

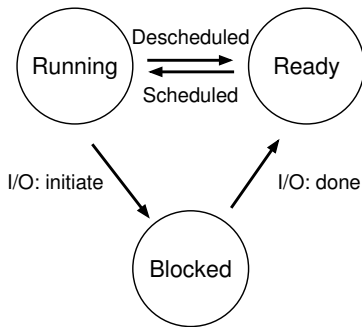
# CSL 301

## OPERATING SYSTEMS

### Lecture 2

#### The Process

Instructor  
Dr. Dhiman Saha



Process State Diagram

Linux commands:

- ▶ `ps ax | wc`
- ▶ `top`
- ▶ `cat /proc/cpuinfo | grep 'model name'`

What is your observation?

- ▶ Recall the issue!

## How to provide the illusion of many CPUs?

- ▶ Although there are only a **few** physical CPUs available, how can the OS provide the illusion of a **nearly-endless supply** of said CPUs?
- ▶ First steps towards virtualizing the CPU
- ▶ The Abstraction: The Notion of Process

By running one process, then stopping it and running another, and so forth, the OS can promote the illusion that many virtual CPUs exist when in fact there is only one physical CPU (or a few).

- ▶ Each process will get its own virtual CPU
  - ▶ Unaware of the sharing behind-the-scenes
- 
- ▶ Other analogues:
    - ▶ Memory: Space Sharing
    - ▶ Disk: Space Sharing

Implementing virtualization of the CPU needs:

- ▶ Low-level machinery: Mechanisms
- ▶ High-level intelligence: Policies

## Mechanisms

Low-level methods or protocols that implement a needed piece of functionality

- ▶ e.g. context-switch mechanism<sup>1</sup>

## Policies

Algorithms for making some kind of decision within the OS.

- ▶ e.g. which process to run next (scheduling policy)

---

<sup>1</sup>Will be detailed in a later class

- ▶ Informally: Process  $\implies$  A running program
- ▶ Basically it is like an instance of a program

Note there may be **many** processes per program

- ▶ Recall the Java analogy
  - ▶ class  $\implies$  “program”
  - ▶ object  $\implies$  “process”

# What constitutes a process?

- ▶ What parts of the machine are important to the execution of a program at a given time?

## The machine state

At any instant in time, we can summarize a process by taking an **inventory** of the different pieces of the system it **accesses** or **affects** during the course of its execution.

# What constitutes a process?

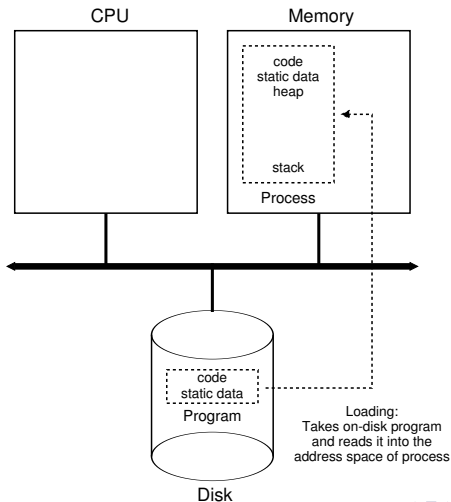
- ▶ The process identifier (PID)
- ▶ Memory image
  - ▶ Code and data (static)
  - ▶ Stack and heap (dynamic)
- ▶ CPU context: registers
  - ▶ Program counter
  - ▶ Current operands
  - ▶ Stack pointer
- ▶ File descriptors
  - ▶ Pointers to open files and devices



These APIs, in some form, are available on any modern operating system.

- ▶ Create
- ▶ Destroy
- ▶ Wait
- ▶ Miscellaneous Control (e.g. Suspend)
- ▶ Status

How programs are transformed into processes?



# How does OS create a process?

- ▶ Allocates memory and creates memory image
  - ▶ Loads code, data from disk exe
  - ▶ Creates runtime stack, heap

# How does OS create a process?

- ▶ Allocates memory and creates memory image
  - ▶ Loads code, data from disk exe
  - ▶ Creates runtime stack, heap
- ▶ Opens basic files
  - ▶ e.g. stdin, stdout, stderr
  - ▶ Recall the basic I/O you do when running a command on the terminal

# How does OS create a process?

- ▶ Allocates memory and creates memory image
  - ▶ Loads code, data from disk exe
  - ▶ Creates runtime stack, heap
- ▶ Opens basic files
  - ▶ e.g. stdin, stdout, stderr
  - ▶ Recall the basic I/O you do when running a command on the terminal
- ▶ Initializes CPU registers
  - ▶ PC points to first instruction

# How does OS create a process?

- ▶ Allocates memory and creates memory image
  - ▶ Loads code, data from disk exe
  - ▶ Creates runtime stack, heap
- ▶ Opens basic files
  - ▶ e.g. stdin, stdout, stderr
  - ▶ Recall the basic I/O you do when running a command on the terminal
- ▶ Initializes CPU registers
  - ▶ PC points to first instruction

## Point to Ponder

Should the entire code/data be loaded at once before running the program?

- ▶ New: being created, yet to run

- ▶ New: being created, yet to run
- ▶ Running: currently executing on CPU

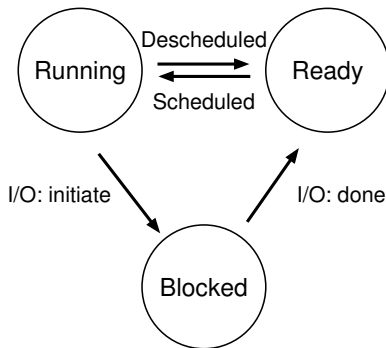


- ▶ New: being created, yet to run
- ▶ Running: currently executing on CPU
- ▶ Ready: waiting to be scheduled

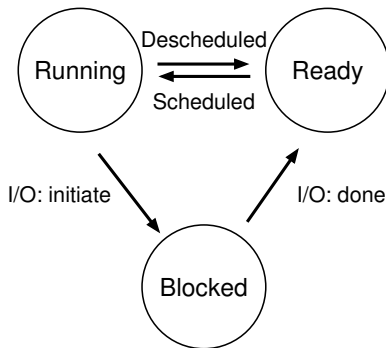
- ▶ New: being created, yet to run
- ▶ Running: currently executing on CPU
- ▶ Ready: waiting to be scheduled
- ▶ Blocked: suspended, not ready to run

- ▶ New: being created, yet to run
- ▶ Running: currently executing on CPU
- ▶ Ready: waiting to be scheduled
- ▶ Blocked: suspended, not ready to run
  - ▶ Why blocked?
  - ▶ When is it unblocked?

- ▶ New: being created, yet to run
- ▶ Running: currently executing on CPU
- ▶ Ready: waiting to be scheduled
- ▶ Blocked: suspended, not ready to run
  - ▶ Why blocked?
  - ▶ When is it unblocked?
- ▶ Dead: terminated

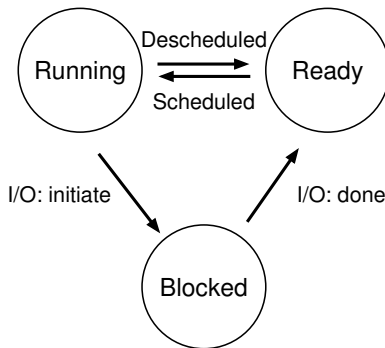


Process State Diagram



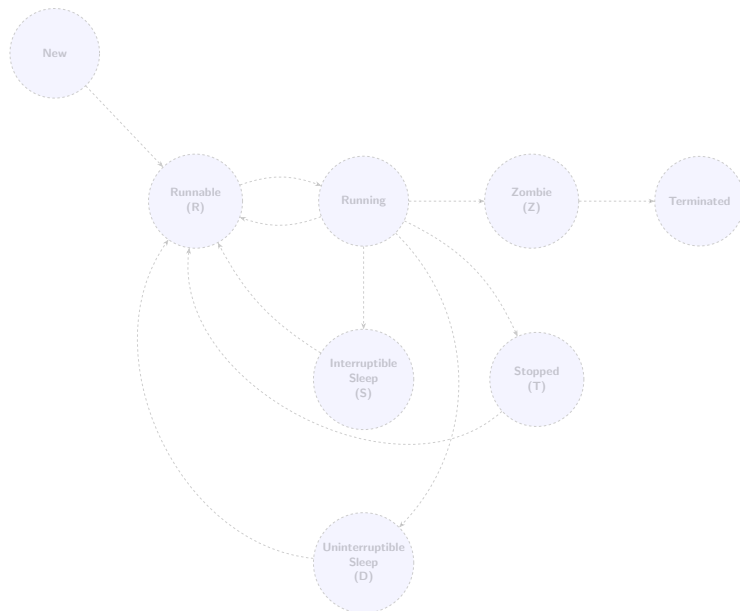
Process State Diagram

- How to transition?  $\Rightarrow$  “mechanism”

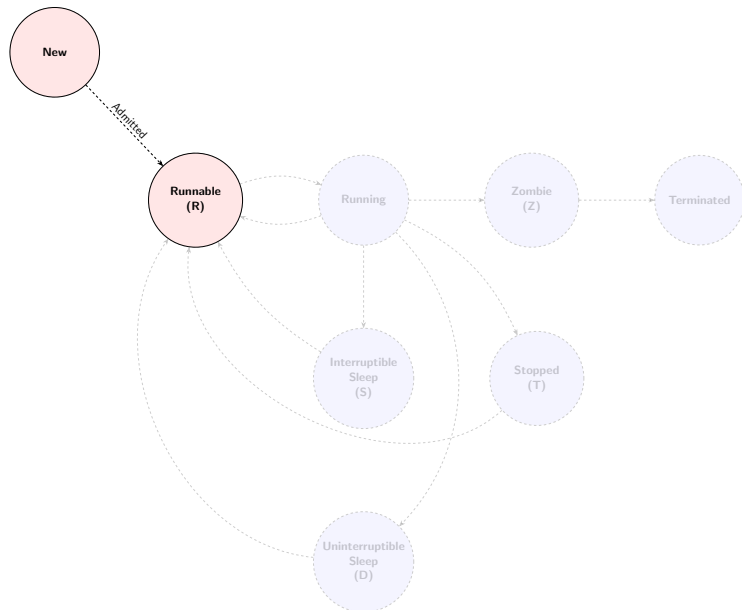


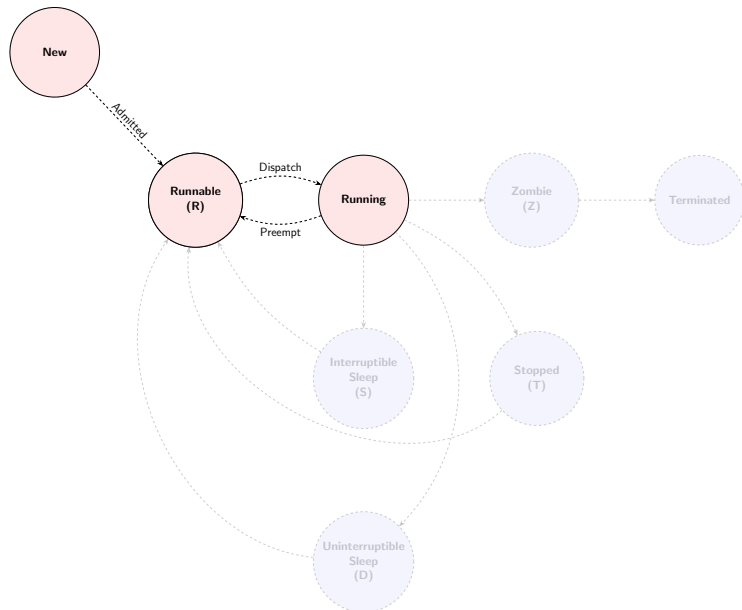
Process State Diagram

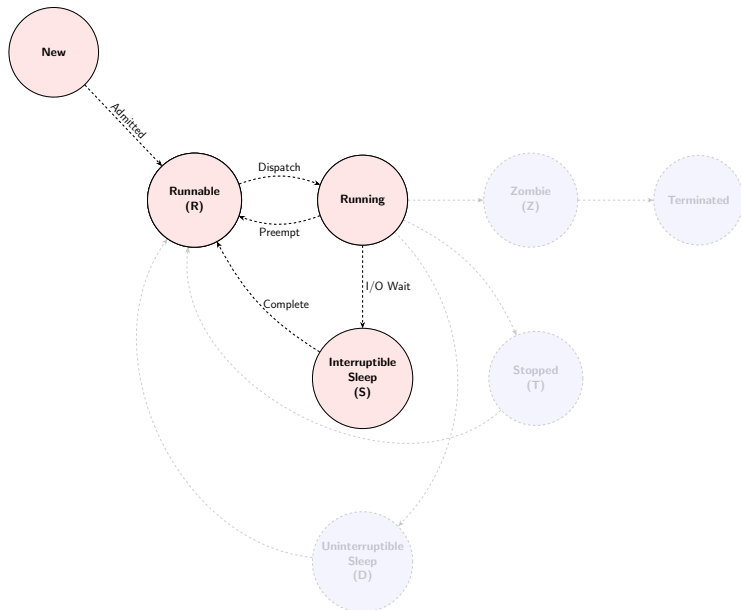
- ▶ How to transition?  $\Rightarrow$  “mechanism”
- ▶ When to transition?  $\Rightarrow$  “policy”

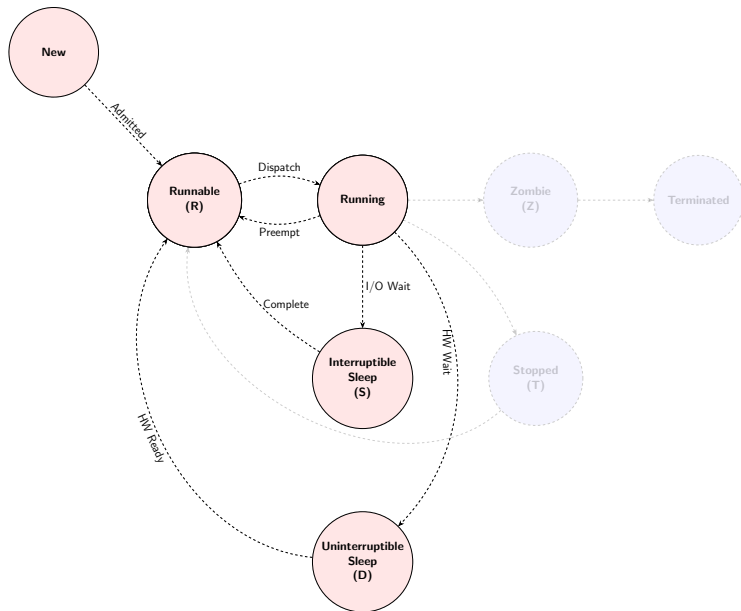


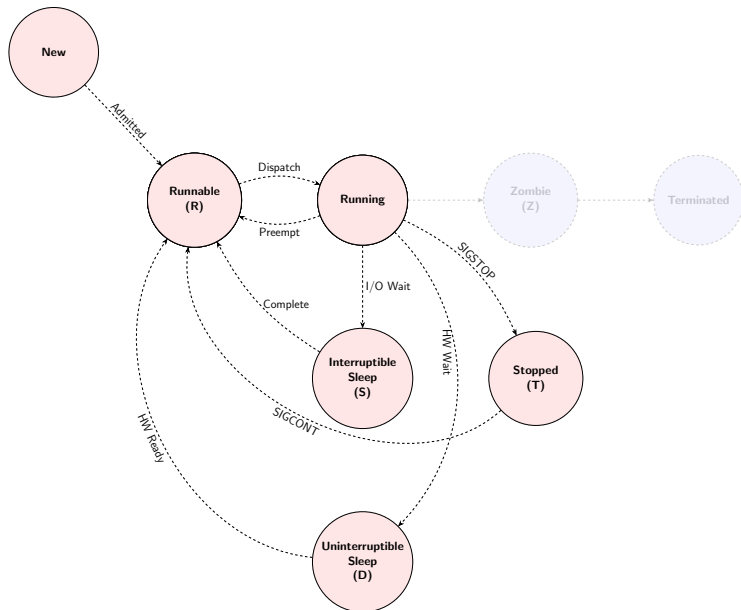


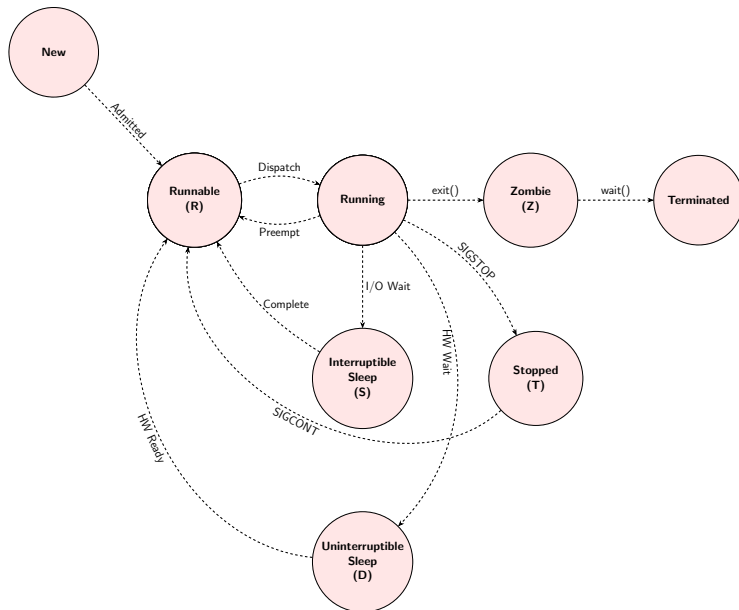




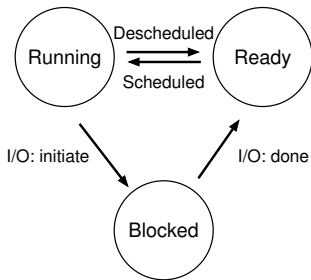






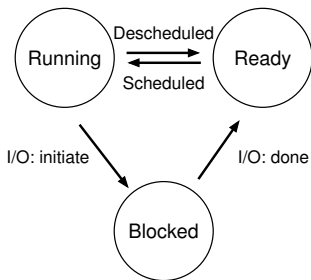


# Tracing Process State: CPU Only



Time	Process <sub>0</sub>	Process <sub>1</sub>	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	
4	Running	Ready	Process <sub>0</sub> now done
5	—	Running	
6	—	Running	
7	—	Running	
8	—	Running	Process <sub>1</sub> now done

# Tracing Process State: CPU and I/O



Time	Process <sub>0</sub>	Process <sub>1</sub>	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	Process <sub>0</sub> initiates I/O
4	Blocked	Running	Process <sub>0</sub> is blocked, so Process <sub>1</sub> runs
5	Blocked	Running	
6	Blocked	Running	
7	Ready	Running	I/O done
8	Ready	Running	Process <sub>1</sub> now done
9	Running	–	
10	Running	–	Process <sub>0</sub> now done



- ▶ The Process List/ Task List
- ▶ To keep track of all the running programs in the system

- ▶ The Process List/ Task List
- ▶ To keep track of all the running programs in the system

How is per process information stored?

The Process Control Block (PCB)

- ▶ Also sometimes called a **process descriptor**
- ▶ Just a (C) **structure** that contains information about a specific process

## Typical contents of a PCB

- ▶ Process identifier
- ▶ Process state
- ▶ Pointers to other related processes
  - ▶ e.g the parent process
  - ▶ Recall the `fork()` function
- ▶ CPU context of the process
  - ▶ Generally saved when the process is suspended
- ▶ Pointers to memory locations
- ▶ Pointers to open files

```
// the registers xv6 will save and restore
// to stop and subsequently restart a process
struct context {
    int eip;
    int esp;
    int ebx;
    int ecx;
    int edx;
    int esi;
    int edi;
    int ebp;
};

// the different states a process can be in
enum proc_state { UNUSED, EMBRYO, SLEEPING,
                  RUNNABLE, RUNNING, ZOMBIE };
```

```
// the information xv6 tracks about each process
// including its register context and state
struct proc {
    char *mem;                // Start of process memory
    uint sz;                  // Size of process memory
    char *kstack;             // Bottom of kernel stack
                                // for this process
    enum proc_state state;    // Process state
    int pid;                  // Process ID
    struct proc *parent;      // Parent 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
    struct context context;    // Switch here to run process
    struct trapframe *tf;      // Trap frame for the
                                // current interrupt
};
```

# Unix Process related system calls

- ▶ `fork()` creates a new child process
  - ▶ All processes are created by forking from a parent
  - ▶ The `init` process is ancestor of all processes
- ▶ `exec()` makes a process execute a given executable
- ▶ `exit()` terminates a process
- ▶ `wait()` causes a parent to block until child terminates
- ▶ Many variants exist of the above system calls with different arguments

Do it yourself!

Chapter 5: Process API (Will be discussed in next class)

## ASIDE: KEY PROCESS TERMS

- The **process** is the major OS abstraction of a running program. At any point in time, the process can be described by its state: the contents of memory in its **address space**, the contents of CPU registers (including the **program counter** and **stack pointer**, among others), and information about I/O (such as open files which can be read or written).
- The **process API** consists of calls programs can make related to processes. Typically, this includes creation, destruction, and other useful calls.
- Processes exist in one of many different **process states**, including running, ready to run, and blocked. Different events (e.g., getting scheduled or descheduled, or waiting for an I/O to complete) transition a process from one of these states to the other.
- A **process list** contains information about all processes in the system. Each entry is found in what is sometimes called a **process control block (PCB)**, which is really just a structure that contains information about a specific process.