

# **CSE 120**

## **Principles of Operating Systems**

**Fall 2001**

**Lecture 3: Processes**

Geoffrey M. Voelker

## **Processes**

---

- This lecture starts a class segment that covers processes, threads, and synchronization
  - ♦ These topics are perhaps the most important in this class.
  - ♦ You can rest assured that they will be covered in the exams.
- Today's topics are processes and process management
  - ♦ What are the units of execution?
  - ♦ How are those units of execution represented in the OS?
  - ♦ How is work scheduled in the CPU?
  - ♦ What are the possible execution states of a process?
  - ♦ How does a process move from one state to another?

# The Process

---

- The process is the OS **abstraction for execution**
  - It is the unit of execution
  - It is the unit of scheduling
  - It is the dynamic execution context of a program
- A process is sometimes called a **job** or a **task** or a **sequential process**
- A sequential process is a **program in execution**
  - It defines the sequential, instruction-at-a-time execution of a program
  - Programs are static entities with the potential for execution

September 26, 2001

CSE 120 – Lecture 3 – Processes

3

# Process Components

---

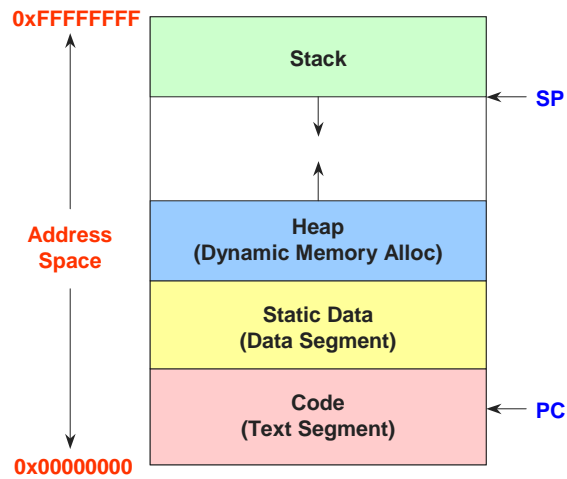
- A process contains all of the state for a program in execution
  - An address space
  - The code for the executing program
  - The data for the executing program
  - An execution stack encapsulating the state of procedure calls
  - The program counter (PC) indicating the next instruction
  - A set 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)

September 26, 2001

CSE 120 – Lecture 3 – Processes

4

# Process Diagram



September 26, 2001

CSE 120 – Lecture 3 – Processes

5

# Process State

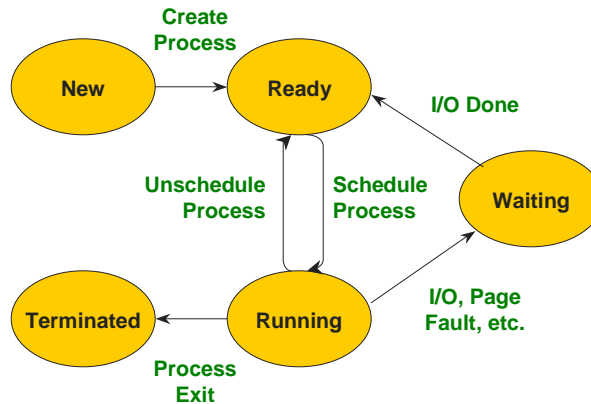
- A process has an **execution state** that indicates what it is currently doing
  - ♦ **Running**: Executing instructions on the CPU
    - » It is the process that has control of the CPU
    - » **How many processes can be in the running state simultaneously?**
  - ♦ **Ready**: Waiting to be assigned 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)
- As a process executes, it moves from state to state
  - ♦ Unix "ps": **STAT** column indicates execution state
  - ♦ **What state do you think a process is in most of the time?**

September 26, 2001

CSE 120 – Lecture 3 – Processes

6

## Process State Graph



September 26, 2001

CSE 120 – Lecture 3 – Processes

7

## Process Data Structures

How does the OS represent a process in the kernel?

- At any time, there are many processes in the system, each in its particular state
- The OS data structure representing each process is called the **Process Control Block (PCB)**
- The PCB contains all of the info about a process
- The PCB also is where the OS keeps all of a process' hardware execution state (PC, SP, regs, etc.) when the process is not running
  - This state is everything that is needed to restore the hardware to the same configuration it was in when the process was switched out of the hardware

September 26, 2001

CSE 120 – Lecture 3 – Processes

8

## PCB Data Structure

---

- The PCB contains a huge amount of information in one large structure
  - » Process ID (PID)
  - » Execution state
  - » Hardware state: PC, SP, regs
  - » Memory management
  - » Scheduling
  - » Accounting
  - » Pointers for state queues
  - » Etc.
- Unix: PCB is defined in sys/proc.h as `struct proc`
  - ♦ FreeBSD: 81 fields, 408 bytes (*not* lightweight)

September 26, 2001

CSE 120 – Lecture 3 – Processes

9

## PCBs and Hardware State

---

- When a process is running, its hardware state (PC, SP, regs, etc.) is in the CPU
  - ♦ The hardware registers contain the current values
- When the OS stops running a process, it saves the current values of the registers into the process' PCB
- When the OS is ready to start executing a new process, it loads the hardware registers from the values stored in that process' PCB
  - ♦ What happens to the code that is executing?
- The process of changing the CPU hardware state from one process to another is called a context switch
  - ♦ This can happen 100 or 1000 times a second!

September 26, 2001

CSE 120 – Lecture 3 – Processes

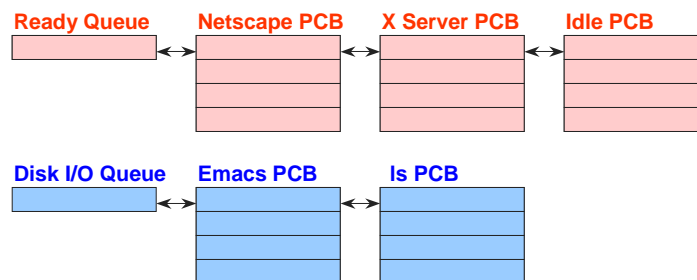
10

# State Queues

How does the OS keep track of processes?

- The OS maintains a collection of queues that represent the state of all processes in the system
- Typically, the OS has 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.)

## PCBs and State Queues

---

- PCBs are data structures dynamically allocated in OS memory
- When a process is created, the OS allocates a PCB for it, initializes, and placed on the ready queue
- As the process computes, does I/O, etc., its PCB moves from one queue to another
- When the process terminates, its PCB is deallocated

## Process Creation

---

- A process is created by another process
  - Parent is creator, child is created (Unix: ps “PPID” field)
  - What 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: NT

---

- The system call on NT 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 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
  - Huh?

## 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;
    }
}
```

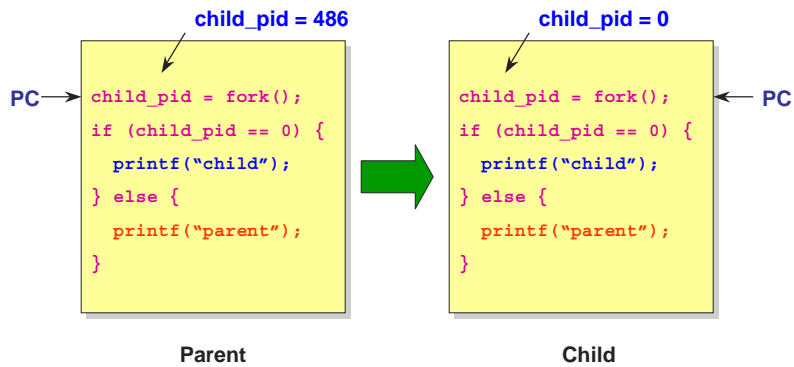
What does this program print?

## Example Output

---

```
alpenglow (18) ~/tmp> cc t.c
alpenglow (19) ~/tmp> a.out
My child is 486
Child of a.out is 486
```

# Duplicating Address Spaces

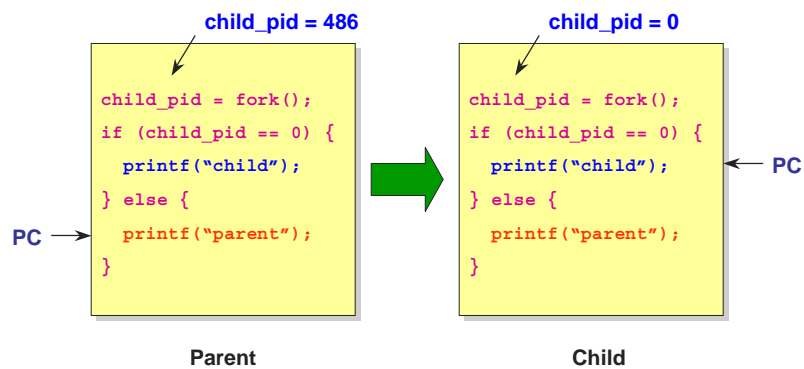


September 26, 2001

CSE 120 – Lecture 3 – Processes

19

# Divergence



September 26, 2001

CSE 120 – Lecture 3 – Processes

20

## Example Continued

---

```
alpenglow (18) ~/tmp> cc t.c
alpenglow (19) ~/tmp> a.out
My child is 486
Child of a.out is 486
alpenglow (20) ~/tmp> 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?

## 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 {
        wait(child_pid);
    }
}
```

## Process Creation: Unix (3)

---

- Fork is used to create a new process, exec is used to load a program into the address space
  - ♦ Why does NT have CreateProcess while Unix uses fork and exec?
- What happens if you run “exec csh” in your shell?
- What happens if you run “exec ls” in your shell? Try it.

## Process Summary

---

- 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...

---

- Read Section 2.2
- Homework #1 due