CSE 120
Principles of Operating Systems

Spring 2023

Lecture 3: Processes

Amy Ousterhout

Administrivia

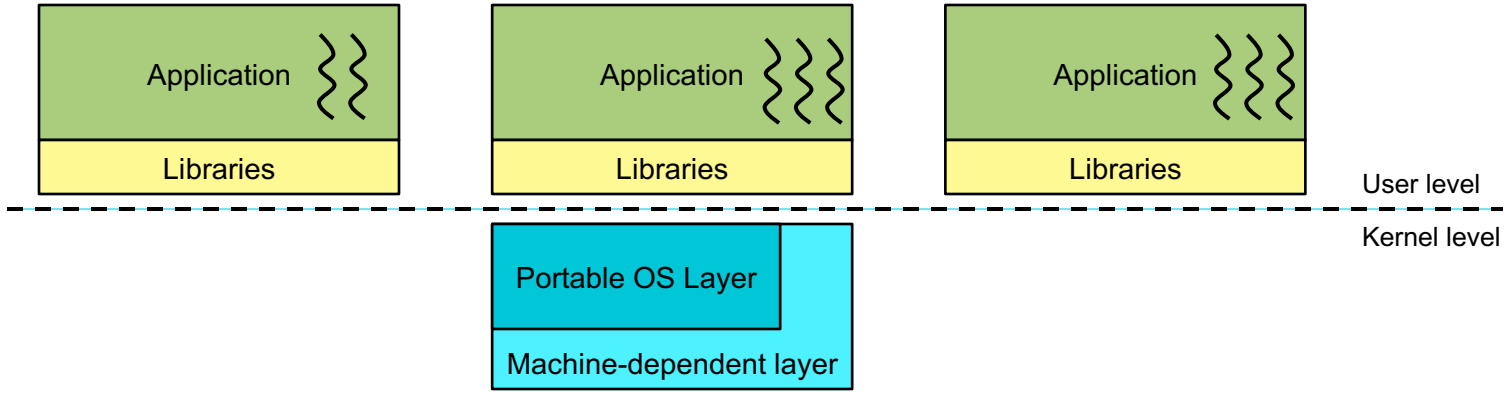
- Project 0
  - Due 4/11 11:59pm (today), done individually
- Homework #1
  - Due 4/18 11:59 pm (next Tuesday), submit via Gradescope
- Project 1
  - Released today, due 4/25
  - Start early!
- Course Feedback #FinAid
  - On Canvas, due Friday 4/14 11:59 pm

Review Question: Events

- How are interrupts (e.g., timer interrupts) similar to exceptions (faults and system calls)?
  - Handled by trapping to the operating system
  - Handler runs in kernel mode
  - Hardware saves important state (PC, registers, etc.)
- How are interrupts different from exceptions?
  - Interrupts are caused by external events (asynchronous) whereas exceptions are caused by a program executing instructions (synchronous)

Hint, consider:
- What causes them?
- Where are they handled?
- What is the role of hardware?

Next Several Lectures



- What if I want to run multiple applications? processes (today)
- What if I want to perform multiple tasks within one application? threads (Thursday)
- How can different tasks running concurrently safely share state? synchronization

Today's Outline

- Processes
  - How can we represent a running program?
- Process management
  - How can we schedule different processes on the CPU?
  - What APIs can we use to interact with processes?

The Process

- The process is the OS abstraction for a running program
  - Used to manage execution, scheduling, and other resources
- Simplest process: sequential process
  - Everything happens sequentially
  - One instruction at a time

Process Components

- A process contains all state for a program in execution
  - A memory address space
  - The code for the executing program
  - The data for the executing program
  - An execution stack encapsulating the state of procedure calls
  - The program counter (PC) indicating the next instruction
  - A set of general-purpose registers with current values
  - A set of operating system resources
    - » Open files, network connections, etc.
- A process is named using its process ID (PID)

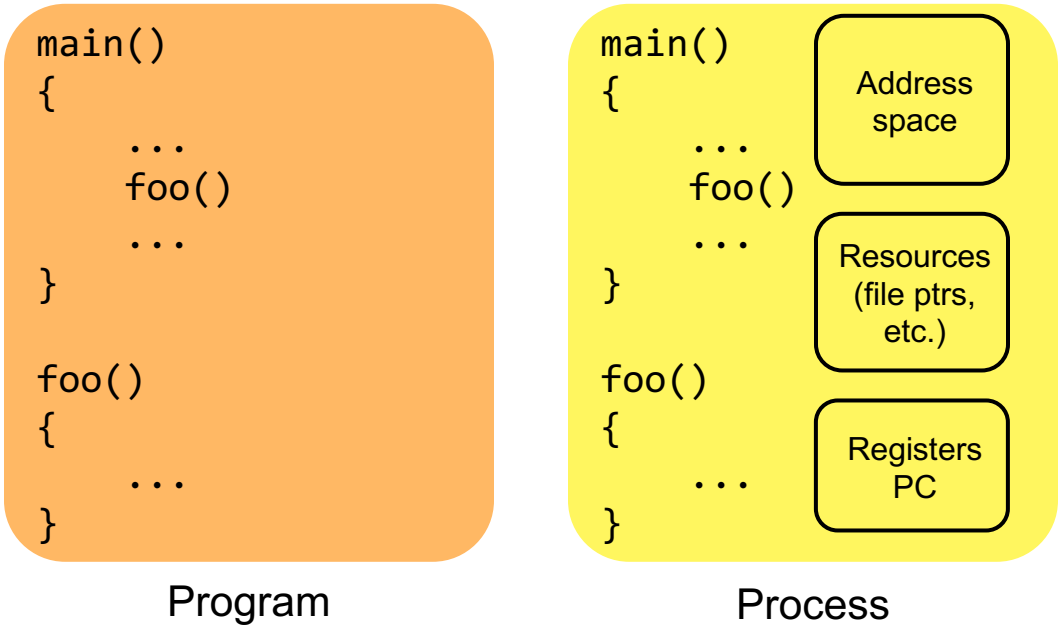
Unix PIDs

- Command: `top`

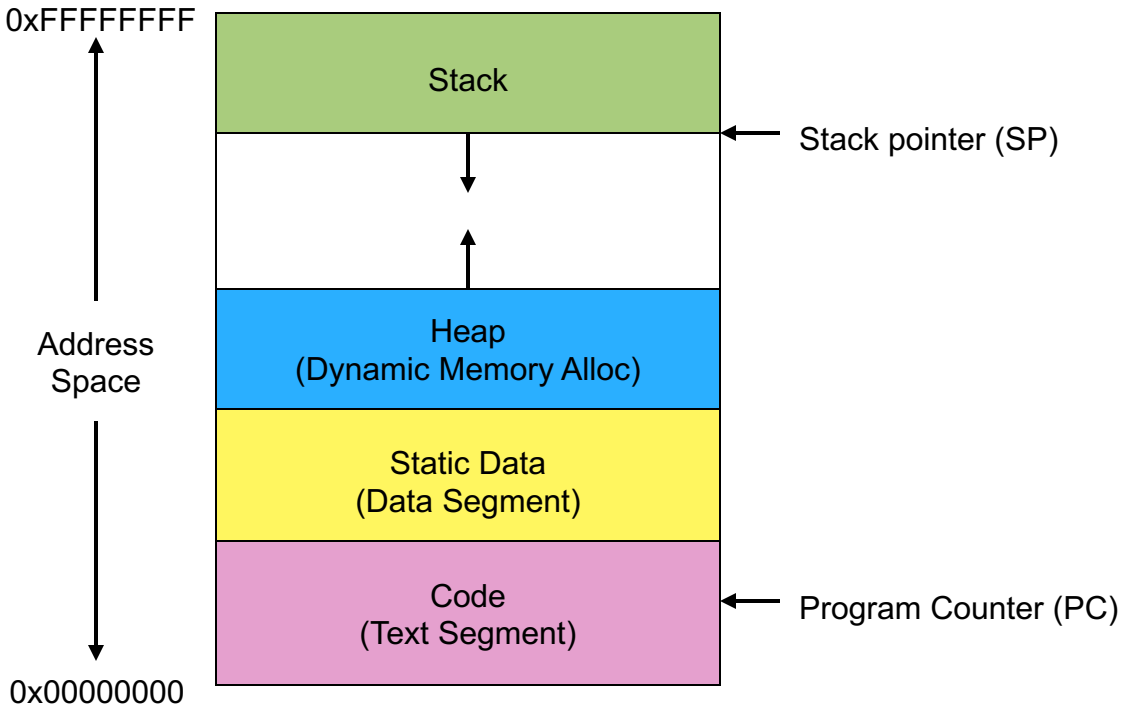
```
Processes: 749 total, 2 running, 747 sleeping, 4932 threads                                                
```

Process vs. Program

- A process is an instance of a program in execution



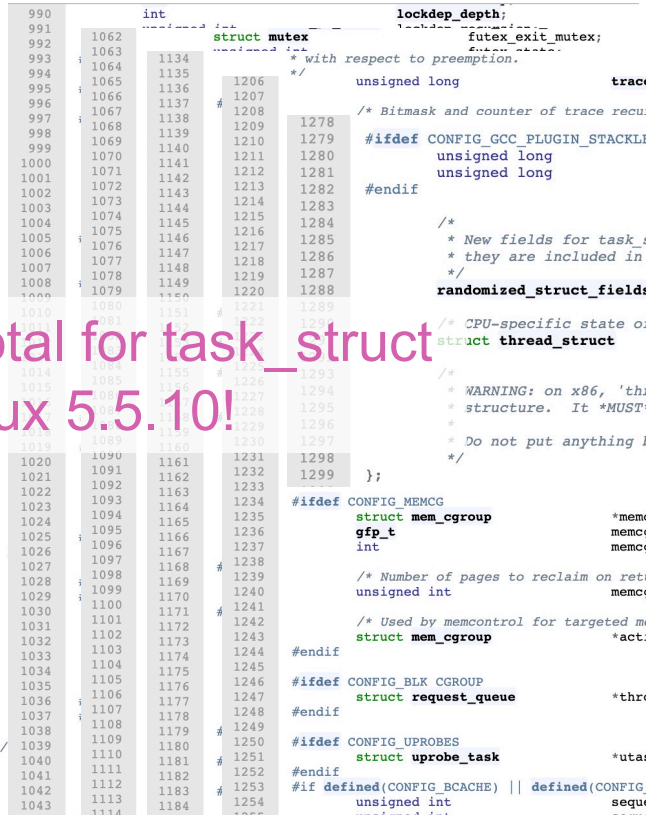
Basic Process Address Space



Process Data Structure

- Many processes running simultaneously
- How does the OS represent a process in the kernel?
- Process Control Block (PCB)
  - Contains all of the information about a process
  - Memory management information
  - Process management information
  - I/O and file management
- It is a **heavyweight** abstraction

PCB in Linux



670 LOC in total for task_struct in Linux 5.5.10!

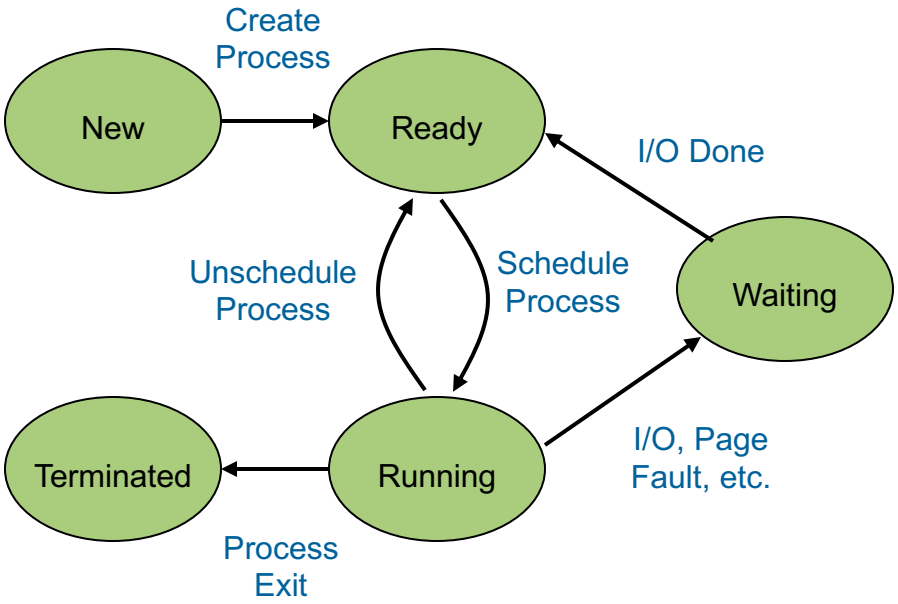
Today's Outline

- Processes
  - How can we represent a running program?
- Process management
  - How can we schedule different processes on the CPU?
  - What APIs can we use to interact with processes?

Process State

- A process has an execution state that indicates what it is currently doing
  - Running: executing instructions on the CPU
    - » It is the process that has control of the CPU
  - Ready: waiting to be assigned to the CPU
    - » Ready to execute, but another process is executing on the CPU
  - Waiting (blocked): waiting for an event, e.g., I/O completion
    - » It cannot make progress until an event occurs (e.g., disk read completes)
- As a process executes, it moves from state to state

Process State Graph



PCBs and Hardware State

- When a process is running, its hardware state (PC, SP, regs, etc. is in the CPU)
  - Hardware registers contain the current values
- When the OS stops running a process, it saves the registers into the processes' PCB
- When the OS resumes running a process, it loads the registers from the values store that process' PCB
- Context switch – the process of changing the CPU hardware state from one process to another
  - As often as every millisecond!

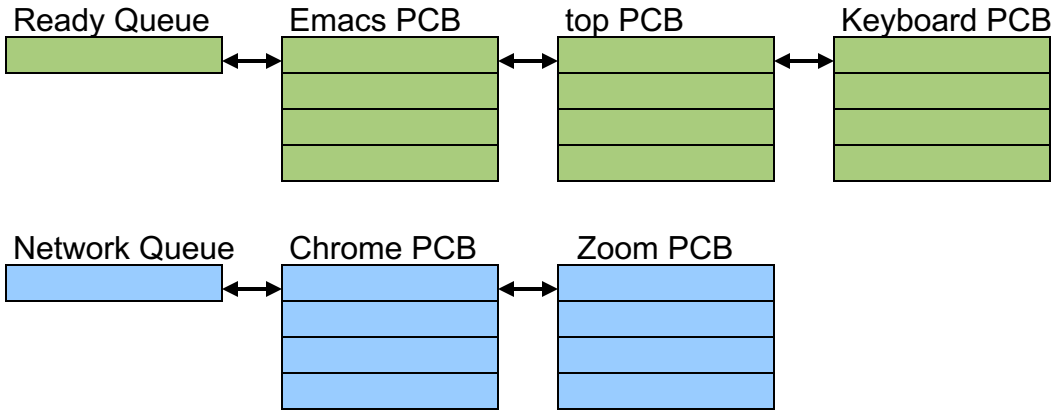
The Processing Illusion

- Every process thinks it owns the CPU, but:
  - With 1 CPU, all processes share the same physical CPU
  - With multiple CPUs, processes share the multiple CPUs
- How is this possible?
  - Timer interrupts
  - Process control block – holds execution state
  - Scheduling (we'll talk more about this later)

State Queues

- How does the OS keep track of processes?
- The OS maintains a collection of queues
  - Running queue
  - Ready queue – processes that are ready to run
  - Waiting queues
- Each PCB is queued on a state queue according to its current state
  - When a process changes state, the OS unlinks its PCB from one queue and links it into another

State Queues



Disk I/O Queue

Sleep Queue

.

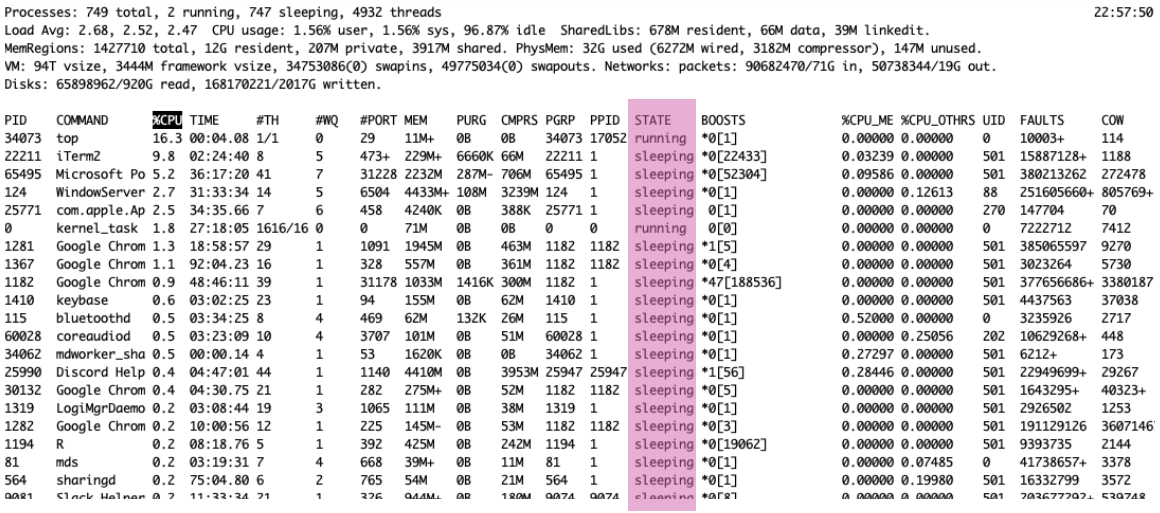
.

.

There may be many wait queues, one for each type of wait (disk, timer, network, synchronization, etc.)

Questions about Processes

- How many processes can be in the running state simultaneously?
- What state do you think a process is in most of the time?
- How many processes can a system support?



```
Processes: 749 total, 2 running, 747 sleeping, 4932 threads                                                
```

Today's Outline

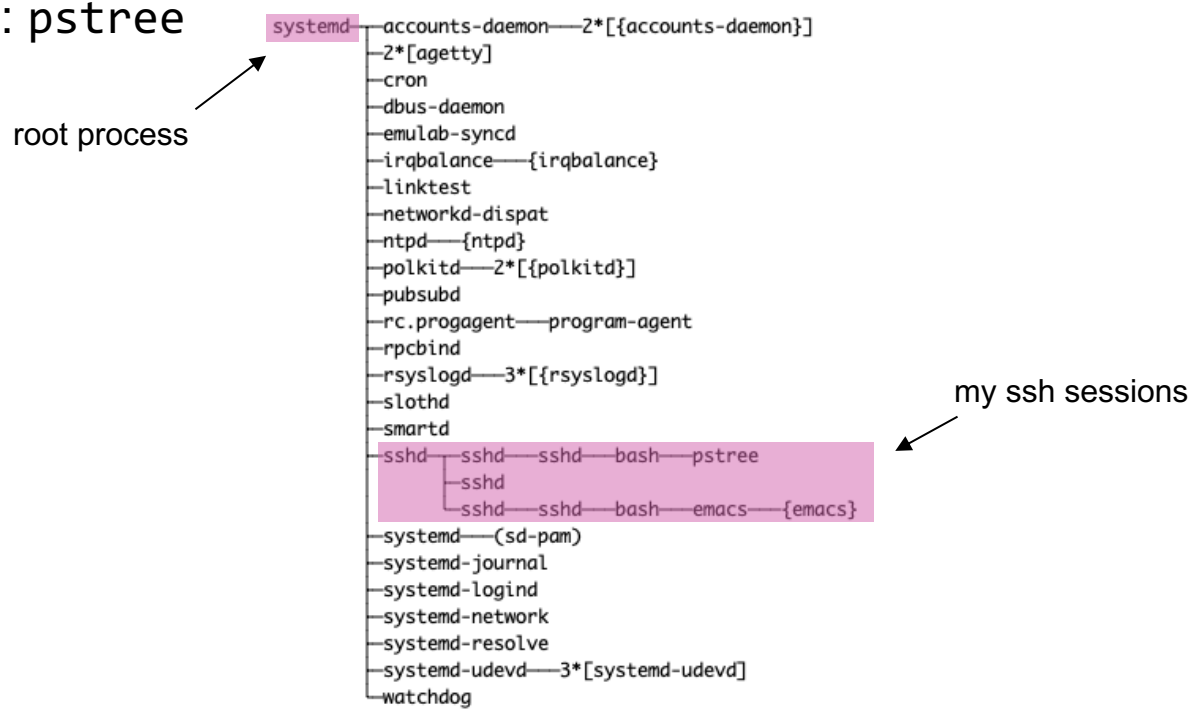
- Processes
  - How can we represent a running program?
- Process management
  - How can we schedule different processes on the CPU?
  - What APIs can we use to interact with processes?

Process Creation

- Every process is created by another process
  - Parent process creates a child process using a system call
- Child inherits some properties from parent
  - Unix: process user ID – children execute with your privileges
- After creating a child, the parent may either wait for it to finish its task or continue in parallel
- The OS creates the first process
  - Unix: init (PID 1)

Process Tree Example

- Command: `pstree`



Process Creation API

- What should the system call for creating a new process look like?
  - Create from scratch
  - Clone from an existing process

Process Creation with `CreateProcess` (Windows)

- System call on Windows for creating a process:
  - `BOOL CreateProcess(char *prog, char *args)` (simplified)
- `CreateProcess`
  - Creates and initializes a new PCB
  - Creates and initializes a new address space
  - Loads the program specified by `prog`
  - Copies `args` into the memory allocated in the address space
  - Initializes the saved hardware context
  - Places the PCB on the ready queue

Process Creation with `CreateProcess` (Windows)

CreateProcessA function (processthreadsapi.h)

Article • 09/23/2022 • 13 minutes to read

Creates a new process and its primary thread. The new process runs in the security context of the calling process.

If the calling process is impersonating another user, the new process uses the token for the calling process, not the impersonation token. To run the new process in the security context of the user represented by the impersonation token, use the CreateProcessAsUser or CreateProcessWithLogonW function.

Syntax

C++ Copy

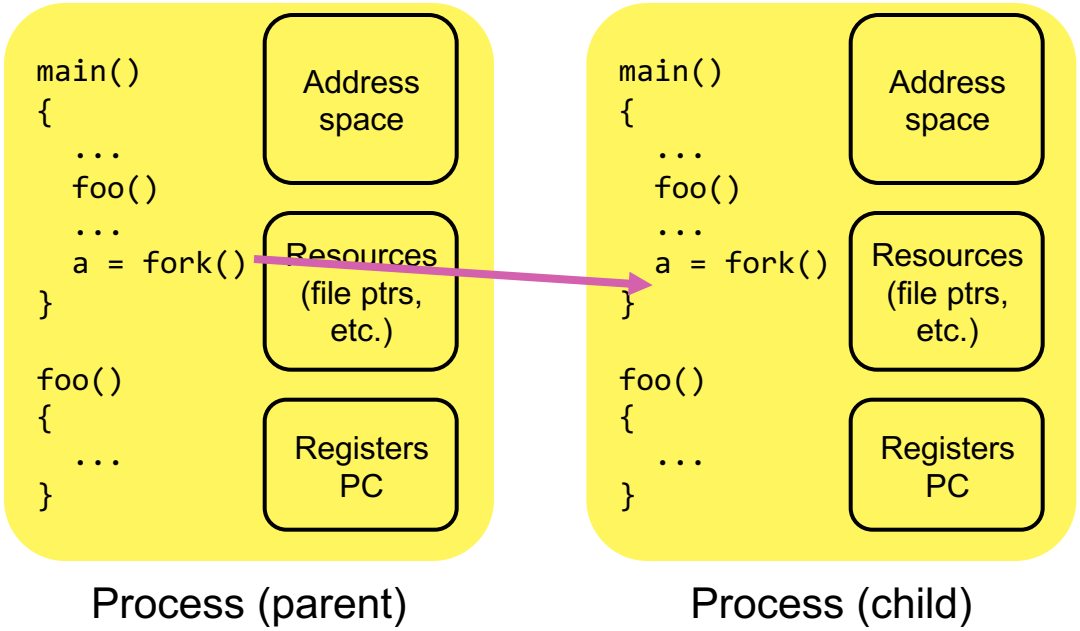
```
BOOL CreateProcessA(
  [in, optional]      LPCSTR                lpApplicationName,
  [in, out, optional] LPSTR                 lpCommandLine,
  [in, optional]      LPSECURITY_ATTRIBUTES lpProcessAttributes,
  [in, optional]      LPSECURITY_ATTRIBUTES lpThreadAttributes,
  [in]                BOOL                  bInheritHandles,
  [in]                DWORD                 dwCreationFlags,
  [in, optional]      LPVOID                lpEnvironment,
  [in, optional]      LPCSTR                lpCurrentDirectory,
  [in]                LPSTARTUPINFOA        lpStartupInfo,
  [out]               LPPROCESS_INFORMATION lpProcessInformation
);
```

Process Creation with `fork` (Unix)

- System call in Unix for creating a process:
  - `int fork()` (NOT simplified)
- `fork()`
  - Creates and initializes a new PCB
  - Creates and initializes a new address space
  - Initializes the address space with a copy of the entire contents of the address space of the parent
  - Initializes the kernel resources to point to the resources used by the parent (e.g., open files)
  - Initialize hardware context to be a copy of the parents'
  - Places the PCB on the ready queue

Process Creation with `fork` (Unix)

- `fork()` system call creates a duplicate* of the original process
  - `fork()` returns twice!
  - Returns the child's PID to the parent
  - Returns "0" to the child



fork()

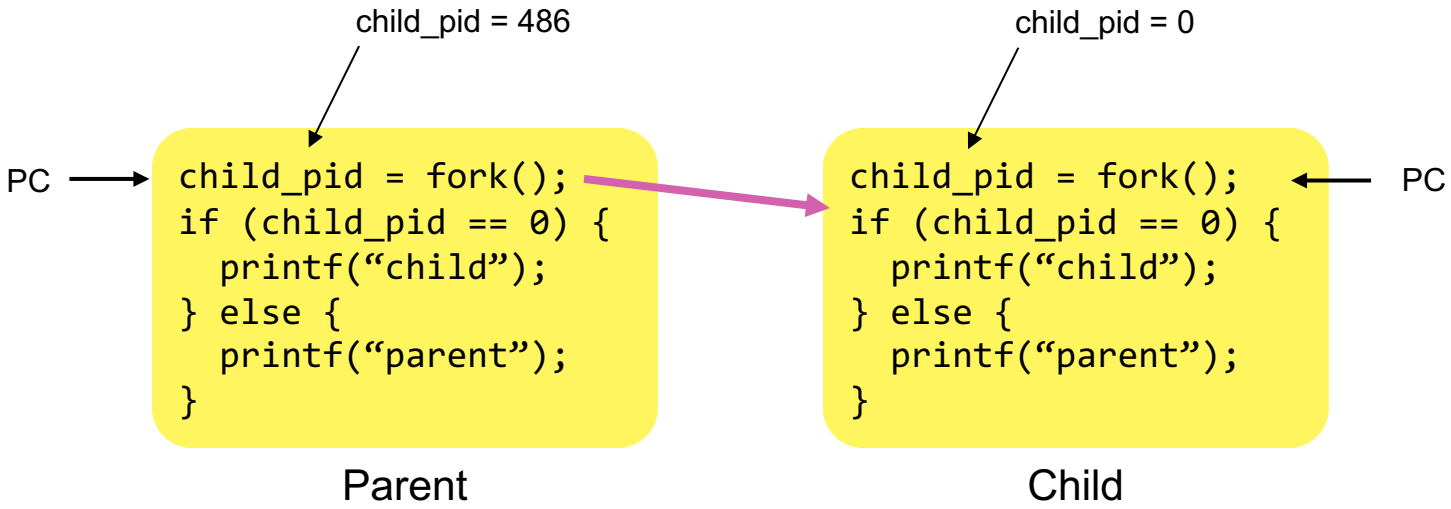
```
int main()
{
   int child_pid = fork();
   if (child_pid == 0) {
          printf("I am the child, my PID is: %d\n", getpid());
          return 0;
   } else {
          printf("My child's PID is: %d\n", child_pid);
          return 0;
   }
}
```

What does this program print?

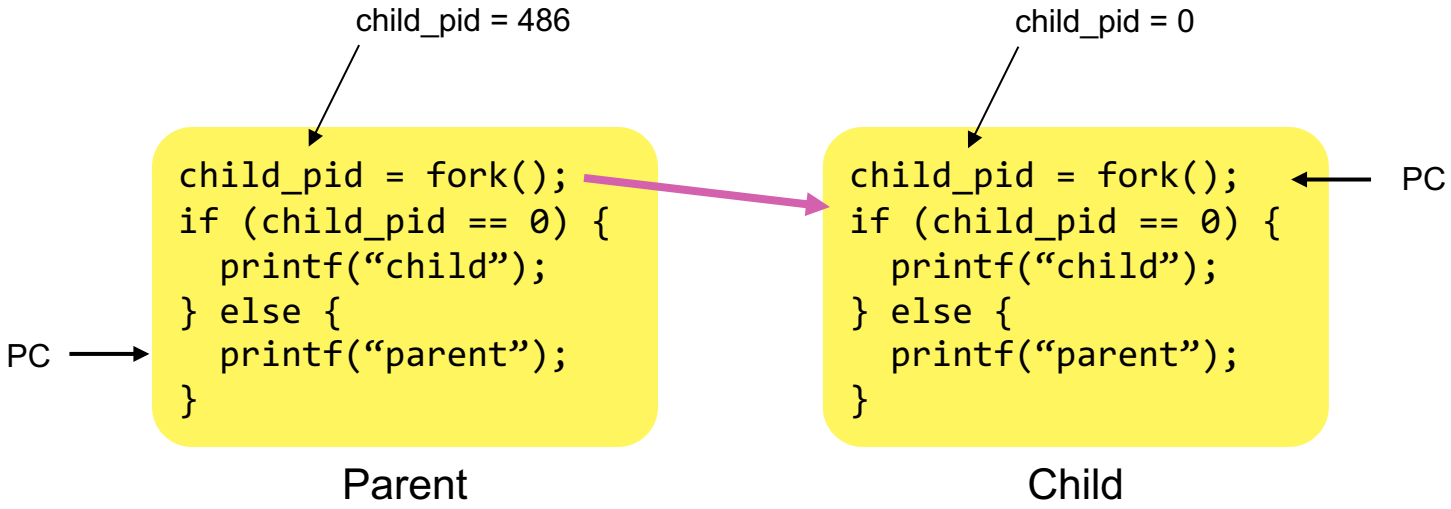
Example output:

```
$ ./a.out
My child's PID is: 35220
I am the child, my PID is: 35220
```

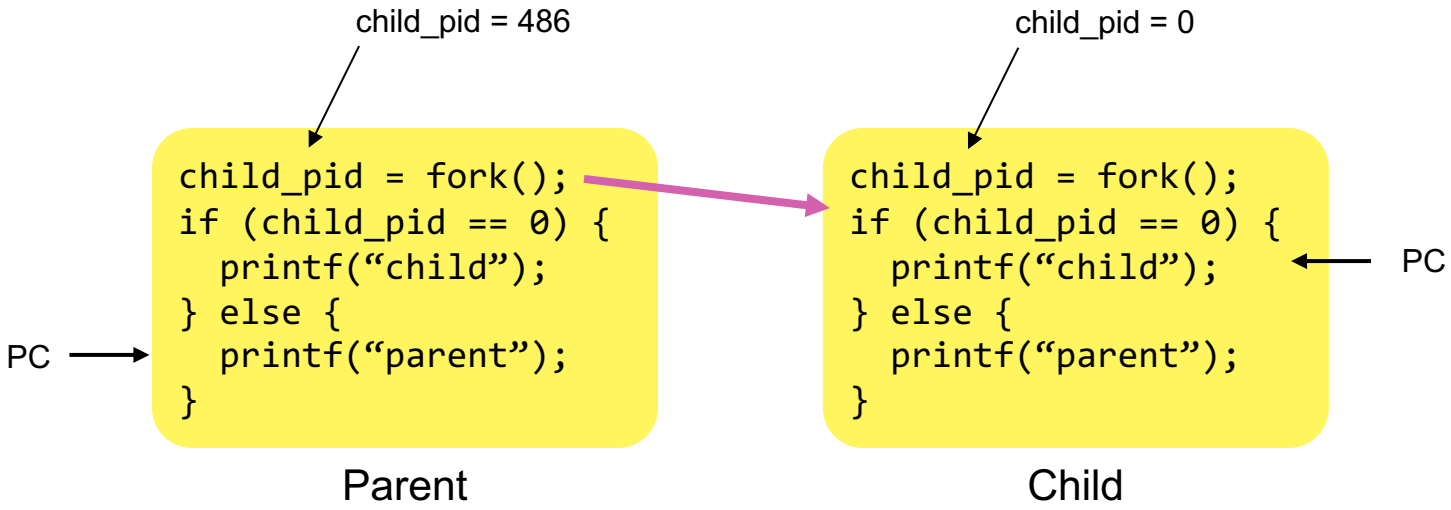
Duplicating and Diverging Address Spaces



Duplicating and Diverging Address Spaces



Duplicating and Diverging Address Spaces



`fork()` example continued

- Example output:

  $ ./a.out

  My child's PID is: 35220

  I am the child, my PID is: 35220

- Another example output:

  $ ./a.out

  I am the child, my PID is: 35220

  My child's PID is: 35220

Why fork()?

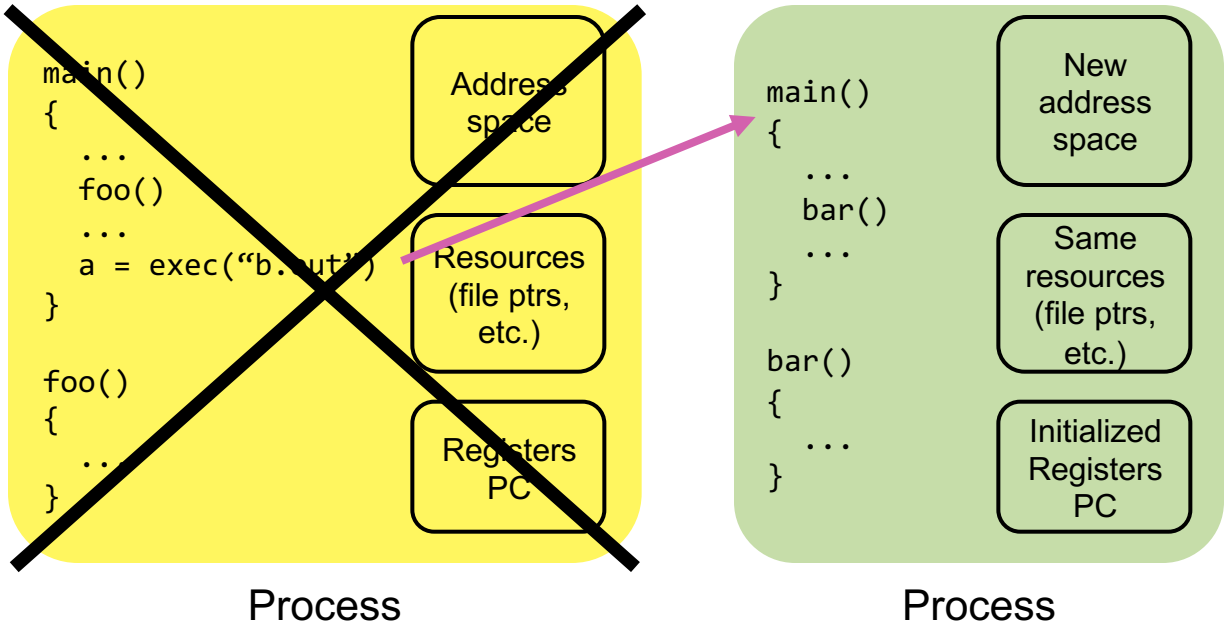
- Very useful when the child…
  - Is cooperating with the parent
  - Relies upon the parent's data to accomplish its task
- Example: Web Server

```
while (1) {
   int sock = accept();
   int child_pid = fork()
   if (child_pid == 0) {
          Handle client request and exit
   } else {
          Continue
   }
}
```

Starting a New Program with exec (Unix)

- System call in Unix for starting a program:
  - `int exec(char *prog, char *argv[])`
- exec()
  - Stops the current process
  - Loads the program prog into the process' address space
  - Initializes hardware context and args for the new program
  - Places the PCB onto the ready queue
  - Note: it does not create a new process
  - Files remain open
- Can exec ever return?
  - Yes, if there's an error

Starting a New Program with exec (Unix)



Process Termination

- All good processes must come to an end. But how?
  - Unix: `exit(int status)`, Windows: `ExitProcess(int status)`
- The OS frees resources and terminates the process
  - Closes open files, network connections
  - Releases allocated memory
  - Terminates all threads (next lecture)
  - Removes PCB from kernel data structures, delete
- Note that a process does not need to clean itself up
  - Why does the OS have to do it?

`wait()` a second…

- Often it is convenient to pause until a child process has finished
  - E.g., executing commands in a shell
- Unix: `wait()`, (Windows: `WaitForSingleObject()`)
  - Suspends the current process until any child process ends
  - `waitpid()` suspends until the specified child process ends
- `wait()` returns the status code of the child
- Unix: every process must be “reaped” by a parent
  - Exited child process waiting to be cleaned up: zombie process
  - What happens if a parent process exits before a child?

Unix Shell

```
while (1) {
   char *cmd = read_command();
   int child_pid = fork();
   if (child_pid == 0) {
          Manipulate STDIN/OUT/ERR file descriptors for pipes, redirection, etc.
          exec(cmd);
          panic("exec failed");
   } else {
          waitpid(child_pid);
   }
}
```

Questions about Process Creation

- What happens if you run `exec bash` in your shell?
- What happens if you run `exec ls` in your shell?

Today's Outline

- Processes
  - How can we represent a running program?
    - » Processes and Process Control Blocks (PCBs)
- Process management
  - How can we schedule different processes on the CPU?
    - » Process states, context switches, and process queues
  - What APIs can we use to interact with processes?
    - » Unix: `fork, exec, exit, wait`
    - » Windows: `CreateProcess, ExitProcess, WaitForSingleObject`

For next class…

- Read chapters 26-27
- Start looking at PR1