

Operating Systems!

Process Implementation (part 1)

From "how to use processes" to "how an OS implements them!"

Prof. Travis Peters

Montana State University

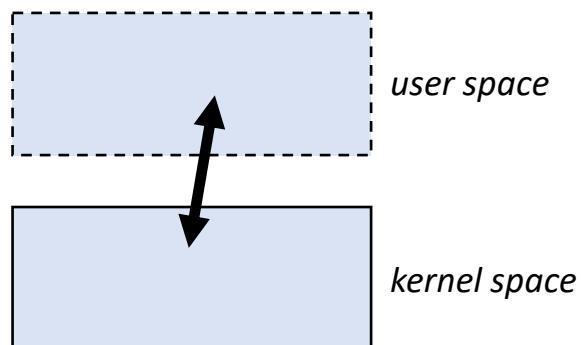
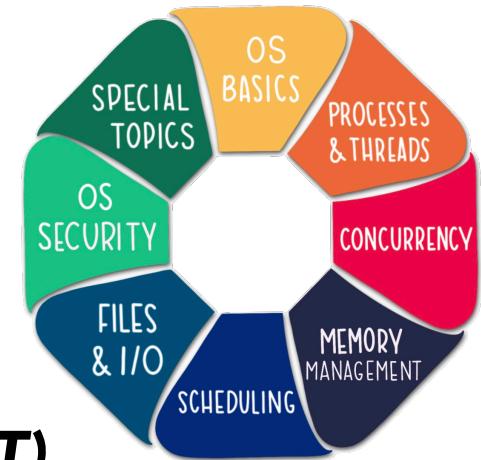
CS 460 - Operating Systems

Fall 2020

<https://www.cs.montana.edu/cs460>

Today

- Announcements
 - Heads up.... PA1 due **Sunday [09/20/2020] @ 11:59 PM (MST)**
- Learning Objectives
 - Understand the big ideas behind the “OS API”
 - ~~user vs. kernel, modes, syscalls, libraries~~
 - Understand the big ideas behind process control
 - [theory] **control info, creation, termination, states, etc.**
 - [reality] ~~fork, exec, getpid, waitpid, etc.~~



PA 1... Don't overthink it! Read directions and study the demos in week04/

exec
execvp

```
int main(void) {
    // Initializations.
    int rc;
    char * photo_in = "photo.jpg";
    char * photo_out = "photo_out.jpg";

    fprintf(stdout, "-> Converting: '%s'\n", photo_in);

    // Convert a photo.
    rc = fork();
    if (0 == rc) {
        // see man execvp. program name, then list of arguments as char strings.
        // arg list must be terminated by a NULL
        // and the zeroth arg is (by convention) the name of the program
        rc = execvp("convert", "convert", "-geometry", "50%", "-monochrome", photo_in, photo_out, NULL);
        fprintf(stderr, "**error: cannot convert '%s' **\n", photo_in);
        exit(-1);
    }

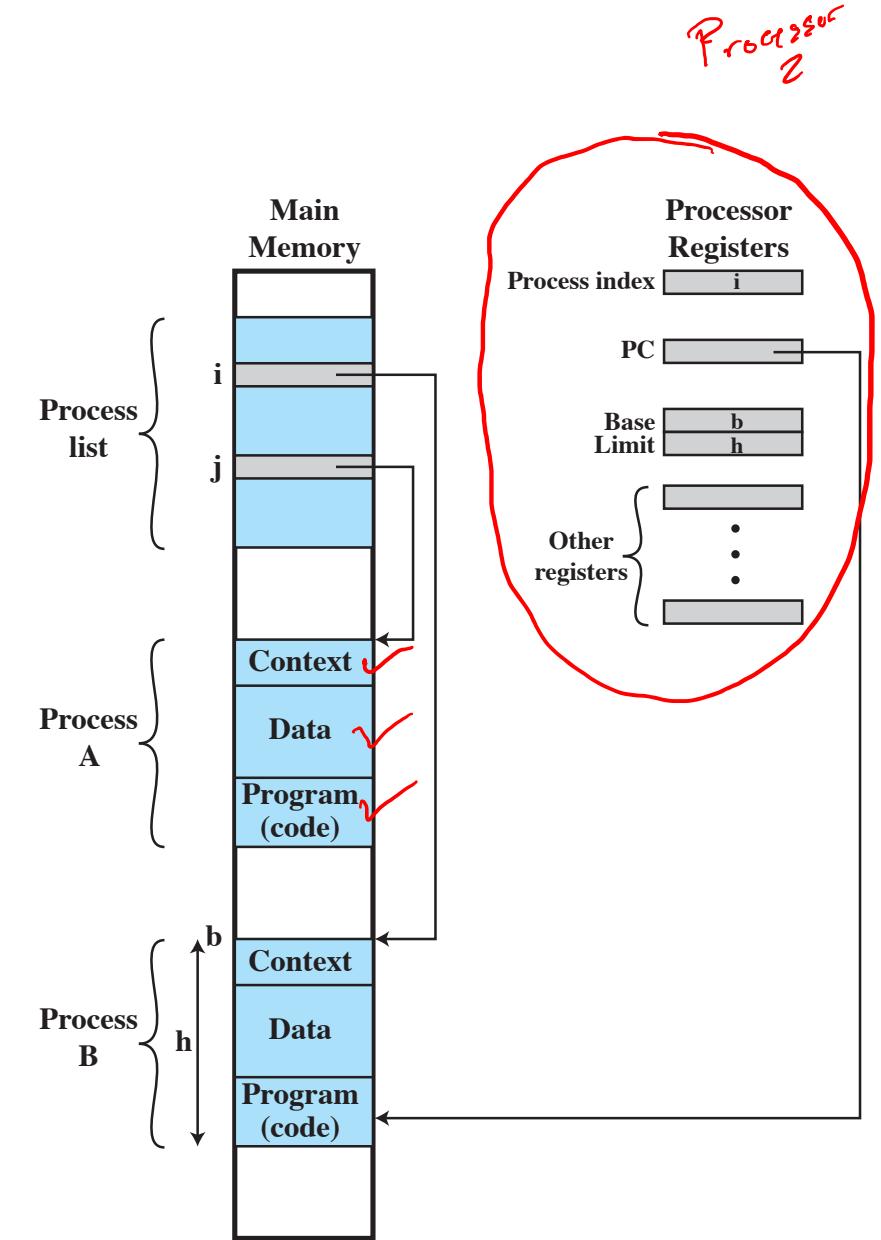
    return 0;
}
```

exec_convert.c

- ✓ **Video demo** (Use TechSmith or YouTube (unlisted!). Link in the README)
- ✓ No code to D2L! Everything must be pushed to **GitHub!** (under **pa1/**)
- ✓ Readable/organized **README**
- ✓ **Read all instructions carefully!** e.g., no **system()** – must use **fork()** and **exec()**!

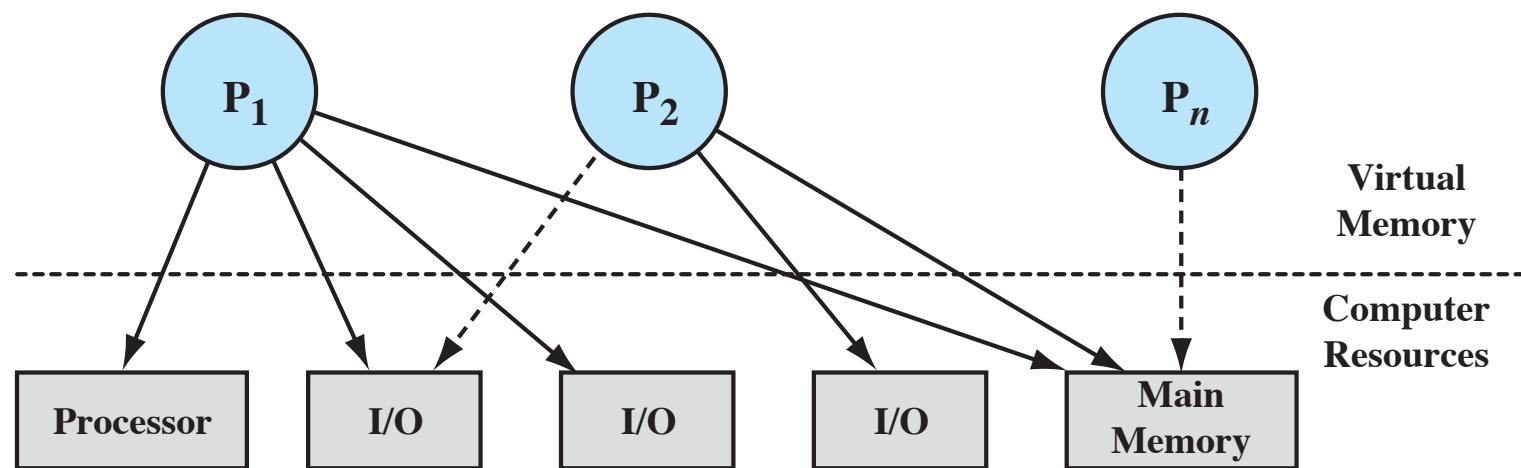
So... Why Use Processes?

- To have a nice abstraction / standardization (code & data)
- To exploit (potential) parallelism] *Threads*
- To exploit pipelining
- To permit concurrency
- To accommodate more than one user
- Convenience for programmers
- Easier portability, extensibility, maintenance
- To allow interleaving / multiprogramming
- etc.



The OS As The Resource Manager

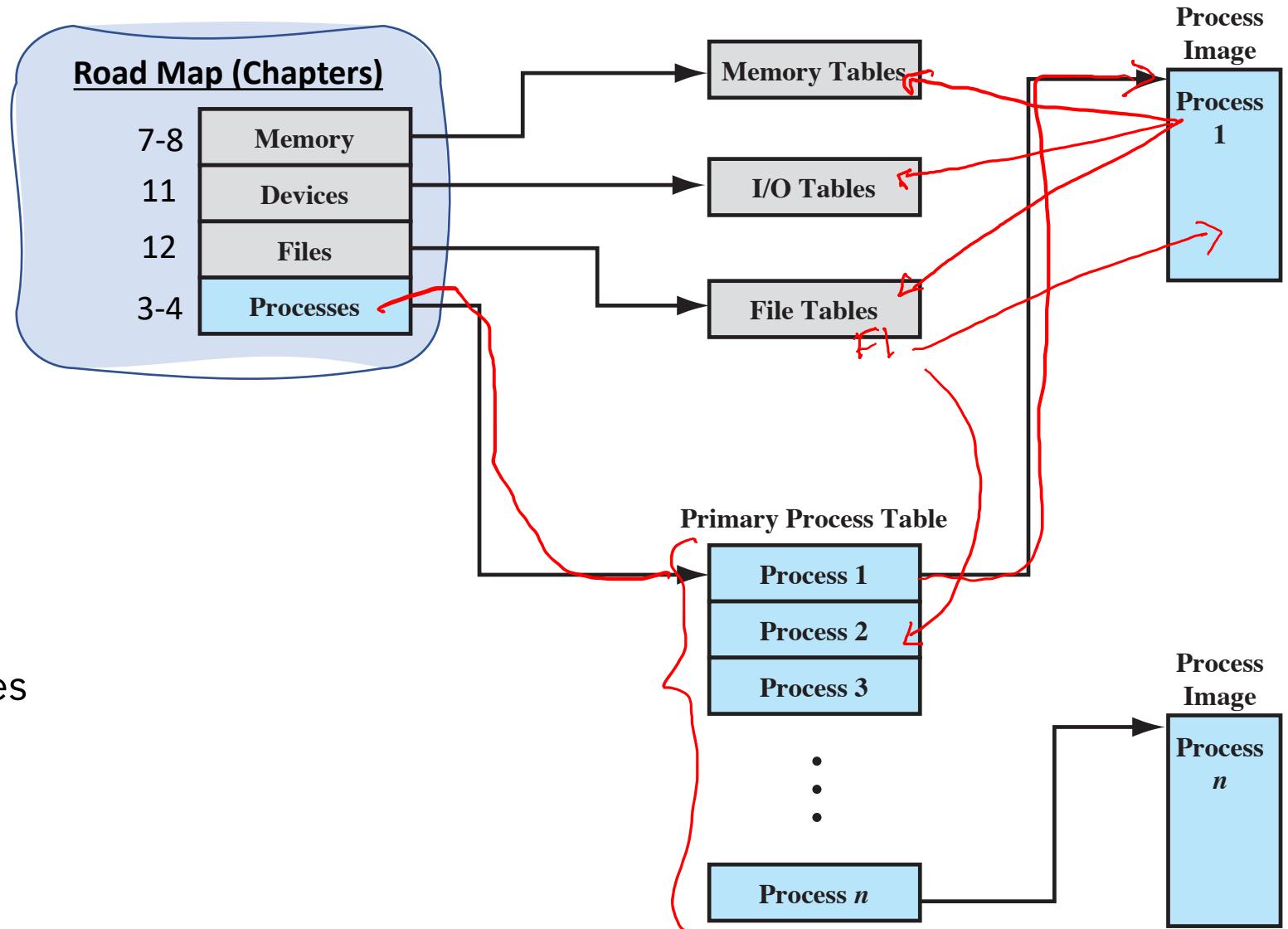
Q: What information does the OS need to know to control processes and manage system resources for them?



- **Process states & data structures**
- **Maintaining information about processes**
- **Address spaces (userspace AND kernelspace)**
- **Reflection/Revisit: Process switching**
- **Reflection/Revisit: Syscalls**

OS Control Structures

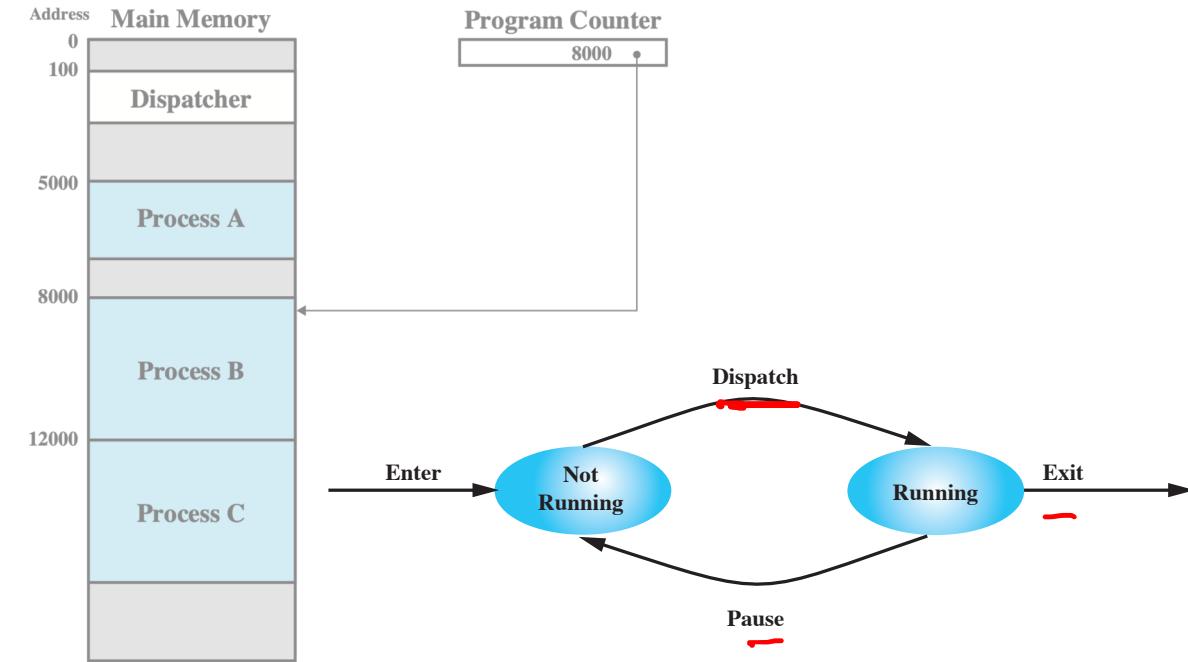
- What does the OS need to know to manage resources?
- The OS maintains tables of information about resources it manages
- There are lots of cross-references between these tables!



Process States & Data Structures

Process States (2 States)

- Simple two-state process model:
 - process is either **running** or **not running**
 - running == on the processor
 - not running == not on the processor



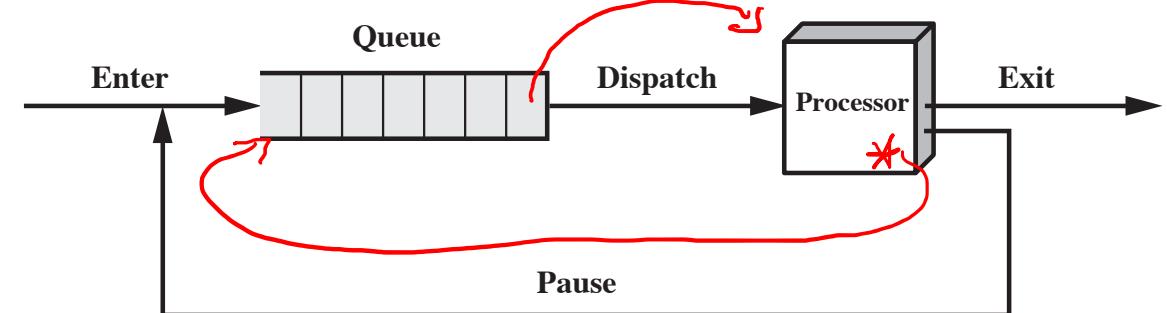
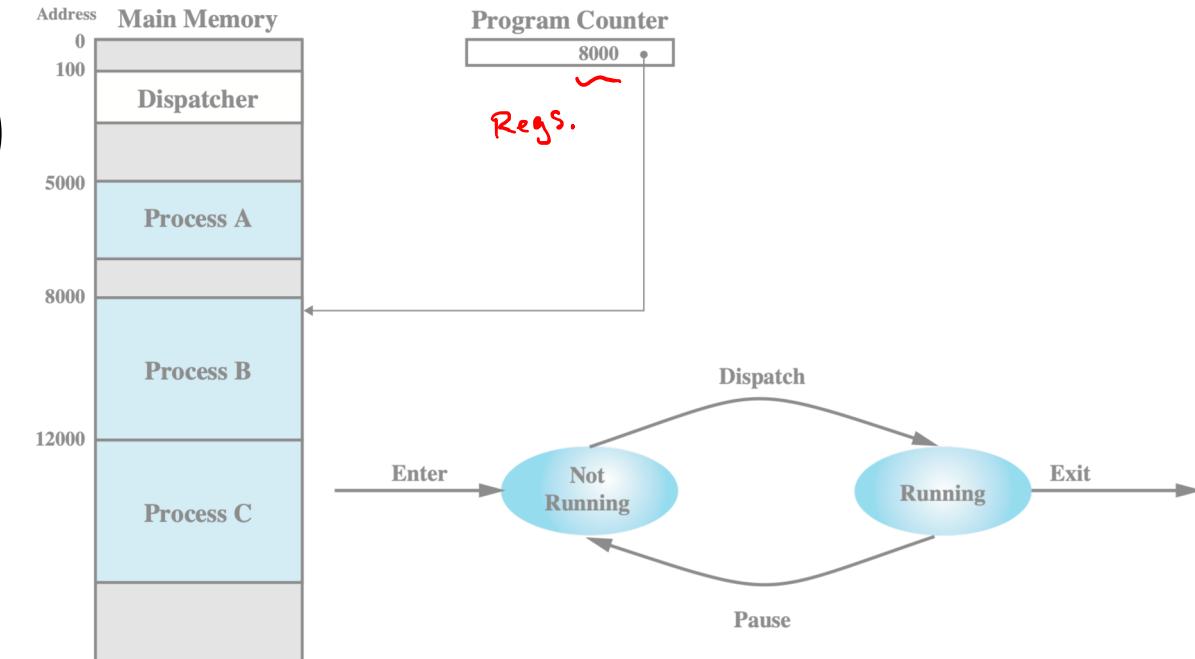
Process States (2 States)

- Simple two-state process model:
 - process is either **running** or **not running**
 - running == on the processor
 - not running == not on the processor

*Q: In a single processor machine,
how many processes can be running?*

Q: What do you do with the other processes?

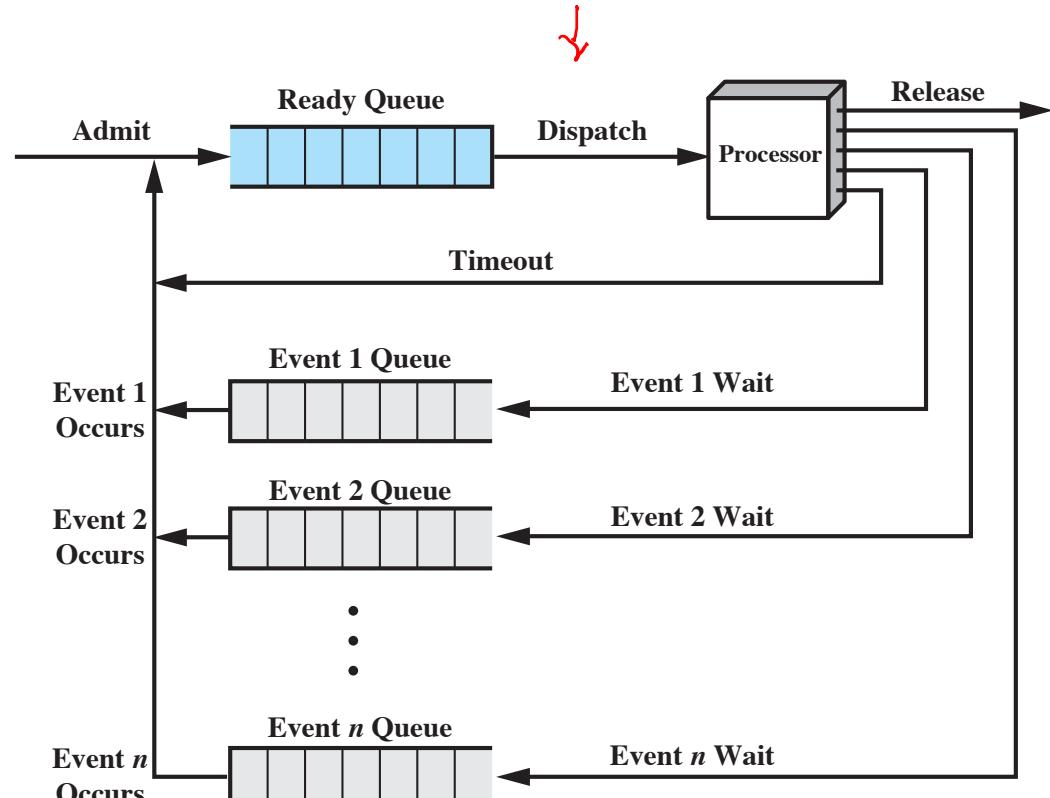
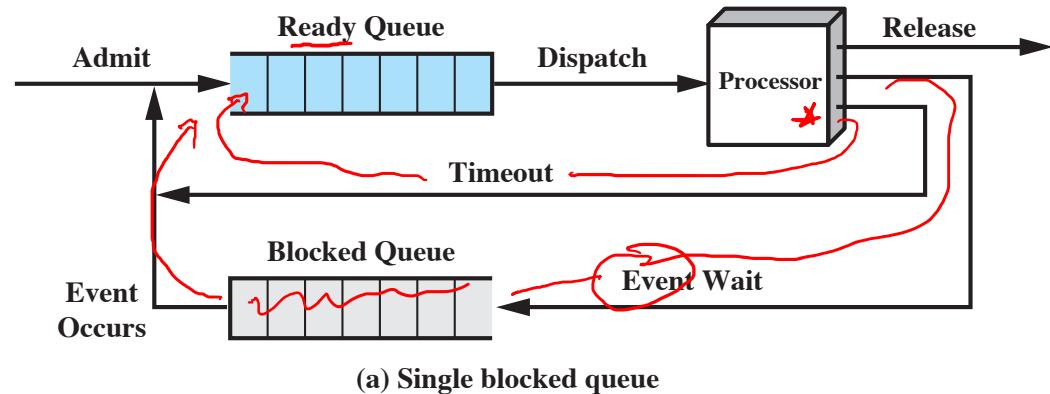
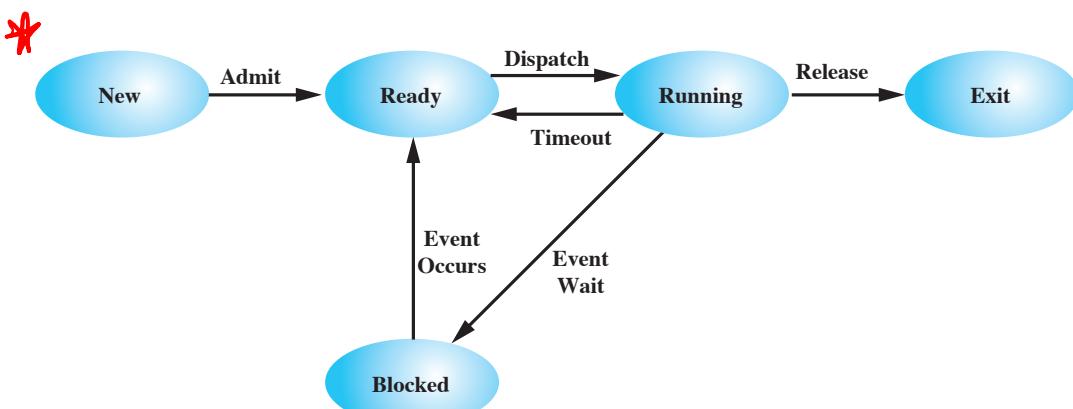
*Q: But what if not all processes are ready to execute?
(What problems could arise with only a single queue?)*



Process States (5 States)

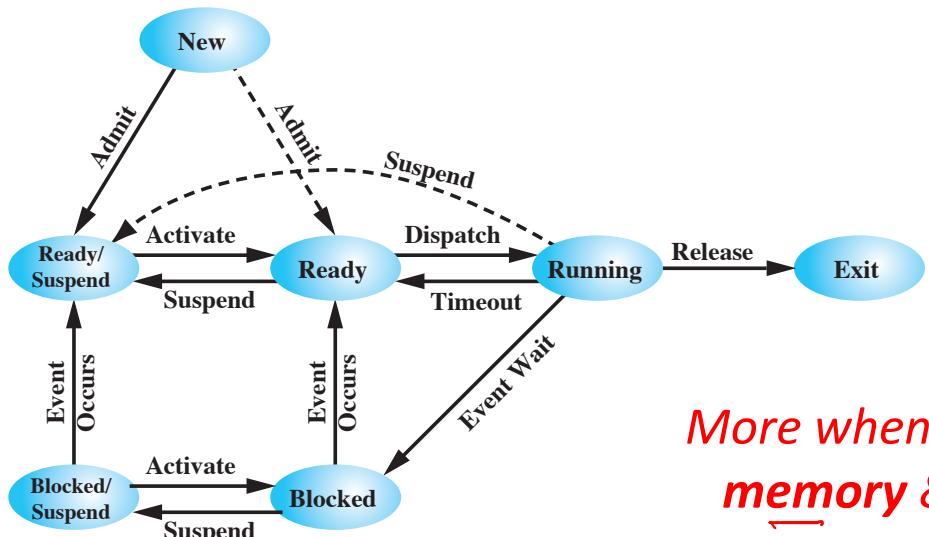
Additional States:

- **New**—the process is being created
- **Ready**—the process is ready to run; waiting to be assigned to a processor
- **Running**—the process is running
- **Blocked**—the process is waiting for an event to occur (e.g., I/O)
- **Exit**—the process has finished execution



See Textbook...

- Step by step “traces” (switching between processes and dispatcher)
- 5-State Process Model + suspension/swapping



*More when we talk about
memory & scheduling*

*	1	5000	P1	27	12004
	2	5001	:	28	12005
	3	5002	:		----- Timeout
	4	5003	:		
	5	5004	:		
	6	5005	:		
	7	100	Timeout	29	100
	8	101	{ P1	30	101
	9	102		31	102
	10	103		32	103
	11	104		33	104
	12	105		34	105
	13	8000	P2	35	5006
	14	8001	:	36	5007
	15	8002	:	37	5008
	16	8003	:	38	5009
	17	100	I/O Request	39	5010
	18	101		40	5011
	19	102			----- Timeout
	20	103			
	21	104			
	22	105			
	23	12000	P3	41	100
	24	12001		42	101
	25	12002		43	102
	26	12003		44	103
	27	12004		45	104
	28	12005		46	105
	29	100		47	12006
	30	101		48	12007
	31	102		49	12008
	32	103		50	12009
	33	104		51	12010
	34	105		52	12011
	35	5006			----- Timeout
	36	5007			
	37	5008			
	38	5009			
	39	5010			
	40	5011			

100 = Starting address of dispatcher program

Shaded areas indicate execution of dispatcher process;
first and third columns count instruction cycles;
second and fourth columns show address of instruction being executed

Process Context

The Details of a “Process”

- The **Process Control Block (PCB)**

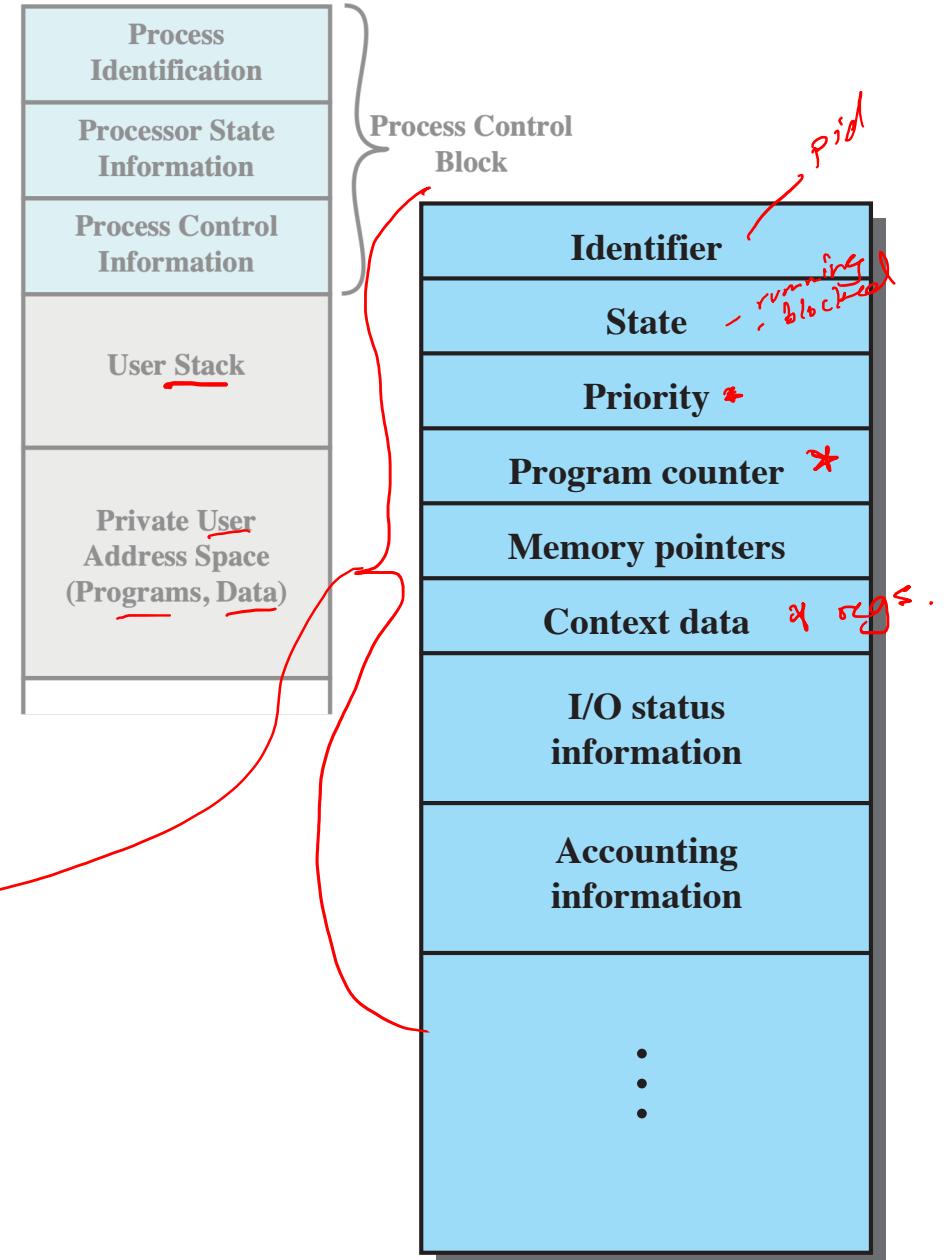
- Holds the **“context”**
- A **“snapshot”** that contains all necessary information to restart a process where it left off.



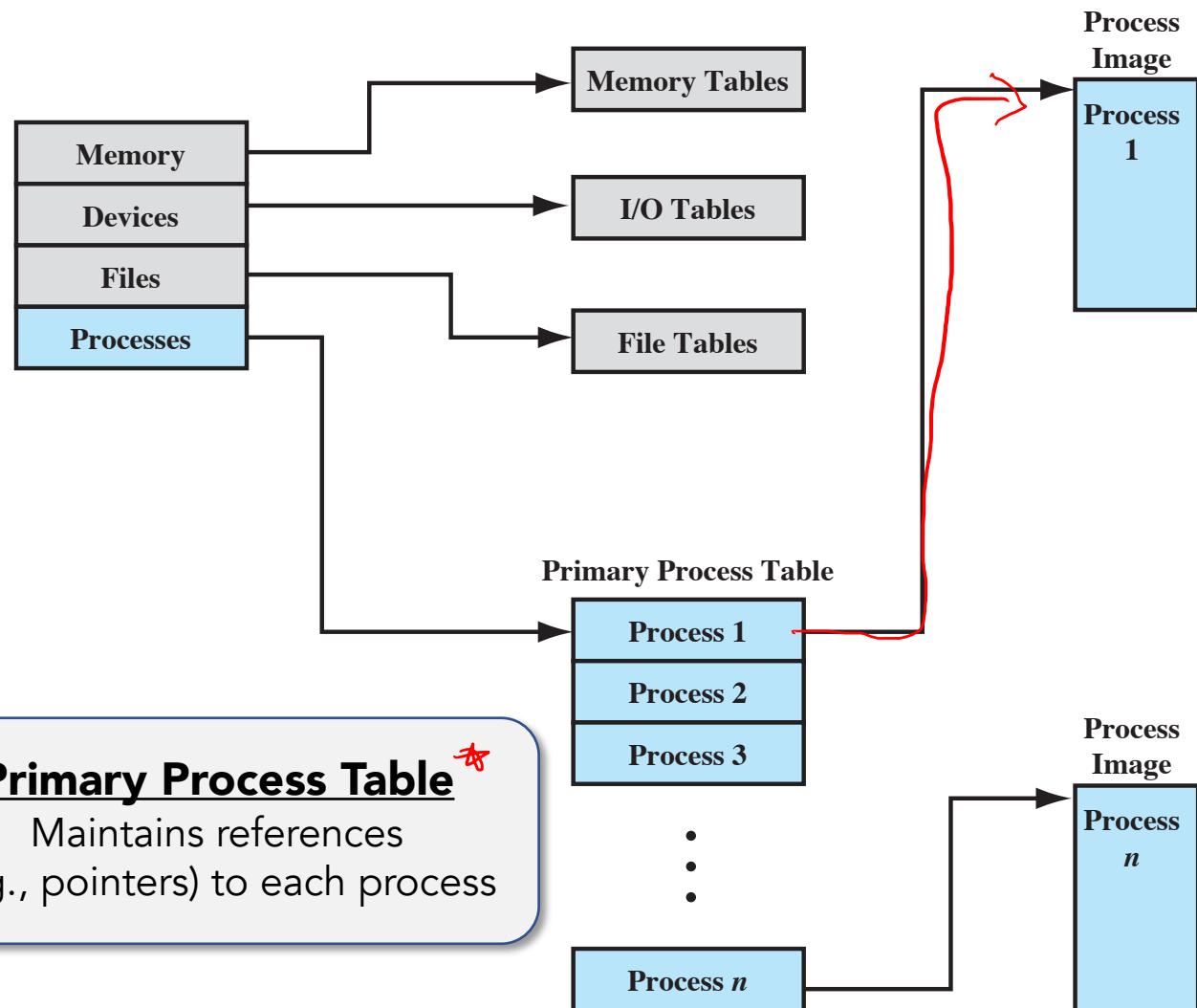
Textbooks call it a **“PCB,”** but many real-world OSs use a different name for this structure (e.g., task_struct)

- We can get all sorts of info about procs.

- getpid(), getppid(), getpriority(), ...
- Also: ps, top, htop, procfs



The Details of a “Process”



Process Image
Collection of code, data, stack, and attributes (PCB)

Process Attributes

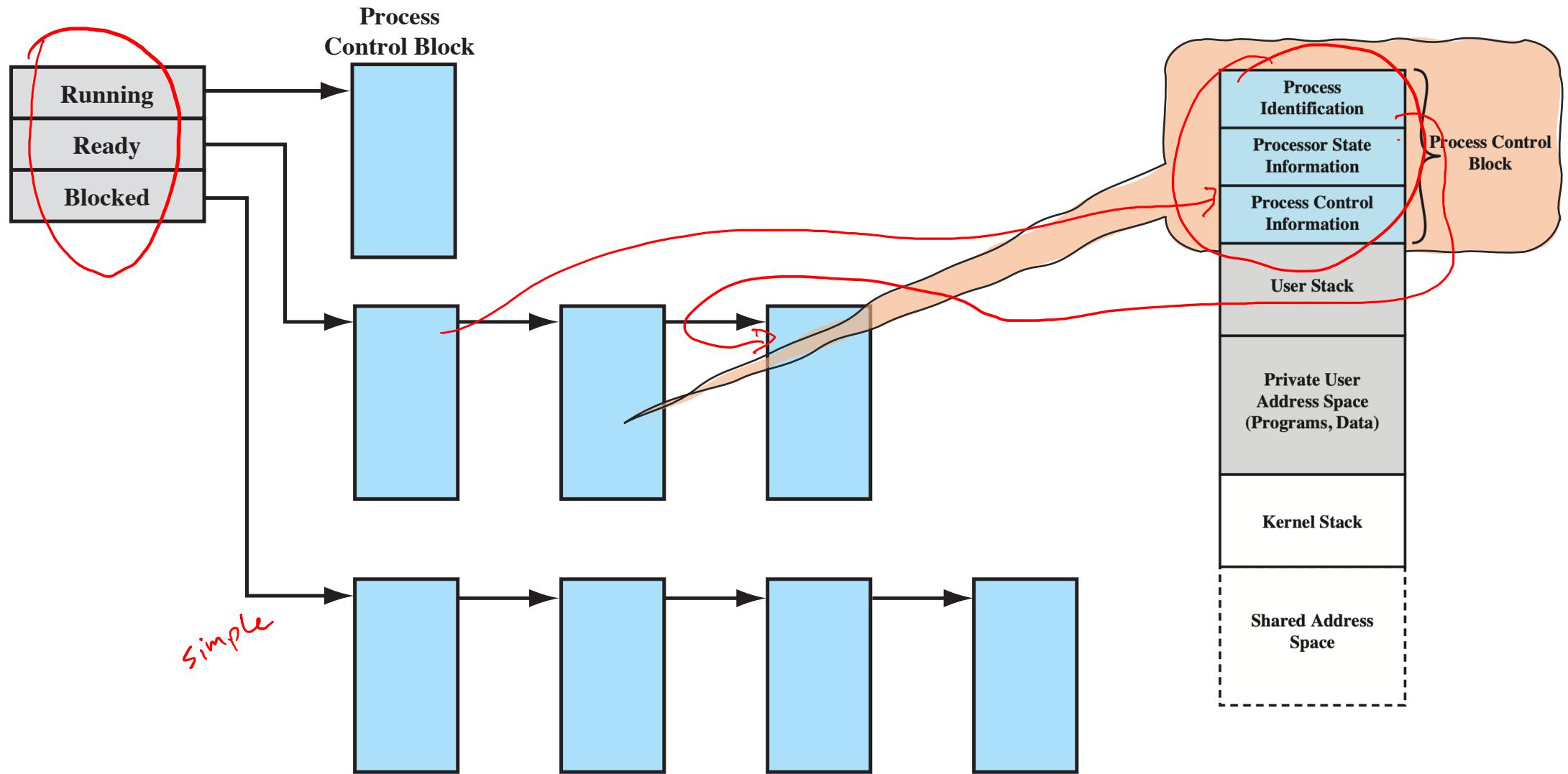
1. Proc. ID Info
PID, PPID, UID, etc.

2. Processor State Info
Registers (User-visible, Status, Control), Stack Pointers, etc.
(maintains info after process is interrupted!)

Identifier
State
Priority
Program counter
Memory pointers
Context data
I/O status information
Accounting information
⋮

3. Proc. Control Info
Scheduling & State Info, IPC, Privileges, Files, Page Tables (Memory), etc.

Recall Process States



Summary: The Role of the PCB

- PCB is the most important data structure in a modern OS
 - contains ALL of the information about a process that is needed by the OS
 - Blocks are read and/or modified by virtually every module in the OS
 - Defines the state of the OS
- Q: What could go wrong?!
- Difficulty is **not access, but protection**
 - A bug in a single OS routine could damage PCBs, which could destroy the system's ability to manage the affected processes
 - A design change in the structure or semantics of the PCB could affect many modules in the OS
 - Must tightly control access to the PCB...
...all PCB changes happen through a designated handler routine

