CS 153
Design of Operating Systems

Fall 21

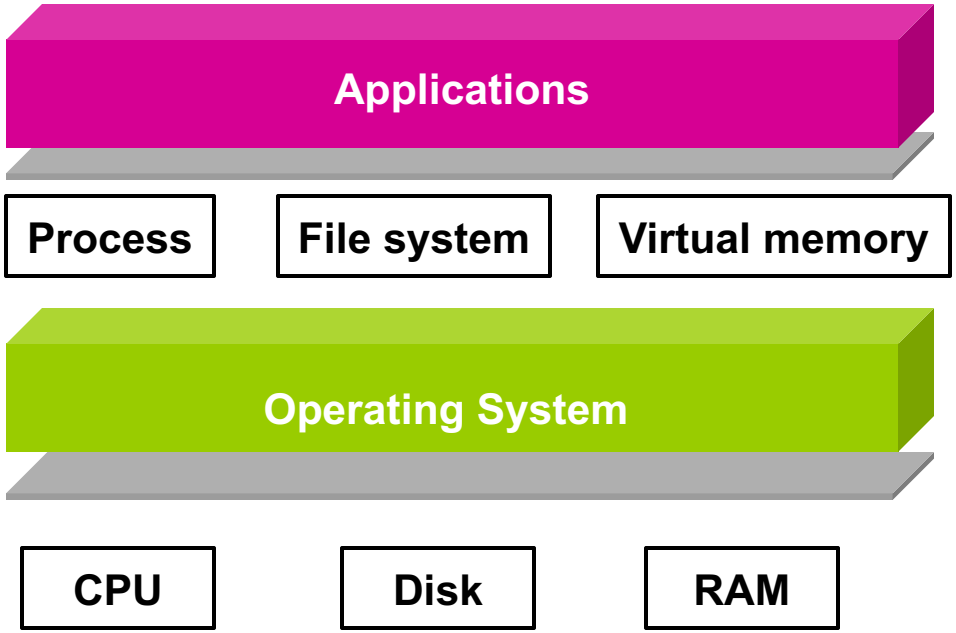
Lecture 4: Process

Instructor: Chengyu Song

Last class

- OS structure, operation, and interaction with user apps
  - **Privileged mode**: To enforce isolation and manage resources, OS must have exclusive powers not available to users
    - » How does the switch happen securely?
  - **OS is not running unless there is an event**:
    - » OS schedules a user process to run then goes to sleep
    - » It wakes up (who wakes it?) to handle events
    - » Many types of events
  - **Program view and system calls**: program asks the OS when it needs a privileged operation

OS Abstractions



Today, we start discussing the first abstraction that enables us to virtualize (i.e., share) the CPU – processes!

What is Virtualization?

- What is a virtual something?
  - Somehow not real? But still functional?
- Provide illusion for each program of own copy of resources
  - Let's say the CPU or memory; every program thinks it has its own
  - In reality, limited physical resources (e.g., 1 CPU)
    - » It must be shared! (in time, or space)
- Frees up programs from worrying about sharing
  - The OS implements sharing, creating illusion of exclusive resources → Virtualization!
- Virtual resource provided as an object with defined operations on it → abstraction

Virtualizing the CPU

- This lecture starts a class segment that covers processes, scheduling, threads, and concurrency
  - Basis for Midterm and Lab 1 & 2

- Today's topics are processes and process management
  - How do we virtualize the CPU?
    - » Give each program the illusion of its own CPU
    - » What is the magic? We only have one real CPU
  - How are applications represented in the OS?
  - How is work scheduled in the CPU?

The Process

- The process is the OS abstraction for execution
  - It is a collection of resources
  - It is a unit for management
- A process is a program in execution
  - Programs are static entities with the potential for execution
  - Process is the animated/active program
    - » Starts from the program, but also includes dynamic state
    - » As the representative of the program, it is the “owner” of other resources (memory, files, sockets, …)

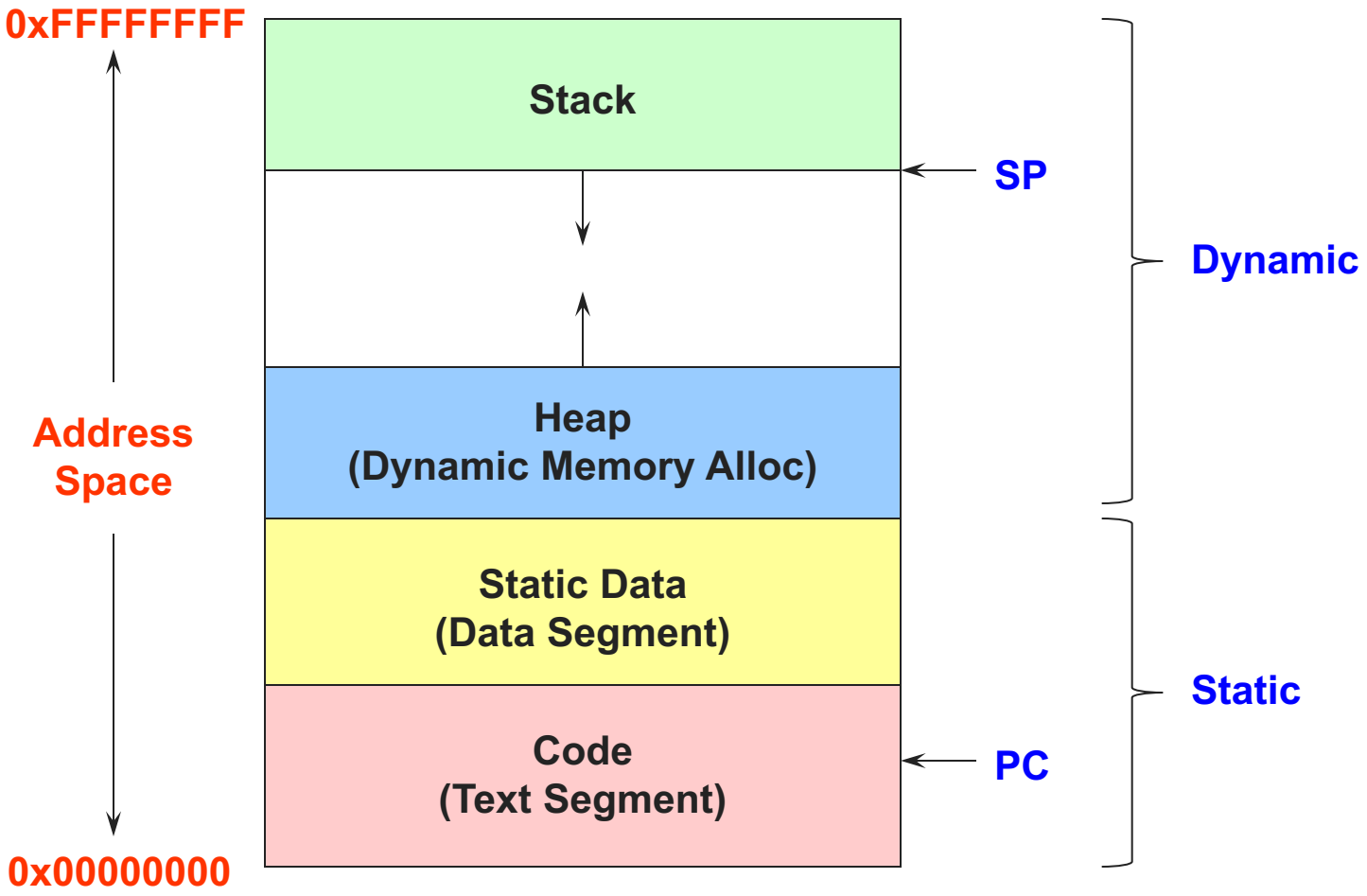
How to support this abstraction?

- First, we'll look at what state a process encapsulates
  - State of the virtual processor we are giving to each program
- Next, we talk about process behavior/CPU time sharing
  - How to implement the process illusion
- Next, we discuss how the OS implements this abstraction
  - What data structures it keeps, and the role of the scheduler
- Finally, we see the process interface offered to programs
  - How to use this abstraction
  - Next class

Process Components

- A process contains all the states for a program in execution
  - An address space containing
    - » Static memory:
      - The code and input data for the executing program
    - » Dynamic memory:
      - The memory allocated by the executing program
      - An execution stack encapsulating the state of procedure calls
  - Control registers such as the program counter (PC)
  - 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)

Address Space



How to support this abstraction?

- First, we'll look at what state a process encapsulates
  - State of the virtual processor we are giving to each program
- Next, we talk about process behavior/CPU time sharing
  - How to implement the process illusion
- Next, we discuss how the OS implements this abstraction
  - What data structures it keeps, and the role of the scheduler
- Finally, we see the process interface offered to programs
  - How to use this abstraction
  - Next class

Process Execution State

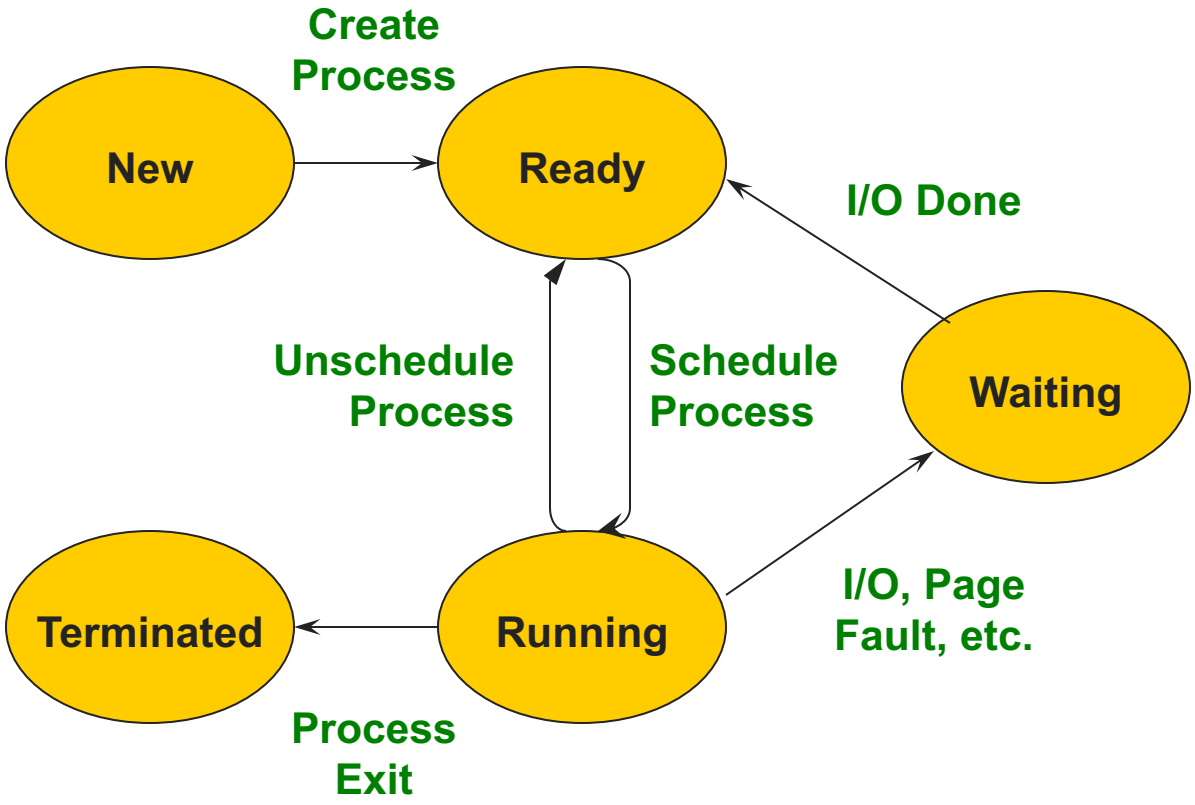
- A process is born, executes for a while, and then dies
- The process execution state that indicates what it is currently doing
  - Running: Executing instructions on the CPU
    - » It is the process that has control of the CPU
    - » How many processes can be in the running state simultaneously?
  - Ready: Waiting to be assigned to the CPU
    - » Ready to execute, but another process is executing on the CPU
  - Waiting: Waiting for an event, e.g., I/O completion
    - » It cannot make progress until event is signaled (disk completes)

Execution State (cont'd)

- As a process executes, it moves from state to state
  - Unix "ps -x": STAT column indicates execution state
  - What state do you think a process is in most of the time?
  - How many processes can a system support?

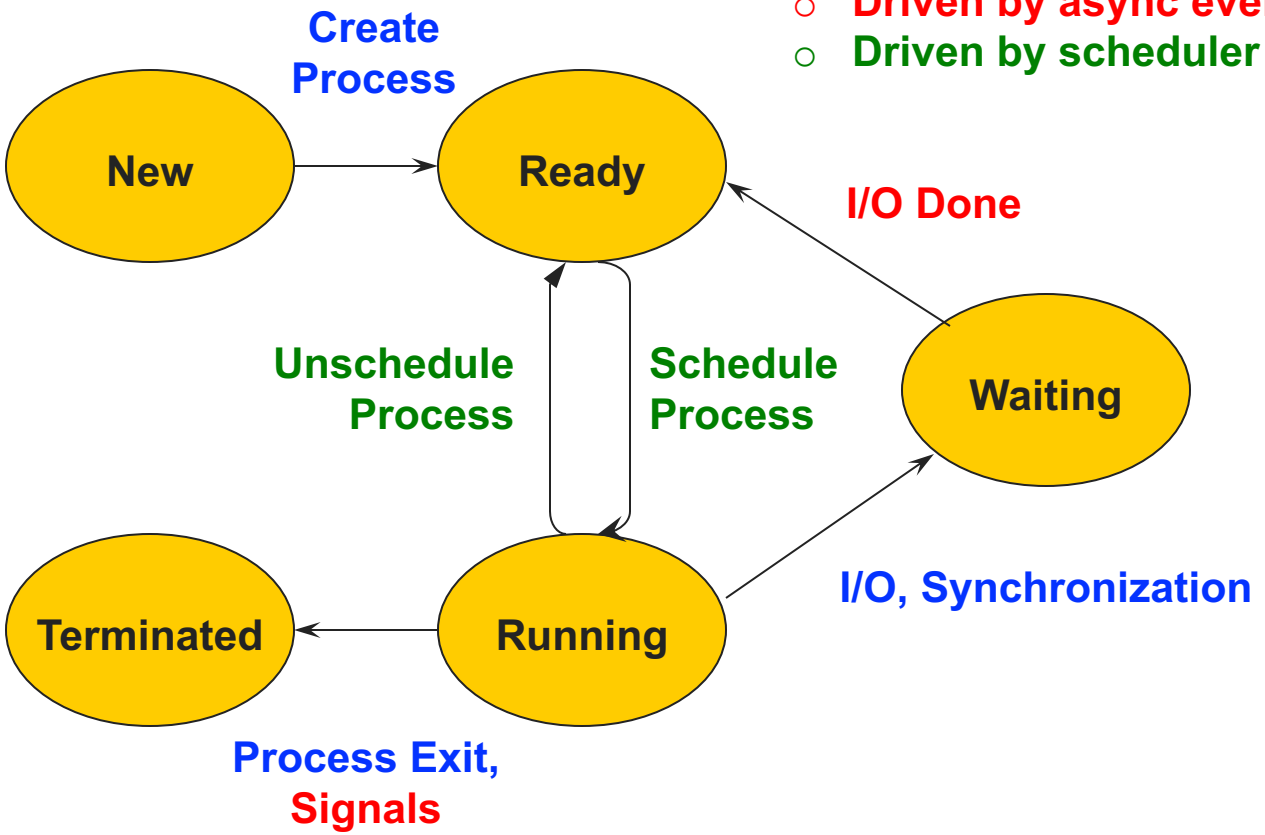
```
PROCESS STATE CODES
       Here are the different values that the s, stat and state output specifiers (header "S
       D    uninterruptible sleep (usually IO)
       R    running or runnable (on run queue)
       S    interruptible sleep (waiting for an event to complete)
       T    stopped, either by a job control signal or because it is being traced.
       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.

       For BSD formats and when the stat keyword is used, additional characters may be displ
       <    high-priority (not nice to other users)
       N    low-priority (nice to other users)
       L    has pages locked into memory (for real-time and custom IO)
       s    is a session leader
       l    is multi-threaded (using CLONE_THREAD, like NPTL pthreads do)
       +    is in the foreground process group.
```

Execution State Graph



Execution State Graph

- Driven by process (system calls)
- Driven by async events
- Driven by scheduler



How to support the process abstraction?

- First, we'll look at what state a process encapsulates
  - State of the virtual processor we are giving to each program
- Next, we will talk about process behavior/CPU time sharing
  - How to implement the process illusion
- Next, we discuss how the OS implements this abstraction
  - What data structures it keeps, and the role of the scheduler
- Finally, we see the process interface offered to programs
  - How to use this abstraction?
  - What system calls are needed?

How does the OS support this model?

We will discuss three issues:

1. How does the OS represent a process in the kernel?
   - The OS data structure representing each process is called the Process Control Block (PCB)
2. How do we pause and restart processes?
   - We must be able to save and restore the full machine state
3. How do we keep track of all the processes in the system?
   - A lot of queues!

PCB Data Structure

- PCB also is where OS keeps all of a process' hardware execution state when the process is not running
  - » Process ID (PID)
  - » Execution state
  - » Hardware state: PC, SP, other registers
  - » Memory management
  - » Scheduling
  - » Accounting
  - » Pointers for state queues
  - » Etc.
- These states are everything that is needed to restore the hardware to the same configuration it was in when the process was switched out of the hardware

Xv6 struct proc

```
enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };

// Per-process state
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Linear address of proc's pgdir
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  volatile int pid;            // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // Switch here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
};
```

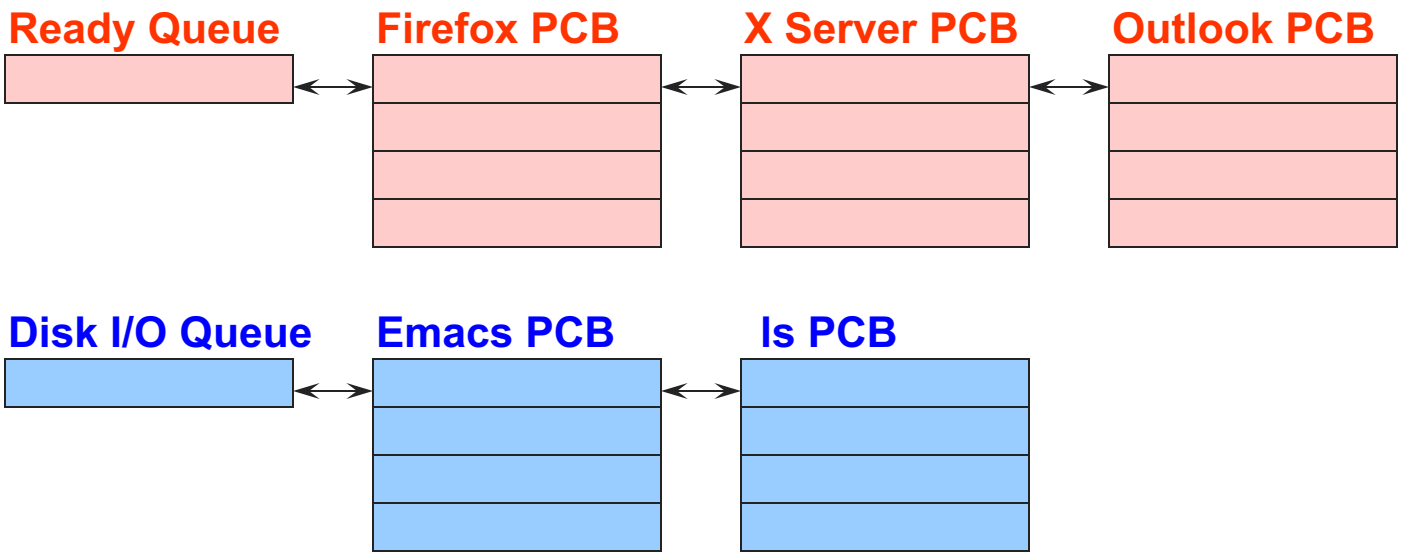
How to pause/restart processes?

- When a process is running, its dynamic state is in memory and some hardware registers
  - Hardware registers include program counter, stack pointer, control registers, data registers, …
  - To be able to stop and restart a process, we need to completely restore this state
- When the OS stops running a process, it saves the current values of the registers (usually in PCB)
- When the OS restarts executing a process, it loads the hardware registers from the stored values in PCB
- Changing CPU hardware state from one process to another is called a context switch
  - This can happen 100s or 1000s of times a second!

How does the OS track processes?

- The OS maintains a collection of queues that represent the state of all processes in the system
- Typically, the OS at least one queue for each state
  - Ready, waiting, etc.
- Each PCB is queued on a state queue according to its current state
- As a process changes state, its PCB is unlinked from one queue and linked into another

State Queues



Console Queue

Sleep Queue

.

.

.

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

How to support the process abstraction?

- First, we'll look at what state a process encapsulates
  - State of the virtual processor we are giving to each program
- Next, we will talk about process behavior/CPU time sharing
  - How to implement the process illusion
- Next, we discuss how the OS implements this abstraction
  - What data structures it keeps, and the role of the scheduler
- Finally, we see the process interface offered to programs
  - How to use this abstraction?
  - What system calls are needed?

Process System Call API

- Process creation: how to create a new process?
- Process termination: how to terminate and clean up a process
- Coordination between processes
  - wait, waitpid, signal, inter-process communication, synchronization
- Other
  - E.g., set quotas or priorities, examine usage, …

Process Creation

- A process is created by another process
  - Why is this the case?
  - Parent is creator, child is created (Unix: ps "PPID" field)
  - Who creates the first process (Unix: init (PID 0 or 1))?
- In some systems, the parent defines (or donates) resources and privileges for its children
  - Unix: Process User ID is inherited – children of your shell execute with your privileges
- After creating a child, the parent may either wait for it to finish its task or continue in parallel (or both)

Process Creation: Windows

- The system call on Windows for creating a process is called, surprisingly enough, CreateProcess:

  `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" into the address space
  - Copies "args" into memory allocated in address space
  - Initializes the saved hardware context to start execution at main (or wherever specified in the file)
  - Places the PCB on the ready queue

Process Creation: Unix

- In Unix, processes are created using fork()

```
int fork()
```

- fork()
  - Creates and initializes a new PCB
  - Creates 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 parent (e.g., open files)
  - Places the PCB on the ready queue
- Fork returns twice
  - Returns the child's PID to the parent, "0" to the child

fork()

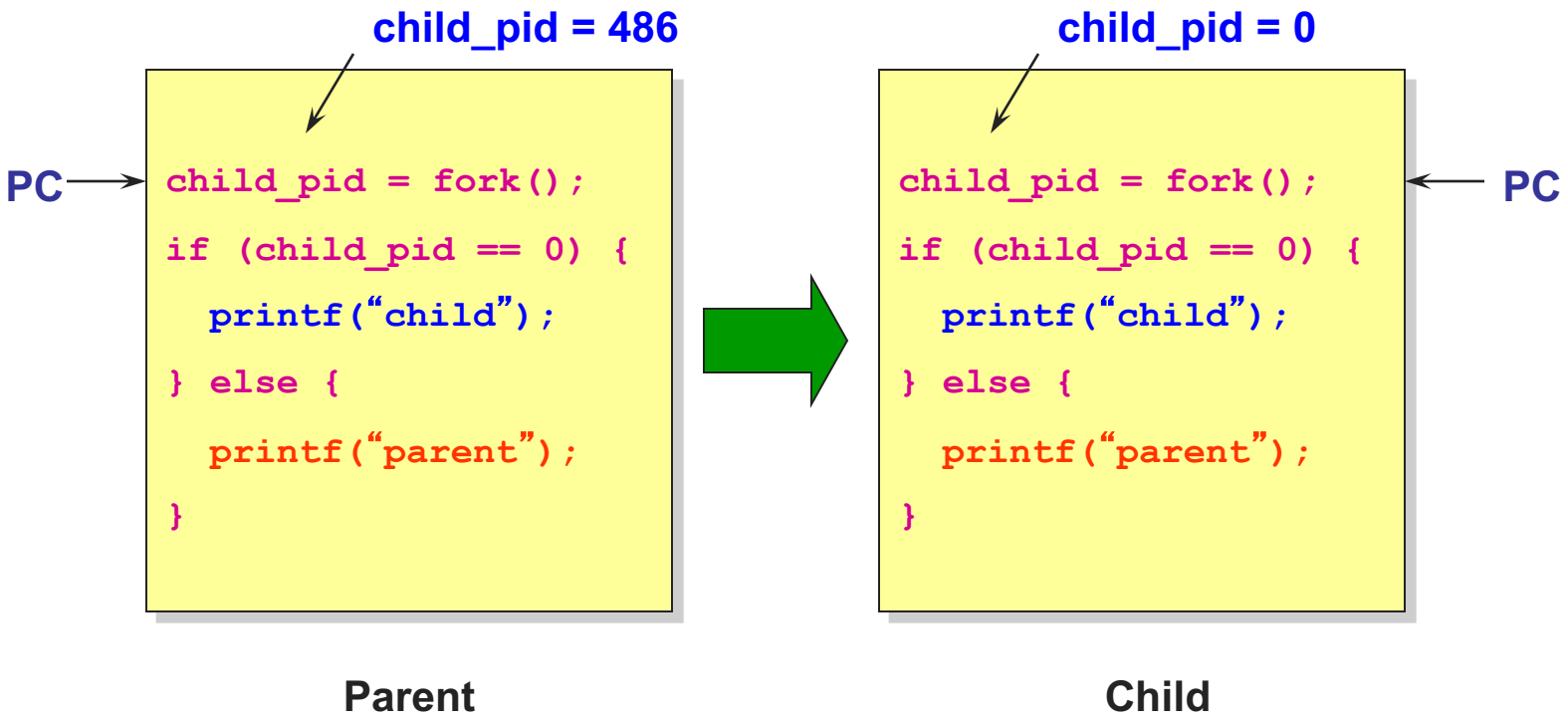
```
int main(int argc, char *argv[])
{
   char *name = argv[0];
   int child_pid = fork();
   if (child_pid == 0) {
        printf("Child of %s is %d\n", name, getpid());
        return 0;
   } else {
        printf("My child is %d\n", child_pid);
        return 0;
   }
}
```

What does this program print?

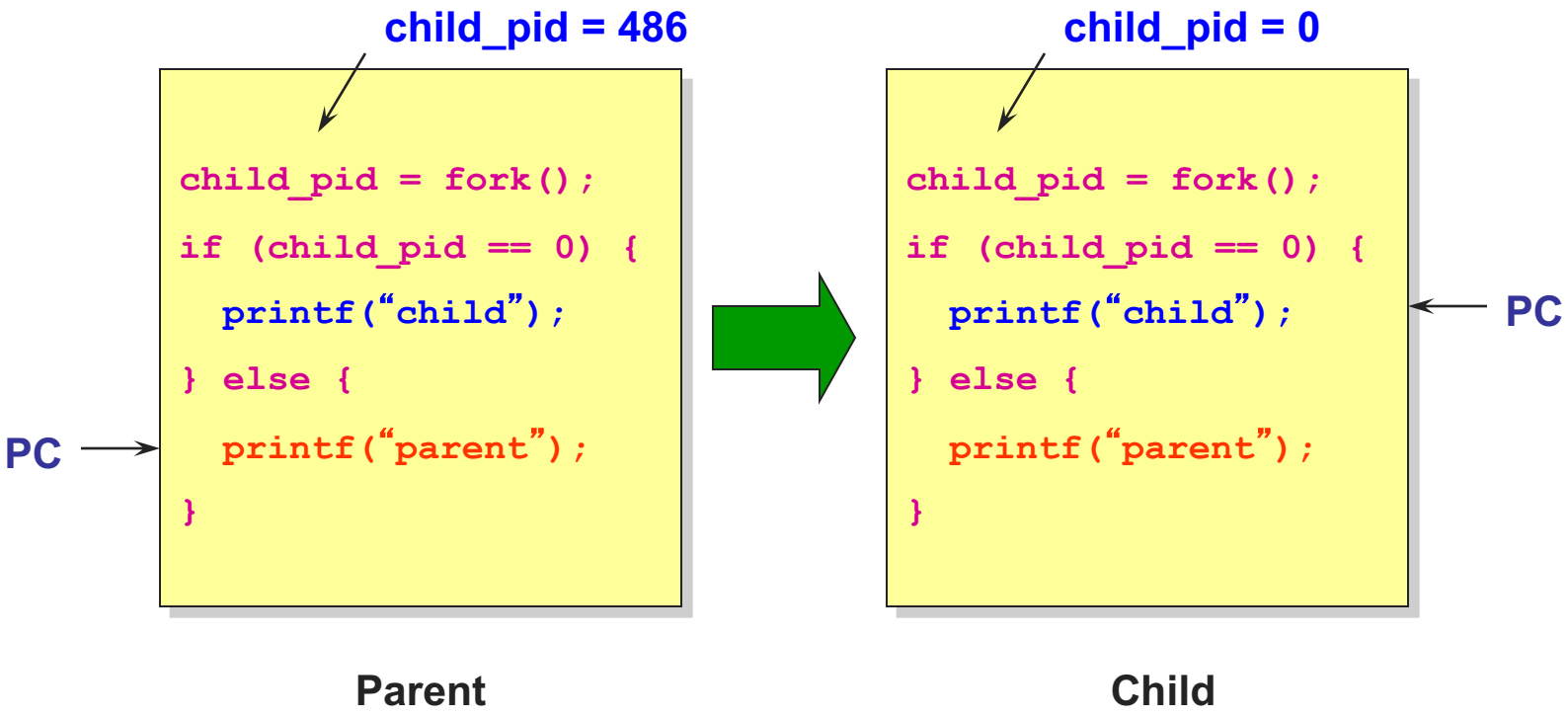
Example Output

```
[well ~]$ gcc t.c
[well ~]$ ./a.out
My child is 486
Child of a.out is 486
```

Duplicating Address Spaces



Divergence


child_pid = 486
child_pid = fork();
if (child_pid == 0) {
printf("child");
} else {
printf("parent");
}
PC
Parent
child_pid = 0
child_pid = fork();
if (child_pid == 0) {
printf("child");
} else {
printf("parent");
}
PC
Child

Example Continued

```
[well ~]$ gcc t.c
[well ~]$ ./a.out
My child is 486
Child of a.out is 486
[well ~]$ ./a.out
Child of a.out is 498
My child is 498
```

Why is the output in a different order?

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();
  if ((child_pid = fork()) == 0) {
    Handle client request
  } else {
    Close socket
  }
}
```

Process Creation: Unix (2)

- Wait a second. How do we actually start a new 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
- What does it mean for exec to return?
- What does it mean for exec to return with an error?

Process Creation: Unix (3)

- fork() is used to create a new process, exec is used to load a program into the address space

- What happens if you run "exec csh" in your shell?
- What happens if you run "exec ls" in your shell? Try it.

- fork() can return an error. Why might this happen?

Process Termination

- All good processes must come to an end. But how?
  - Unix: exit(int status), NT: ExitProcess(int status)
- Essentially, free resources and terminate
  - Terminate all threads (next lecture)
  - Close open files, network connections
  - Allocated memory (and VM pages out on disk)
  - Remove PCB from kernel data structures, delete
- Note that a process does not need to clean up itself
  - OS will handle this on its behalf

wait() a second…

- Often it is convenient to pause until a child process has finished
  - Think of executing commands in a shell
- Use wait() (WaitForSingleObject)
  - Suspends the current process until a child process ends
  - waitpid() suspends until the specified child process ends
- Wait has a return value…what is it?
- Unix: Every process must be reaped by a parent
  - What happens if a parent process exits before a child?
  - What do you think is a "zombie" process?

Unix Shells

```
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 {
        if (!(run_in_background))
              waitpid(child_pid);
   }
}
```

Process: check your understanding

- What are the units of execution?
  - Processes
- How are those units of execution represented?
  - Process Control Blocks (PCBs)
- How is work scheduled in the CPU?
  - Process states, process queues, context switches
- What are the possible execution states of a process?
  - Running, ready, waiting, …
- How does a process move from one state to another?
  - Scheduling, I/O, creation, termination
- How are processes created?
  - CreateProcess (NT), fork/exec (Unix)

Next Time…

- Scheduling
- Preparation
  - Module 7 & 8 of the textbook