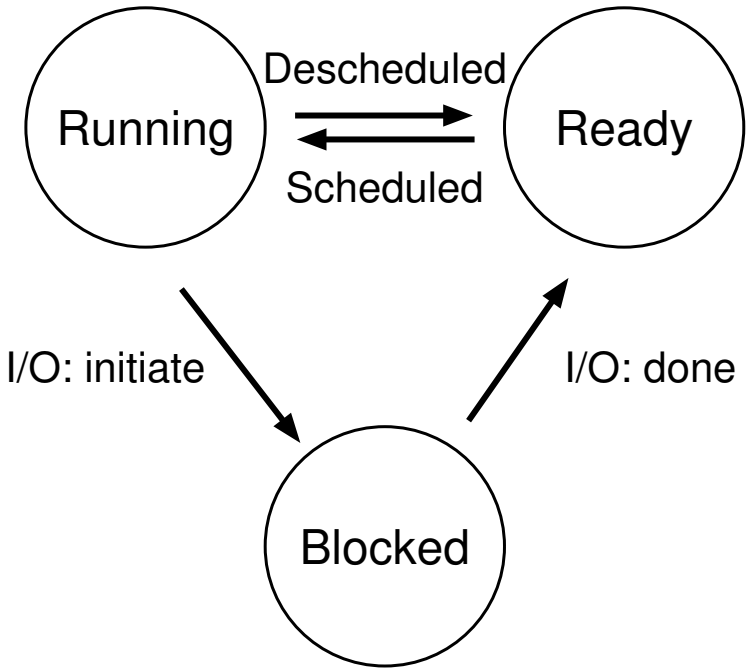


Process State Diagram

CS 250
OPERATING SYSTEMS

Lecture 2
The Process

Instructor
Dr. Dhiman Saha

Linux commands:

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

What is your observation?

‘‘`man ps`’’ to find out more about `ps` command

- 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

Mechanism Vs Policy

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[1]

Policies

Algorithms for making some kind of decision within the OS.

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

[1] Will be detailed in a later class

The Process

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

Note there may **many** processes per program

- Recall the Java analogy
  - class $\Longrightarrow$ "program"
  - object $\Longrightarrow$ "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

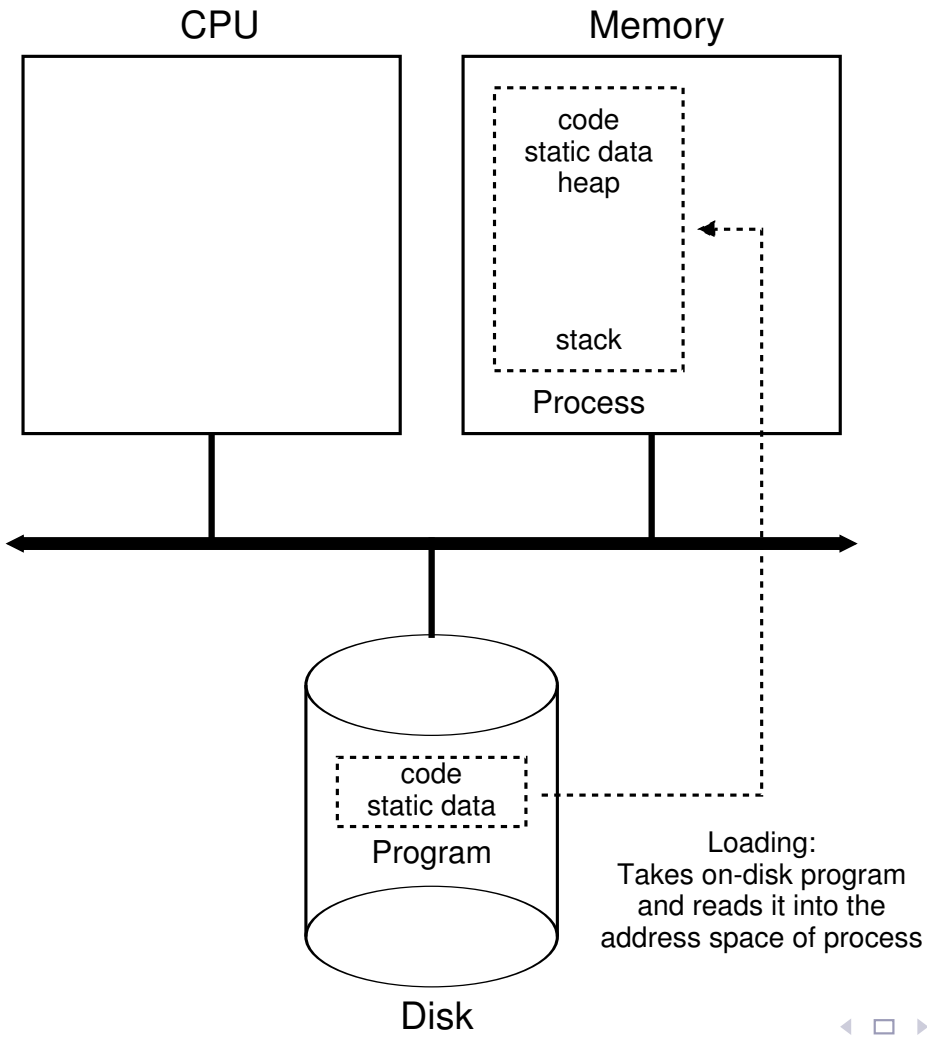
Process API

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

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

Process Creation

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
- 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?

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?

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?

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?

Process States

- 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 States

- 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 States

- 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 States

- 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

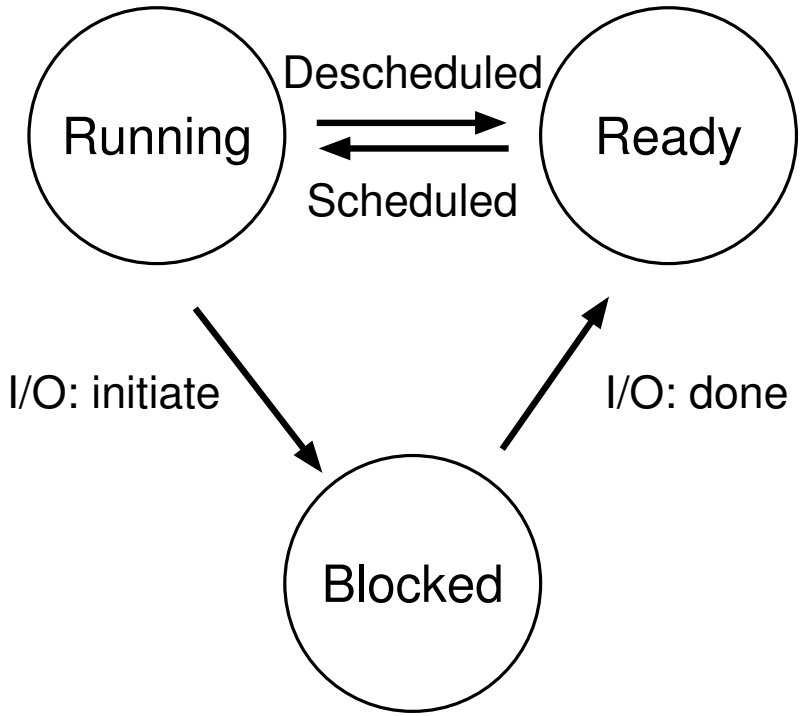
Process States

- 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 States

- 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

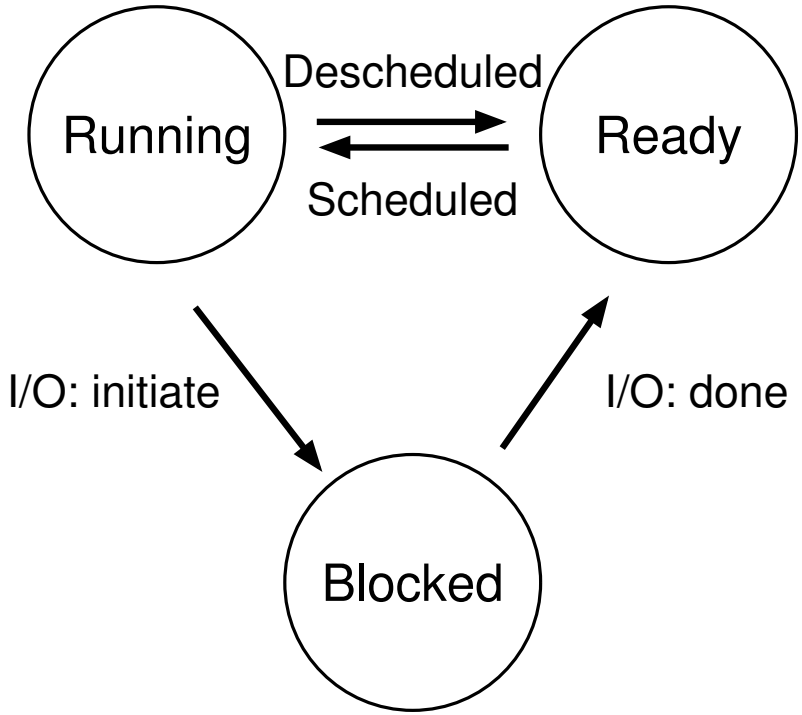
View process state with "ps xa"



Process State Diagram

- How to transition? $\Longrightarrow$ "mechanism"
- When to transition? $\Longrightarrow$ "policy"

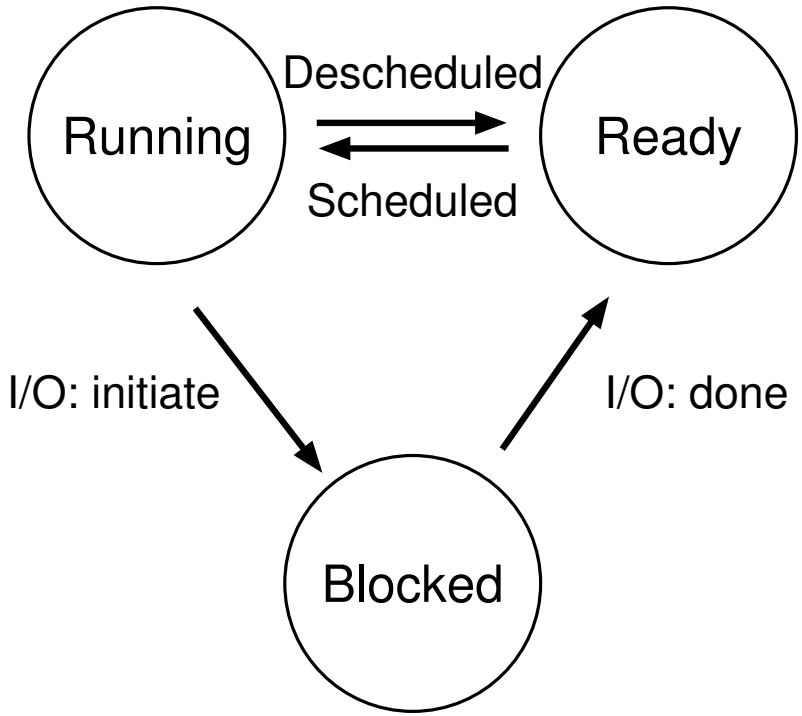
View process state with "ps xa"



Process State Diagram

- How to transition? $\Longrightarrow$ "mechanism"
- When to transition? $\Longrightarrow$ "policy"

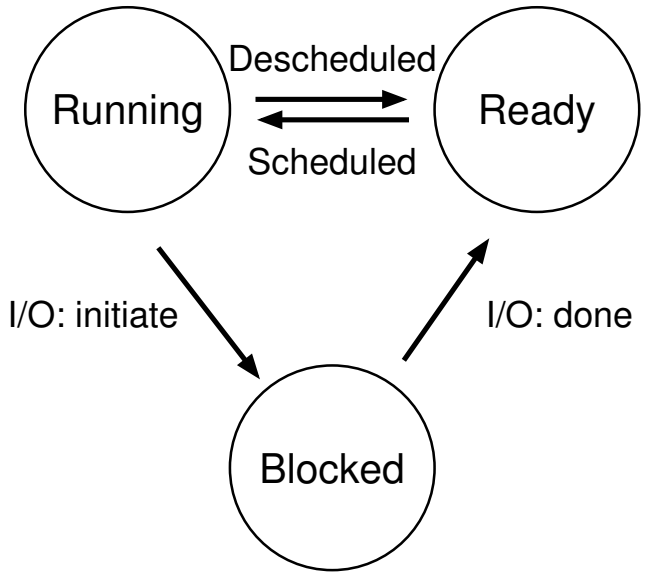
View process state with "ps xa"



Process State Diagram

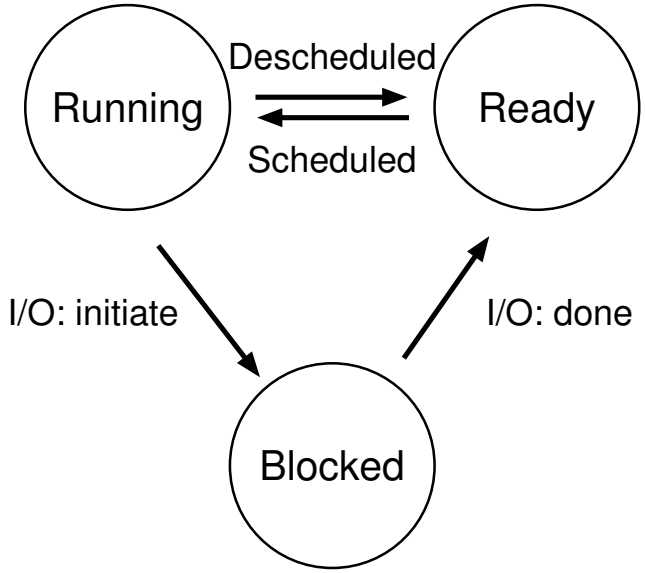
- How to transition? $\Longrightarrow$ "mechanism"
- When to transition? $\Longrightarrow$ "policy"

Tracing Process State: CPU Only



Time	Process$_0$	Process$_1$	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	
4	Running	Ready	Process$_0$ now done
5	–	Running	
6	–	Running	
7	–	Running	
8	–	Running	Process$_1$ now done

Tracing Process State: CPU and I/O



Time	Process_0	Process_1	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	Process_0 initiates I/O
4	Blocked	Running	Process_0 is blocked,
5	Blocked	Running	so Process_1 runs
6	Blocked	Running	
7	Ready	Running	I/O done
8	Ready	Running	Process_1 now done
9	Running	–	
10	Running	–	Process_0 now done

- 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

- 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

Example — The xv6 Proc Structure

```
// 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 };
```

Example — The xv6 Proc Structure

```
// 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 briefly discussed in next class)