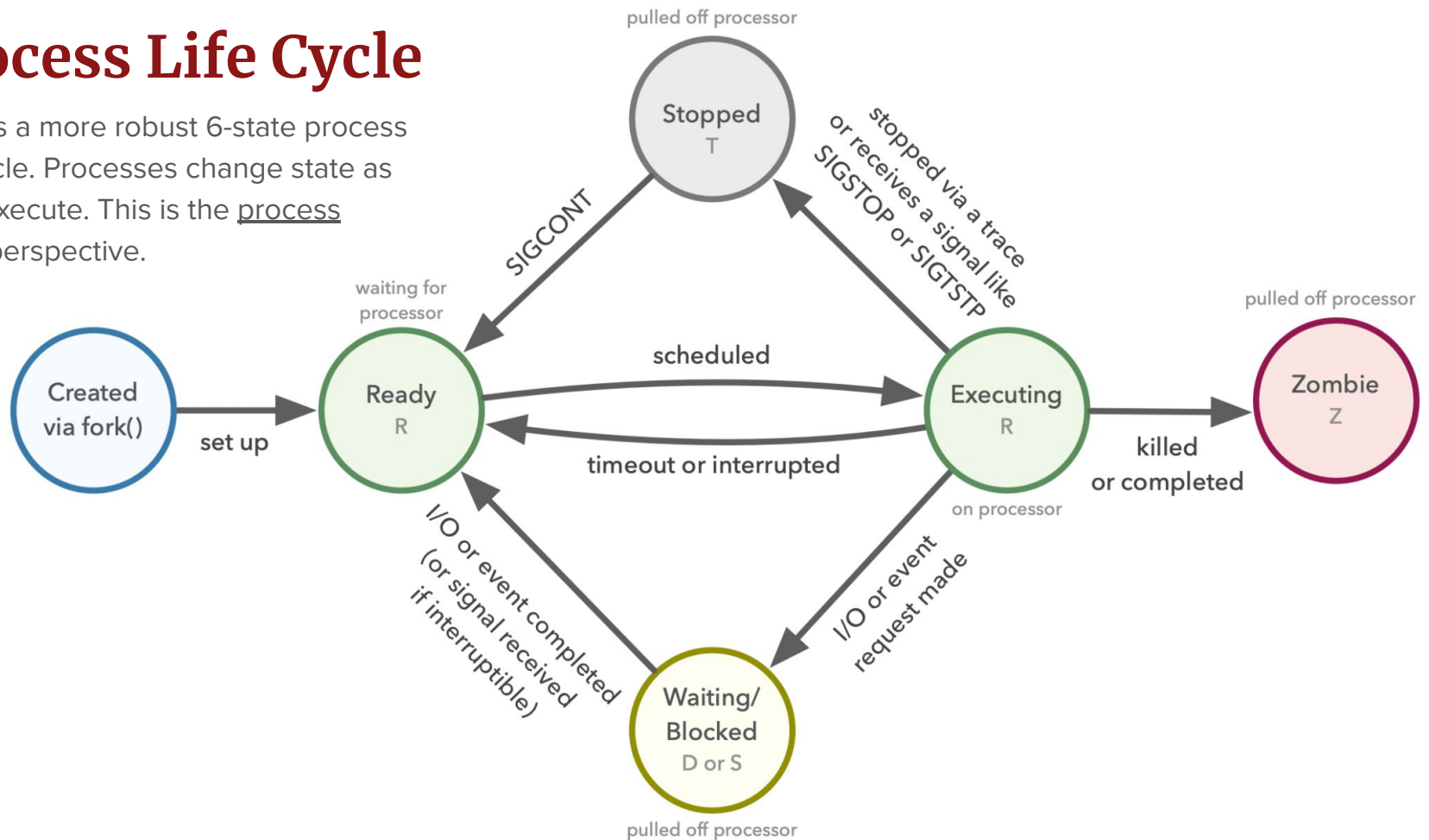
- **Waiting (a.k.a. Sleeping):** blocked because it is waiting for some event to complete (e.g. input/output)

Process Life Cycle

What happens if we stop a program? (Think 'CTRL-Z')

Process Life Cycle

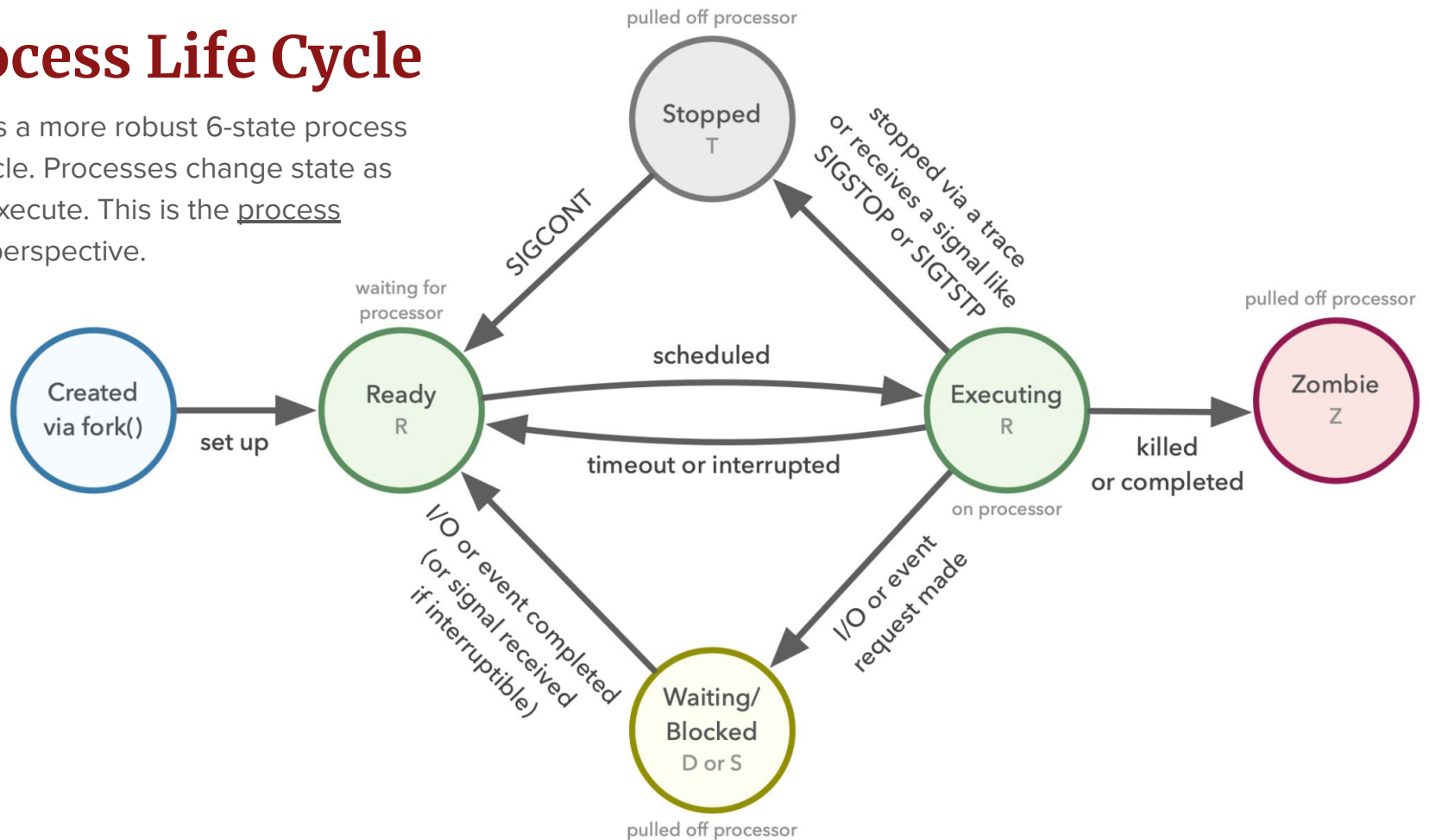
Here is a more robust 6-state process life cycle. Processes change state as they execute. This is the process state perspective.



- **Stopped:** process execution stops and won't be scheduled until it receives a **SIGCONT** signal.

Process Life Cycle

Here is a more robust 6-state process life cycle. Processes change state as they execute. This is the process state perspective.



- As far as the process is concerned, the ready state and executing states are the same (which is why they both are in the 'R' state).

ps State Codes

- When running **ps**, here are the different values that the s, stat and state output specifiers (header "STAT" or "S") will display to describe the state of a process:

```
D    uninterruptible sleep (usually IO)
I    Idle kernel thread
R    running or runnable (on run queue)
S    interruptible sleep (waiting for an event to complete)
T    stopped by job control signal
t    stopped by debugger during the tracing
W    paging (not valid since the 2.6.xx kernel)
X    dead (should never be seen)
Z    defunct ("zombie") process, terminated but not reaped by its parent
```

- We'll ignore the light gray statuses for the purposes of this class; they are just there for completeness.
- For more info (if you're curious), check out the task state bitmask as defined [here](#) (the beauty of open source!)

ps State Codes

```
D    uninterruptible sleep (usually IO)
R    running or runnable (on run queue)
S    interruptible sleep (waiting for an event to complete)
T    stopped by job control signal
Z    defunct ("zombie") process, terminated but not reaped by its parent
```

- **R**: running or runnable (ready)

This state means that the process is either **ready** to run (it has all the resources that it needs to run) but the processor is currently unavailable, or it is currently **executing** on the processor. As far as the process implementation on linux is concerned, the two states are the same and are thus both represented as **R**.

ps State Codes

D	uninterruptible sleep (usually IO)
R	running or runnable (on run queue)
S	interruptible sleep (waiting for an event to complete)
T	stopped by job control signal
Z	defunct ("zombie") process, terminated but not reaped by its parent

- There are two sleep (waiting) states: uninterruptible and interruptible. A process enters a sleeping state when it needs a resource that is currently unavailable.
- **S**: interruptible sleep (e.g. the **sleep** syscall)
The process is waiting for some condition to exist. When the condition is met, the process will come out of this state. It can also wake up prematurely and become runnable if it receives a signal.
- **D**: uninterruptible sleep
Similar to **S**, except the process won't immediately handle a signal (in other words, it won't wake up if it receives a signal). It wakes up after the condition is met or after a time-out occurs during that wait (if specified when put to sleep). This state is used if the condition is expected to be met quickly, or when the process must wait without interruption. When a process is in this state, any signals accumulated during the sleep are noticed when the process returns from the system call or trap. Of the two, this state is used less frequently because it doesn't respond to signals.

ps State Codes

```
D    uninterruptible sleep (usually IO)
R    running or runnable (on run queue)
S    interruptible sleep (waiting for an event to complete)
T    stopped by job control signal
Z    defunct ("zombie") process, terminated but not reaped by its parent
```

- **T**: stopped

This state means that the process was stopped as a result of either being debugged via a trace or receiving a signal like SIGSTOP or SIGTSTP (CTRL-Z)

ps State Codes

```
D    uninterruptible sleep (usually IO)
R    running or runnable (on run queue)
S    interruptible sleep (waiting for an event to complete)
T    stopped by job control signal
Z    defunct ("zombie") process, terminated but not reaped by its parent
```

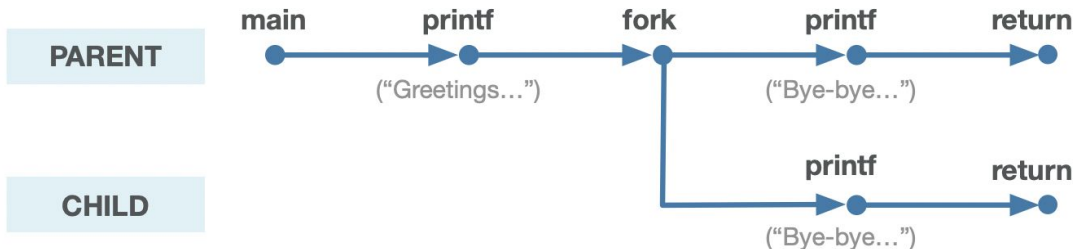
- **Z**: zombie

This state means that the process is terminated (meaning it finished as a child process), but it hasn't yet been reaped (cleaned up) by its parent process.

Reintroducing fork

Recap: System Call fork

- A program may decide that it wants to run multiple processes itself. We will see many examples of why a program may want to do this as the course progresses.
- If a program wants to launch a second process, it uses the **fork** system call.
- **fork()** does exactly this:
 - It creates a new process (a **child** process) that starts on the next instruction after the **fork** call. The parent process also continues on the next instruction, as well.
 - A successful **fork** call returns a **pid_t** (an integer) to both processes (and returns -1 on error). Neither is the actual **pid** of the process that receives it:
 - The parent process gets a return value that is the pid of the *child* process.
 - The child process gets a return value of 0, indicating that it is the child.
 - The child process does, indeed, have its own pid, but it would need to call **getpid** itself to retrieve it.
 - **All** memory is identical between the parent and child, though it is **not** shared (it is copied).



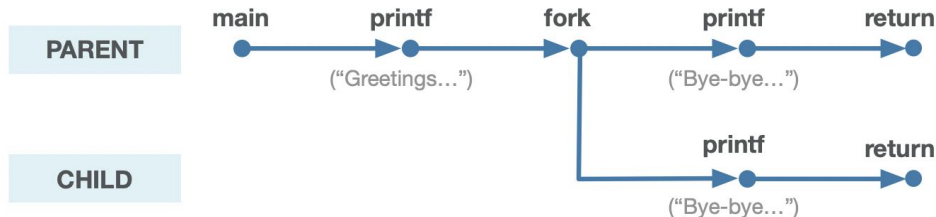
Understanding fork

Here's a simple program that knows how to spawn new processes. It uses system calls named **fork**, **getpid**, and **getppid**. The full program can be viewed [here](#).

```
int main(int argc, char *argv[]) { // basic-fork.c
    printf("Greetings from process %d! (parent %d)\n", getpid(), getppid());
    pid_t pid = fork();
    assert(pid >= 0);
    printf("Bye-bye from process %d! (parent %d)\n", getpid(), getppid());
    return 0;
}
```

- **getpid** and **getppid** return the process id of the caller and the process id of the caller's parent, respectively. Here's the output of the above program.

```
myth54$ echo $$
28740
myth54$ ./basic-fork
Greetings from process 29686! (parent 28740)
Bye-bye from process 29686! (parent 28740)
Bye-bye from process 29687! (parent 29686)
```



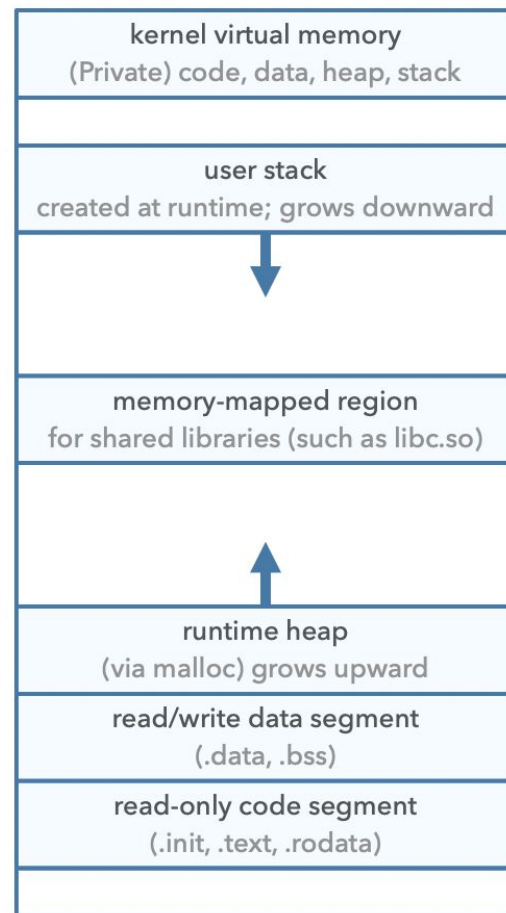
- **echo \$\$** prints the process ID of the current shell.

Understanding fork

```
int main(int argc, char *argv[]) { // basic-fork.c
    printf("Greetings from process %d! (parent %d)\n", getpid(), getppid());
    pid_t pid = fork();
    assert(pid >= 0);
    printf("Bye-bye from process %d! (parent %d)\n", getpid(), getppid());
    return 0;
}
```

fork is called once, but it returns twice.

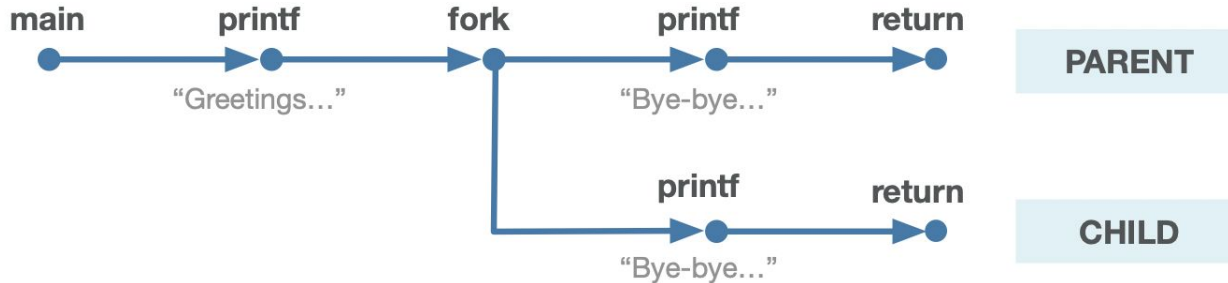
- fork** knows how to clone the calling process, synthesize a nearly identical copy of it (with a new process ID), and schedule the copy as if it were running all along. The child (clone) process gets a separate copy of:
 - The parent's user-level virtual address space (code and data segments, heap, shared libraries, and user stack)
 - All open file descriptors (these copies are donated to the clone). This means the child can read and write any files that were open in the parent when it called **fork**.
- As a result, the output of our above program is the output of two processes.



Understanding fork

```
int main(int argc, char *argv[]) { // basic-fork.c
    printf("Greetings from process %d! (parent %d)\n", getpid(), getppid());
    pid_t pid = fork();
    assert(pid >= 0);
    printf("Bye-bye from process %d! (parent %d)\n", getpid(), getppid());
    return 0;
}
```

- Fork calls can be diagrammed using process graphs (`assert` not shown):



Understanding fork

```
int main(int argc, char *argv[]) { // basic-fork.c
    printf("Greetings from process %d! (parent %d)\n", getpid(), getppid());
    pid_t pid = fork();
    assert(pid >= 0);
    printf("Bye-bye from process %d! (parent %d)\n", getpid(), getppid());
    return 0;
}
```

- Fork calls can be diagrammed using process graphs (**assert** not shown):



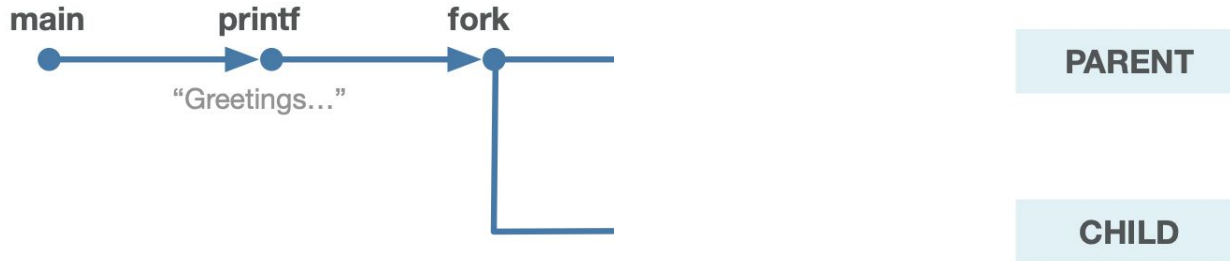
PARENT

CHILD

Understanding fork

```
int main(int argc, char *argv[]) { // basic-fork.c
    printf("Greetings from process %d! (parent %d)\n", getpid(), getppid());
    pid_t pid = fork();
    assert(pid >= 0);
    printf("Bye-bye from process %d! (parent %d)\n", getpid(), getppid());
    return 0;
}
```

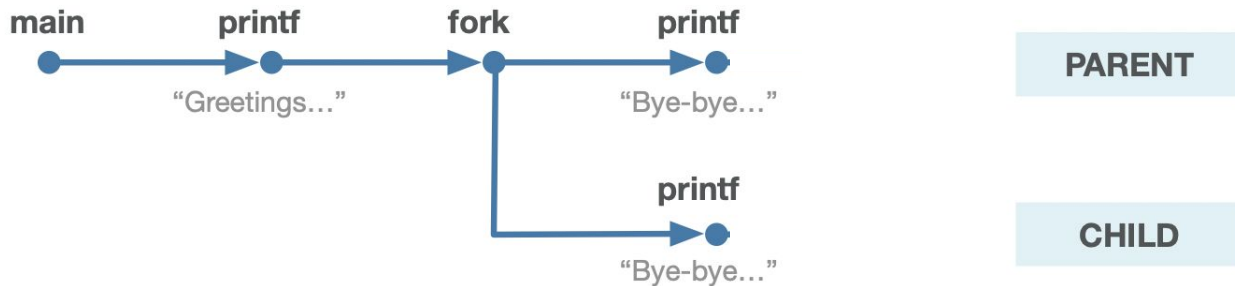
- Fork calls can be diagrammed using process graphs (**assert** not shown):



Understanding fork

```
int main(int argc, char *argv[]) { // basic-fork.c
    printf("Greetings from process %d! (parent %d)\n", getpid(), getppid());
    pid_t pid = fork();
    assert(pid >= 0);
    printf("Bye-bye from process %d! (parent %d)\n", getpid(), getppid());
    return 0;
}
```

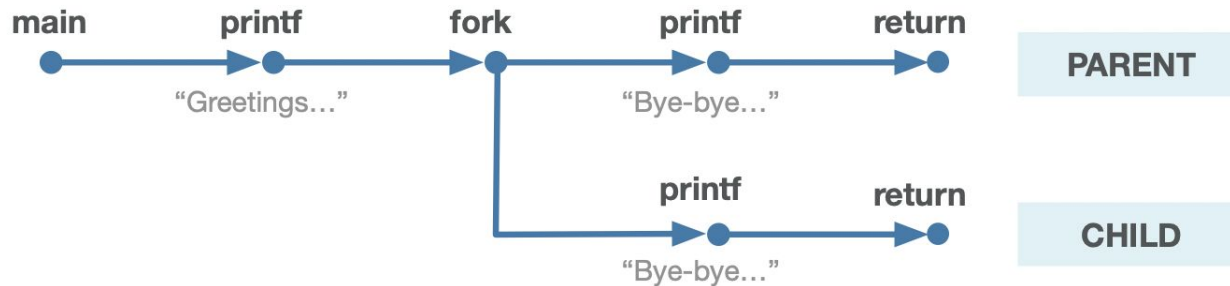
- Fork calls can be diagrammed using process graphs (**assert** not shown):



Understanding fork

```
int main(int argc, char *argv[]) { // basic-fork.c
    printf("Greetings from process %d! (parent %d)\n", getpid(), getppid());
    pid_t pid = fork();
    assert(pid >= 0);
    printf("Bye-bye from process %d! (parent %d)\n", getpid(), getppid());
    return 0;
}
```

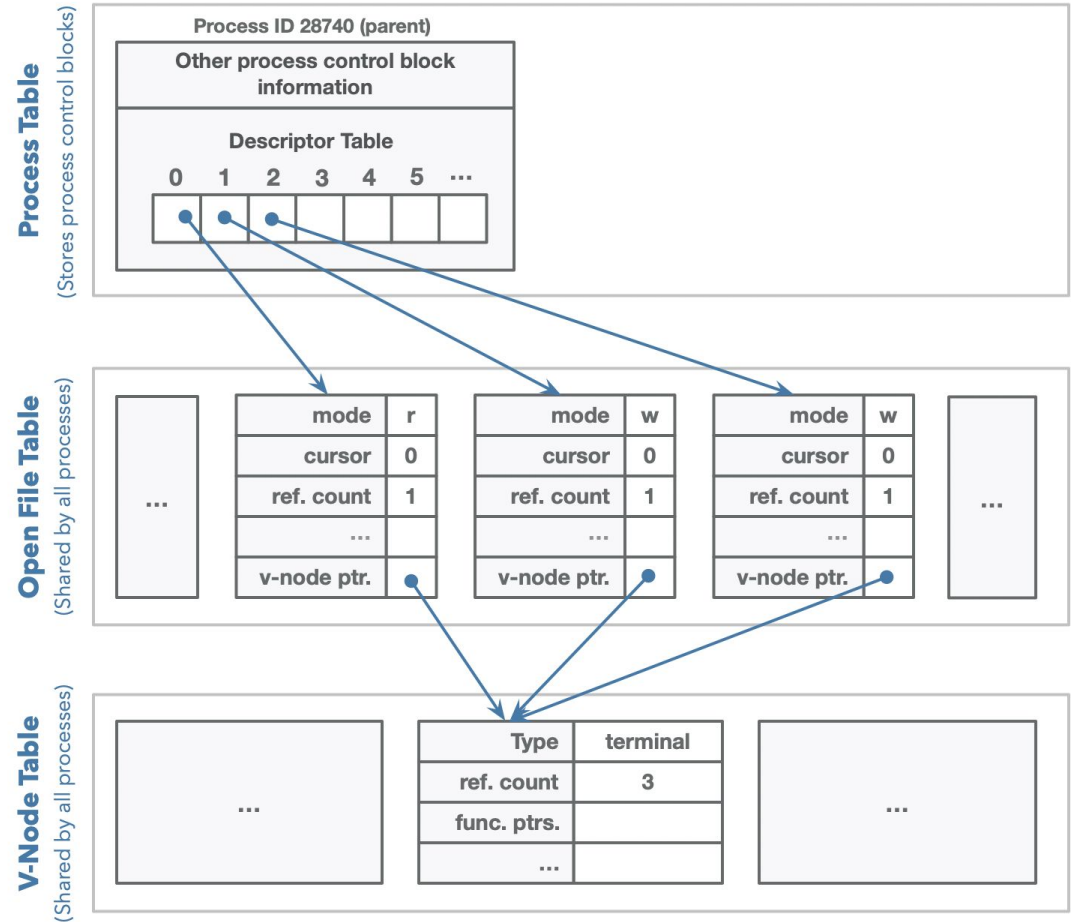
- Fork calls can be diagrammed using process graphs (`assert` not shown):



Question:
Why is the child also able to print to stdout?

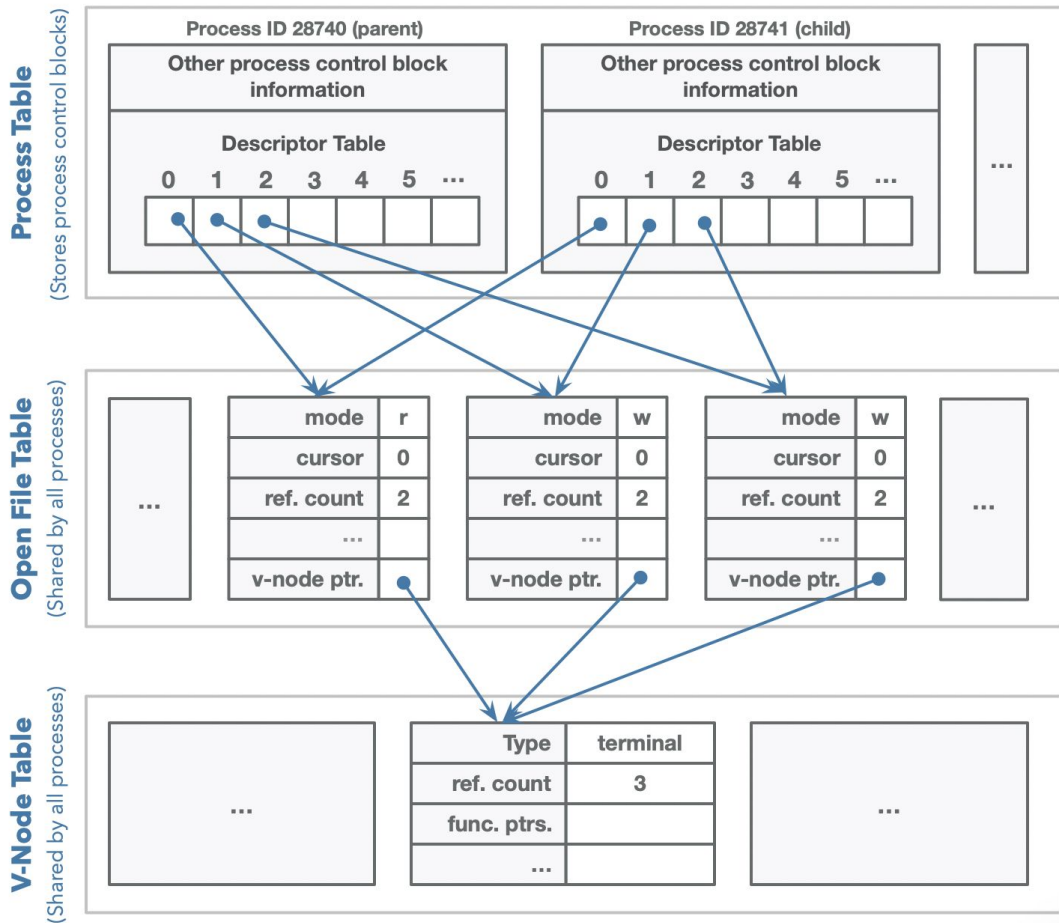
Descriptors and fork

- Answer: recall that the parent process' file descriptor table is cloned on **fork**.



Descriptors and fork

- On **fork**, the reference count in each of the open file table entries that are pointed to by the parent process is incremented by one.
- Thus on **fork**, a child process inherits the stdin, stdout, and stderr linked to the terminal.



Understanding fork

- Differences between parent calling **fork** and child generated by it:
 - The most obvious difference is that each gets a unique process id. That's important. Otherwise, the OS can't tell them apart.
 - Another key difference: **fork**'s return value in the two processes
 - When **fork** returns in the parent process, it returns the pid of the new child.
 - When **fork** returns in the child process, it returns 0. That isn't to say the child's pid is 0, but rather that **fork** elects to return a 0 as a way of allowing the child process to easily self-identify as the child process.
 - The return value can be used to dispatch each of the two processes in a different direction (although in this introductory example, we don't do that).

Understanding fork

```
myth54$ echo $$  
28740  
myth54$ ./basic-fork  
Greetings from process 29686! (parent 28740)  
Bye-bye from process 29686! (parent 28740)  
Bye-bye from process 29687! (parent 29686)
```

- Here's why the program output makes sense:
 - Process IDs are generally assigned consecutively. That's why you see 29686 and 29687.
 - 28740 is the pid of the shell itself, and you can see that the **basic-fork** process—with pid 29686—is a direct child processes of the terminal. The output tells us so.
 - The clone of the original is assigned pid 29687, and the output is clear about the parent-child relationship between 29686 and 29687.
 - You can run this program multiple times to get a better understanding of the process IDs.

Understanding fork

- The parent and child are separate processes that run concurrently.
 - **The instructions in their logical control flows are arbitrarily interleaved by the kernel**, so as programmers, we cannot make assumptions about the order in which the instructions will run in different processes. The output is **nondeterministic**.
- There is **no** default sharing of data between the two processes, though the parent process can **wait** (more on this later) for child processes to complete.
- You can use shared memory to communicate between processes, but this must be explicitly set up before making **fork** calls.

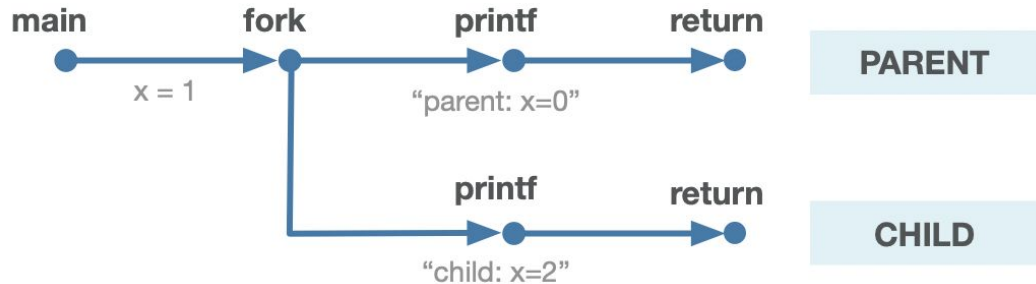
Understanding fork

- Another example:

```
int main() // fork-ints.c
{
    pid_t pid;
    int x = 1;

    pid = fork();
    if (pid == 0) { // Child
        printf("child: x=%d\n", ++x);
        return 0;
    }

    // Parent
    printf("parent: x=%d\n", --x);
    return 0;
}
```



- Since the parent and child are separate processes, they each have their own private address spaces, so any changes to one of the processes will not be reflected in the memory of the other process.

What will the values of x be in each process?

```
myth54$ ./fork-ints
parent: x=0
child: x=2
```

Understanding fork

- Here's trickier example (code [here](#)). Take a look and see if you can understand what would happen.

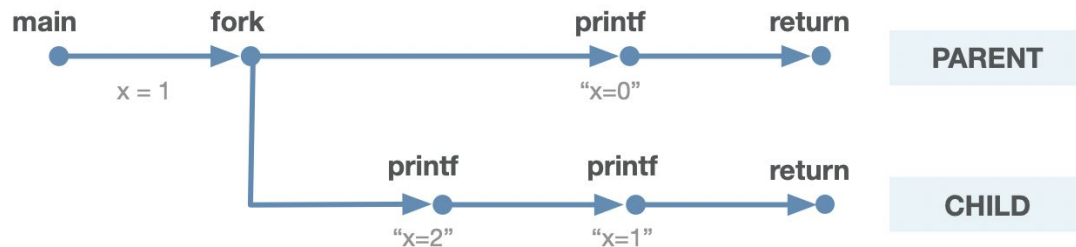
```
int main(int argc, char *argv[]) {
    int x = 1;

    pid_t pid = fork();
    assert(pid >= 0);

    if (pid == 0) { // in child
        printf("x=%d\n", ++x);
    }

    printf("x=%d\n", --x);

    return 0;
}
```



- The catch is that the child actually will execute both print statements since it doesn't return/exit in the `if` statement! Also remember the possible nondeterministic orderings here. Think about a topological sort of the process graph.

What will the values of x be in each process? Answer: In the child: x=2, then x=1. In the parent: x=0.

```
myth54$ ./forkprob0
x=0
x=2
x=1
```

```
myth54$ ./forkprob0
x=2
x=1
x=0
```

```
myth54$ ./forkprob0
x=2
x=0
x=1
```


Another fork Example That Shows Process States

Using ps To See Process States

- You can run the following to see the process ID, state, and command for each process:

```
$ ps -o pid,state,command
  PID S  COMMAND
1157608 S  -bash
1157958 R  ps -o pid,state,command
```

- The command we just ran to get this output is state **R**, which shows the state as running at the time that we executed the command.
- Note: the current shell (bash) is state **S**, which is interruptible sleep. Rather than constantly checking the keyboard for input, the shell puts itself to sleep while waiting on an event, such as an interrupt from the keyboard when it has input for the shell to process.

Bonus Slide: Using ps To See Process States

- To actually see the shell in a running state:

```
myth66$ echo $$
1157608
myth66$ tty
/dev/pts/18
myth66$ while true; do NOTHING=1; done
```

```
myth66$ tty
/dev/pts/22
myth66$ ps -xo pid,state,command,TTY | grep pts/18
1157600 S sshd: rcyrus@pts/18      ?
1157608 R -bash                          pts/18
1226776 S grep pts/18                      pts/22
```

- Left: First we get the terminal that the current shell is running in with **tty**. Then we create an infinite loop that does, well, nothing meaningful, but it keeps bash busy.
- Right: Next, in a separate terminal that is connected to the same myth machine, we run the **ps** command to get info about the bash process that is connected to the first terminal we ran the while loop on. It shows that bash is running! 🎉
- Remember to hit CTRL-C in first terminal to stop the while loop 😊

Using ps To See Process States

Here's an example that has the parent sleep for a while before exiting so we can see what happens when the child exits before its sleepy parent.

```
#include <stdio.h>           // for printf
#include <unistd.h>         // for fork, getpid, getppid

int main(int argc, char *argv[]) {
    printf("PARENT: process %d (parent %d)\n", getpid(), getppid());
    pid_t pid = fork();
    if (pid == 0) {
        printf("CHILD: process %d (parent %d) - time to eat!\n", getpid(), getppid());
        sleep(10);
        printf("CHILD done eating!\n");
        return 0;
    }
    printf("PARENT (process %d): I'm going to sleep for a bit before cleaning up after my kid...\n", getpid());
    sleep(20);
    printf("PARENT exiting!\n");
    return 0;
}
```

Using ps To See Process States

Results shown across two terminals:

```
myth66$ ./slowparent
PARENT: process 1236838 (parent 1157608)
PARENT (process 1236838): I'm going to sleep
for a bit before cleaning up after my kid...
CHILD: process 1236839 (parent 1236838) - time
to eat!
CHILD done eating!
PARENT exiting!
```

```
myth66$ tty
/dev/pts/22
myth66$ ps -xo pid,state,command,TTY | grep pts/18
1157600 S sshd: rcyrus@pts/18      ?
1157608 S -bash                          pts/18
1236838 S ./slowparent                    pts/18
1236839 Z [slowparent] <defunct>    pts/18
1236847 S grep pts/18                     pts/22
```

- Left: First we run the program and get the expected print outs. The parent process goes to sleep so we have time to check the process states in another terminal, shown on right. We must do this before the ‘PARENT exiting!’ line appears after ~20 seconds.
- Right: Next, in a separate terminal that is connected to the same myth machine, we run the **ps** command to get info about the **slowparent** parent and child processes that are running in our first terminal. It shows that the parent is sleeping (**S**) and the child is a zombie (in the **Z** state), as expected! 🎉🎉

Using ps To See Process States

Let's check for one more state...Run the program again:

```
myth66$ ./slowparent
PARENT: process 1237813 (parent 1157608)
PARENT (process 1237813): I'm going to sleep
for a bit before cleaning up after my kid...
CHILD: process 1237814 (parent 1237813) - time
to eat!
CHILD done eating!
^Z
[1]+  Stopped                  ./slowparent
```

```
myth66$ ps -o pid,state,command,TTY | grep slowparent
1237813 T ./slowparent pts/18
1237814 Z [slowparent] <defunct> pts/18
1238443 S grep slowparent pts/18
```

- Left: We run the program again, but then press CTRL-Z to stop the process right after until child is “done eating” (at which point it exits).
- Right: Next, we run the **ps** command to get info about the **slowparent** parent and child processes. It shows that the parent is stopped (**T**) and the child is a zombie (in the **Z** state), as expected! 🎉🎉🎉
- If we enter **fg** into the first terminal, the process will continue and then exit.

Using lsof To See File Descriptors

- To show that the child gets a copy of the parent's file descriptors, let's view the file descriptors for the **slowparent** program using **lsof**:

```
myth66$ ./slowparent
PARENT: process 1444075 (parent 1441382)
PARENT (process 1444075): I'm going to sleep for a bit before
cleaning up after my kid...
CHILD: process 1444076 (parent 1444075) - time to eat!
^Z

[1]+  Stopped                  ./slowparent
```

```
myth66$ lsof /dev/pts/18
[...]
COMMAND      PID    USER   FD    TYPE  DEVICE  SIZE/OFF  NODE  NAME
bash         1441382 rcyrus  0u    CHR  136,18    0t0   21  /dev/pts/18
bash         1441382 rcyrus  1u    CHR  136,18    0t0   21  /dev/pts/18
bash         1441382 rcyrus  2u    CHR  136,18    0t0   21  /dev/pts/18
bash         1441382 rcyrus 255u   CHR  136,18    0t0   21  /dev/pts/18
slowparent   1444075 rcyrus  0u    CHR  136,18    0t0   21  /dev/pts/18
slowparent   1444075 rcyrus  1u    CHR  136,18    0t0   21  /dev/pts/18
slowparent   1444075 rcyrus  2u    CHR  136,18    0t0   21  /dev/pts/18
slowparent   1444076 rcyrus  0u    CHR  136,18    0t0   21  /dev/pts/18
slowparent   1444076 rcyrus  1u    CHR  136,18    0t0   21  /dev/pts/18
slowparent   1444076 rcyrus  2u    CHR  136,18    0t0   21  /dev/pts/18
lsof         1444499 rcyrus  0u    CHR  136,18    0t0   21  /dev/pts/18
lsof         1444499 rcyrus  1u    CHR  136,18    0t0   21  /dev/pts/18
lsof         1444499 rcyrus  2u    CHR  136,18    0t0   21  /dev/pts/18
```

- We hit CTRL-Z before the child prints “done eating” so that we can check the entries for both the parent and the child before they exit. You can see on the right that the parent and the child indeed both have fd 0, 1, and 2 open.
- Question: What other parent/child relationship do you see in the output shown above on the right?**

Using ps To See Process States

We've demonstrated how to view all the process states that we care about. What about the **D** (uninterruptible) state?

- This state is harder to reproduce because processes that end up in this state really shouldn't take that long to begin with, else they end up stuck there with no way to be killed (because they can't be interrupted, they can't receive signals to kill them!). But just believe me when I say that this state is possible!

Understanding fork

- Yet another example: a tree of fork calls (the full program can be viewed [here](#)):

```
static const char const *kTrail = "abcd";

int main(int argc, char *argv[]) {
    size_t trailLength = strlen(kTrail);

    for (size_t i = 0; i < trailLength; i++) {
        printf("%c\n", kTrail[i]);
        pid_t pid = fork();
        assert(pid >= 0);
    }

    return 0;
}
```

- While you rarely have reason to use **fork** this way, it's instructive to trace through a short program where spawned processes themselves call **fork**.

What kind of output do you expect?

Understanding fork

- Two samples runs are shown on the right.
- Reasonably obvious: A single **a** is printed by the soon-to-be-great-great-granddaddy process.
- Less obvious: The first child and the parent each return from **fork** and continue running in mirror processes, each with their own copy of the global "**abcd**" string, and each advancing to the **i++** line within a loop that promotes a 0 to 1. It's hopefully clear now that two **b**'s will be printed, and this pattern will continue until the end.
- Key questions to answer:
 - **Why aren't the two b's always consecutive?**
 - **How many c's get printed?**
 - **How many d's get printed?**
 - **Why is there a shell prompt in the middle of the output of the second run on the right?**

```
myth60$ ./fork-puzzle
```

```
a
b
c
b
d
c
d
c
c
d
d
d
d
d
d
d
```

```
myth60$
```

```
myth60$ ./fork-puzzle
```

```
a
b
b
c
d
c
d
c
d
d
c
d
d
myth60$ d
d
d
```


Introducing `waitpid`

Waiting For Children to Finish

- Notice in the fork puzzle example that it was possible for the **d**'s to be printed after the prompt. This is because the parent finishes before the child, but the child still has access to stdout and continues printing its data.
- Synchronization between parent and child can be done by using the system call **waitpid**. It can be used to temporarily block a process until a child process terminates or stops.

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

- The first argument specifies the wait set, which for the moment is just the ID of the child process that needs to complete before **waitpid** can return.
- The second argument supplies the address of an integer where termination information can be placed (or we can pass in **NULL** if we don't care for the information).
- The third argument is a collection of bitwise-or'ed flags we'll study later. For the time being, we'll just go with 0 as the required parameter value, which means that **waitpid** should only return when a process in the supplied wait set exits.
- The return value is the pid of the child that exited, or -1 if **waitpid** was called and there were no child processes in the supplied wait set.

Waiting For Children to Finish

Consider the following program, which is more representative of how **fork** really gets used in practice (full program, with error checking, is [right here](#)):

```
int main(int argc, char *argv[]) {
    printf("Before.\n");
    pid_t pid = fork();
    printf("After.\n");
    if (pid == 0) {
        printf("I am the child, and the parent will wait up for me.\n");
        return 110; // contrived exit status
    } else {
        int status;
        waitpid(pid, &status, 0);
        if (WIFEXITED(status)) {
            printf("Child exited with status %d.\n", WEXITSTATUS(status));
        } else {
            printf("Child terminated abnormally.\n");
        }
        return 0;
    }
}
```

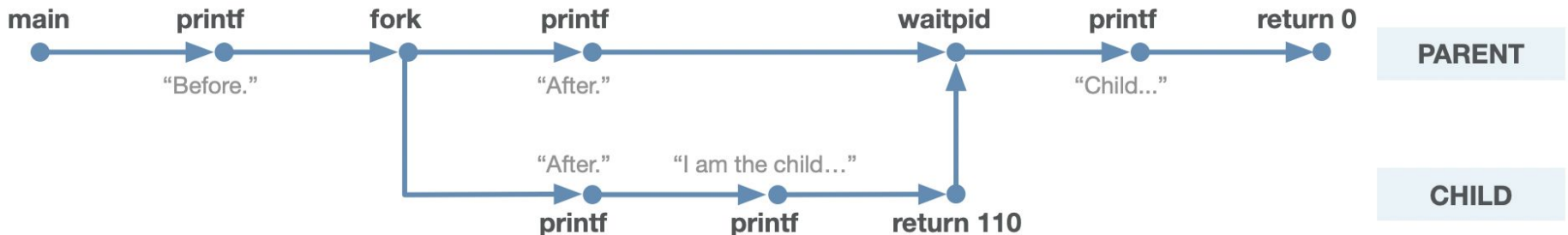
- The parent process correctly waits for the child to complete using **waitpid**.
- The parent lifts child exit information out of the **waitpid** call, and uses the **WIFEXITED** macro to examine some high-order bits of its argument to confirm the process exited normally, and it uses the **WEXITSTATUS** macro to extract the lower eight bits of its argument to produce the child return value (which is 110 as expected).
- The **waitpid** call also donates child process-oriented resources back to the system.

Waiting For Children to Finish

The output is likely what is shown on the left below every single time the program is executed, since the parent likely continues running without halting when it calls fork (since all fork does is set up new data structures for a new process, returns, and then carries on). However, it is theoretically possible to get the output on the right if the child runs first:

```
myth60$ ./separate
Before.
After.
After.
I am the child, and the parent will wait up for me.
Child exited with status 110.
myth60$
```

```
myth60$ ./separate
Before.
After.
I'm the child, and the parent will wait up for me.
After.
Child exited with status 110.
myth60$
```



Debugging Two Processes

- You might be asking yourself, *How do I debug two processes at once?* This is a very good question! **gdb** has built-in support for debugging multiple processes, as follows:
 - **set detach-on-fork off**
 - This tells **gdb** to capture any **fork**'d processes, though it pauses them upon the **fork**.
 - **info inferiors**
 - This lists the processes that **gdb** has captured.
 - **inferior X**
 - Switch to a different process to debug it.
 - **detach inferior X**
 - Tell **gdb** to stop watching the process, and continue it
 - You can see an entire debugging session on the basic-fork program [right here](#).

The Point Of It All

- So, now we know how to create processes... but why would we do that in the first place? There are a few major reasons:
 - performance (ability to use multiple CPUs)
 - security (isolation of possibly sensitive components of an application)
 - Next time, we'll talk about a third reason: starting executables from disk.

End of Lecture 5

