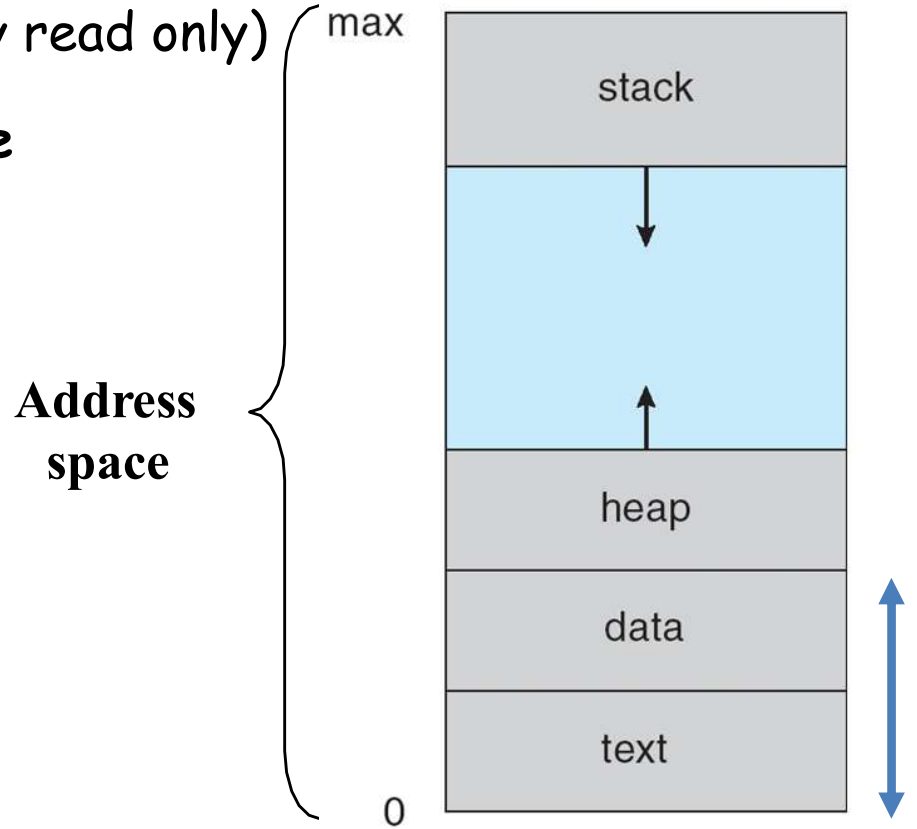
The Process Concept

The Process Concept

- **Process – a program in execution**
  - Program
    - description of how to perform an activity
    - instructions and static data values
    - Located in Disk
  - Process
    - a snapshot of a program in execution
    - memory (program instructions, static and dynamic data values)
    - CPU state (registers, PC, SP, etc)
    - operating system state (open files, accounting statistics etc)

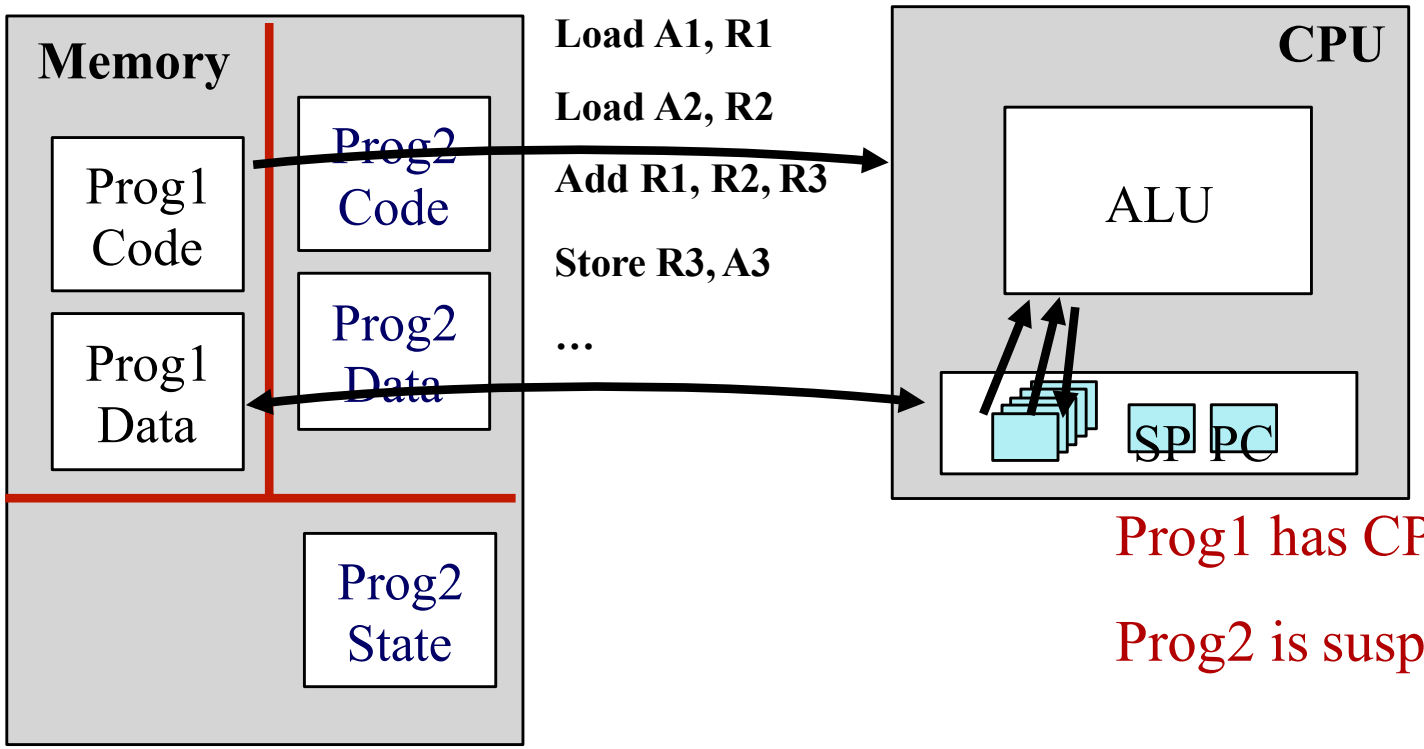
Process address space

- **Each process runs in its own virtual memory *address space* that consists of:**
  - *Stack space* – used for function and system calls
  - Heap space – used for dynamic memory allocations
  - *Data space* – Static and global variables
  - *Text* – the program code (usually read only)
- **Invoking the same program multiple times results in the creation of multiple distinct address spaces**



Switching among multiple processes

- Program instructions operate on operands in memory and (temporarily) in registers

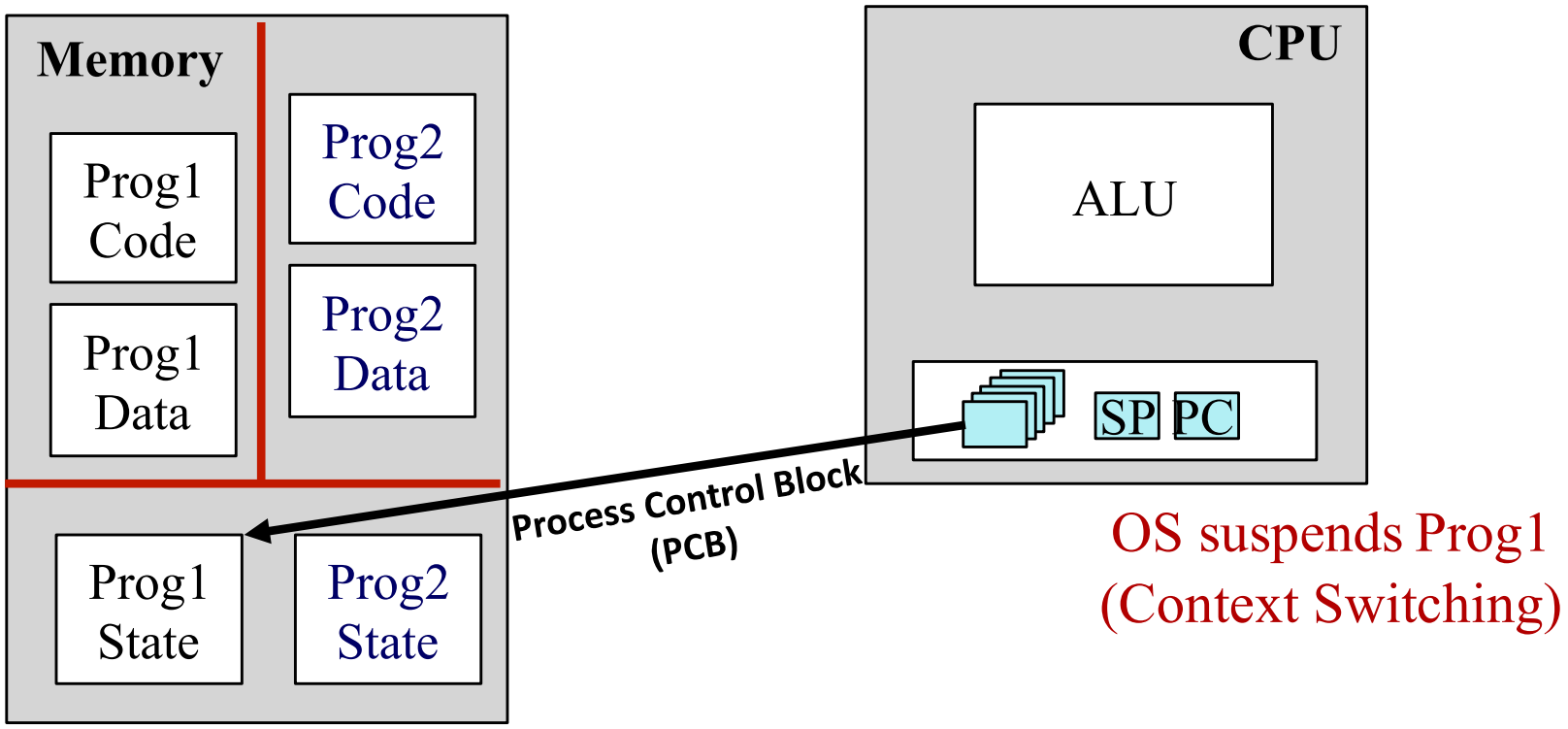


Prog1 has CPU

Prog2 is suspended

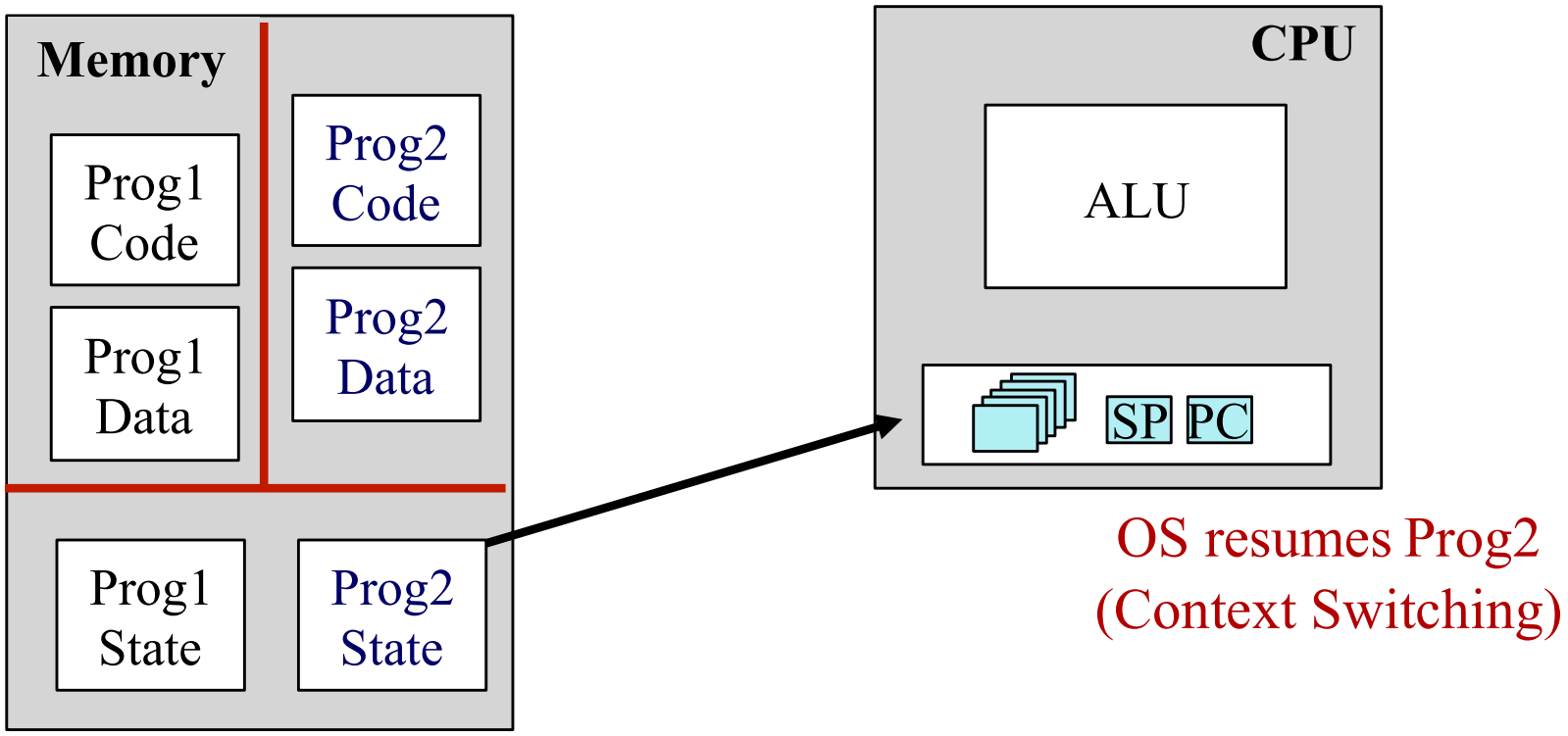
Switching among multiple processes

- Saving all the information about a process allows a process to be *temporarily suspended* and later *resumed* from the same point



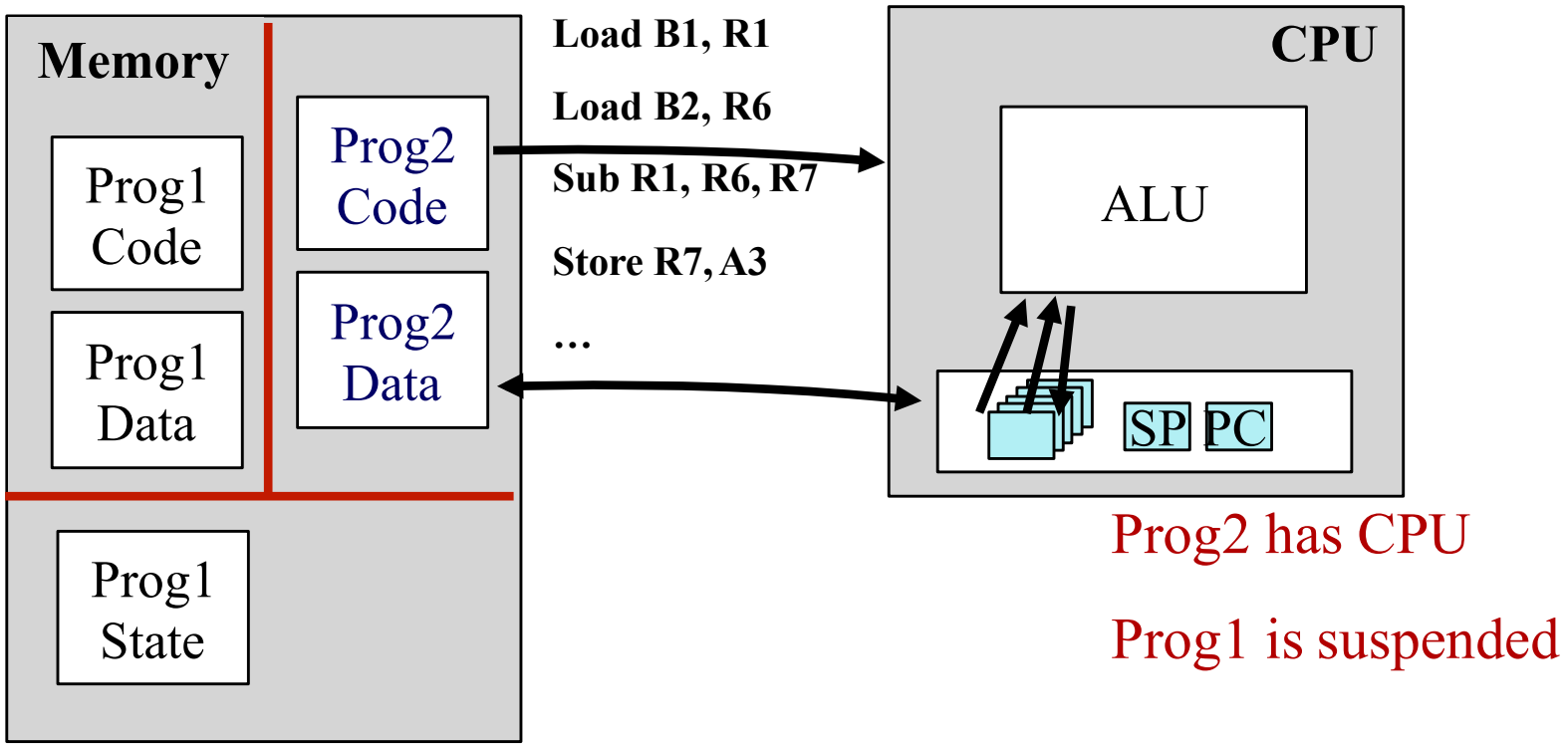
Switching among multiple processes

- Saving all the information about a process allows a process to be *temporarily suspended* and later *resumed*

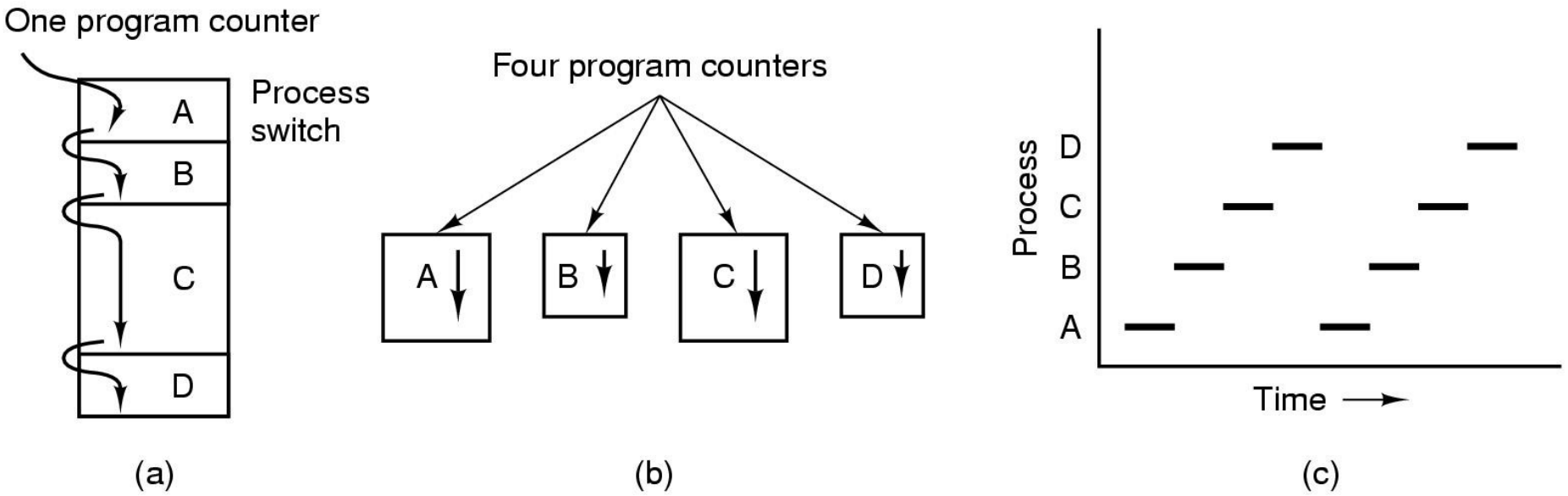


Switching among multiple processes

- Program instructions operate on operands in memory and in registers

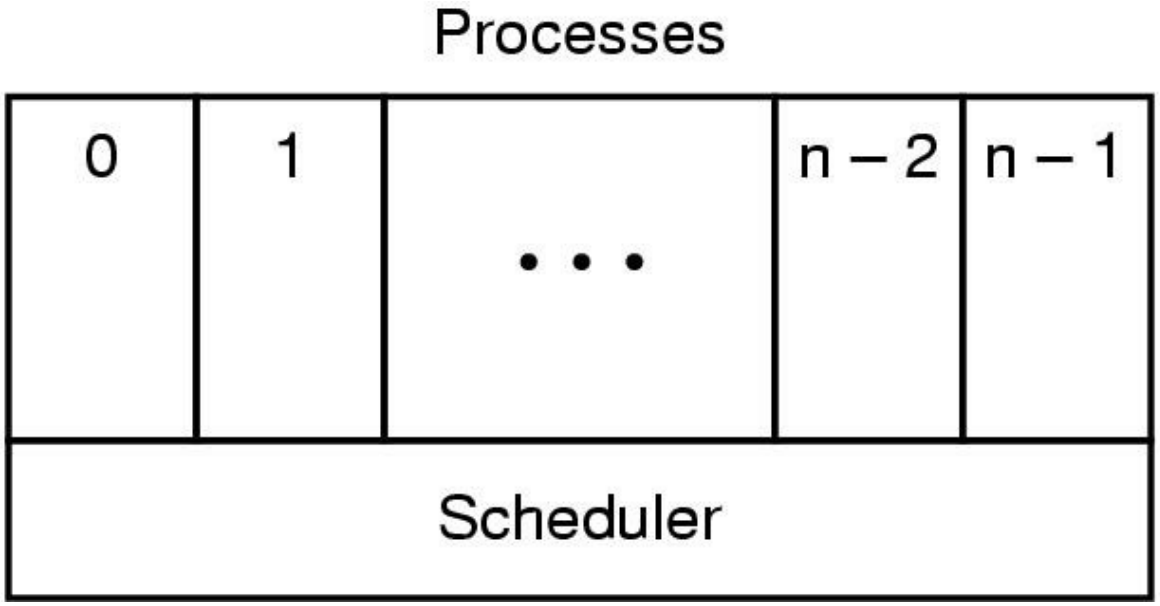


Why use the process abstraction?



- Multiprogramming of four programs in the same address space
- Conceptual model of 4 independent, sequential processes
- Only one program active at any instant

The role of the scheduler



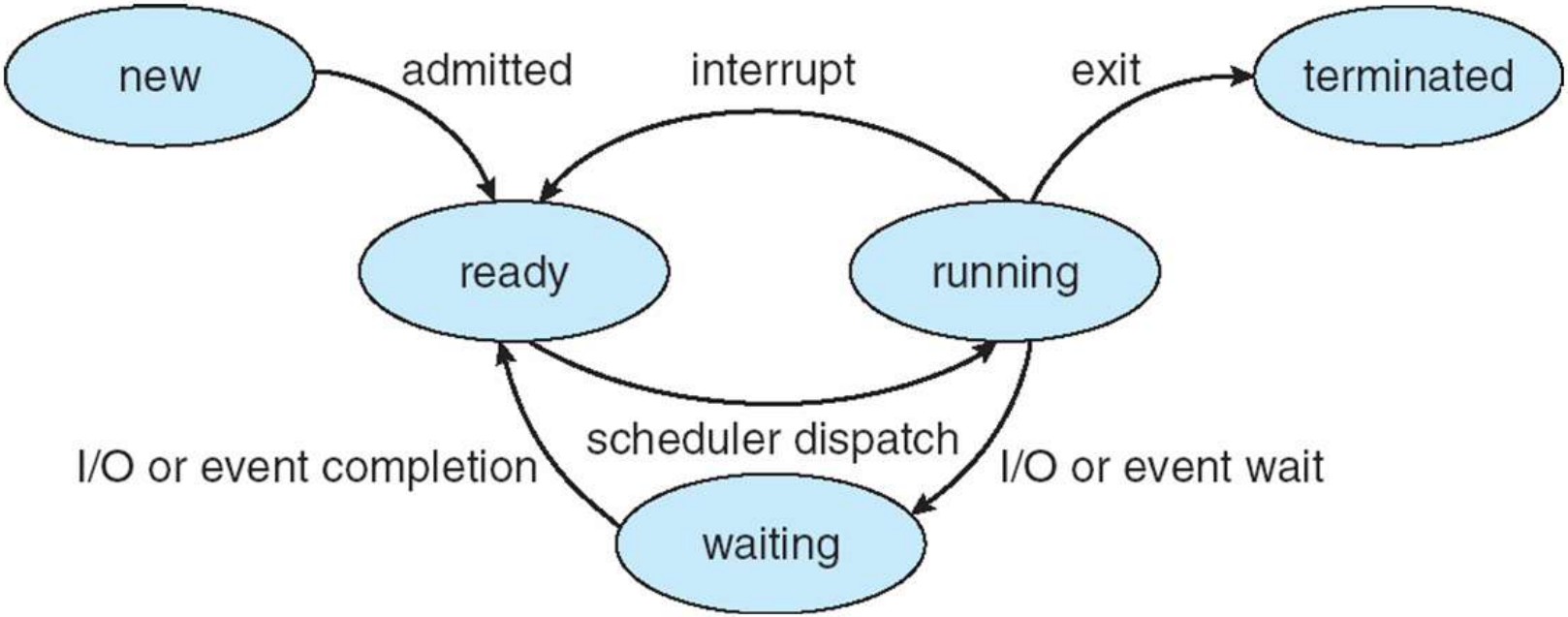
- **Lowest layer of process-structured OS**
  - handles interrupts & scheduling of processes
- **Sequential processes only exist above that layer**

Process State

As a process executes, it changes state

- **new**: The process is being created
- **running**: Instructions are being executed
- **waiting**: The process is waiting for some event to occur
- **ready**: The process is waiting to be assigned to a processor
- **terminated**: The process has finished execution

Diagram of Process State



How do processes get created?

Principal events that cause process creation

- System initialization
- Initiation of a batch job
- User request to create a new process
- Execution of a process creation system call from another process

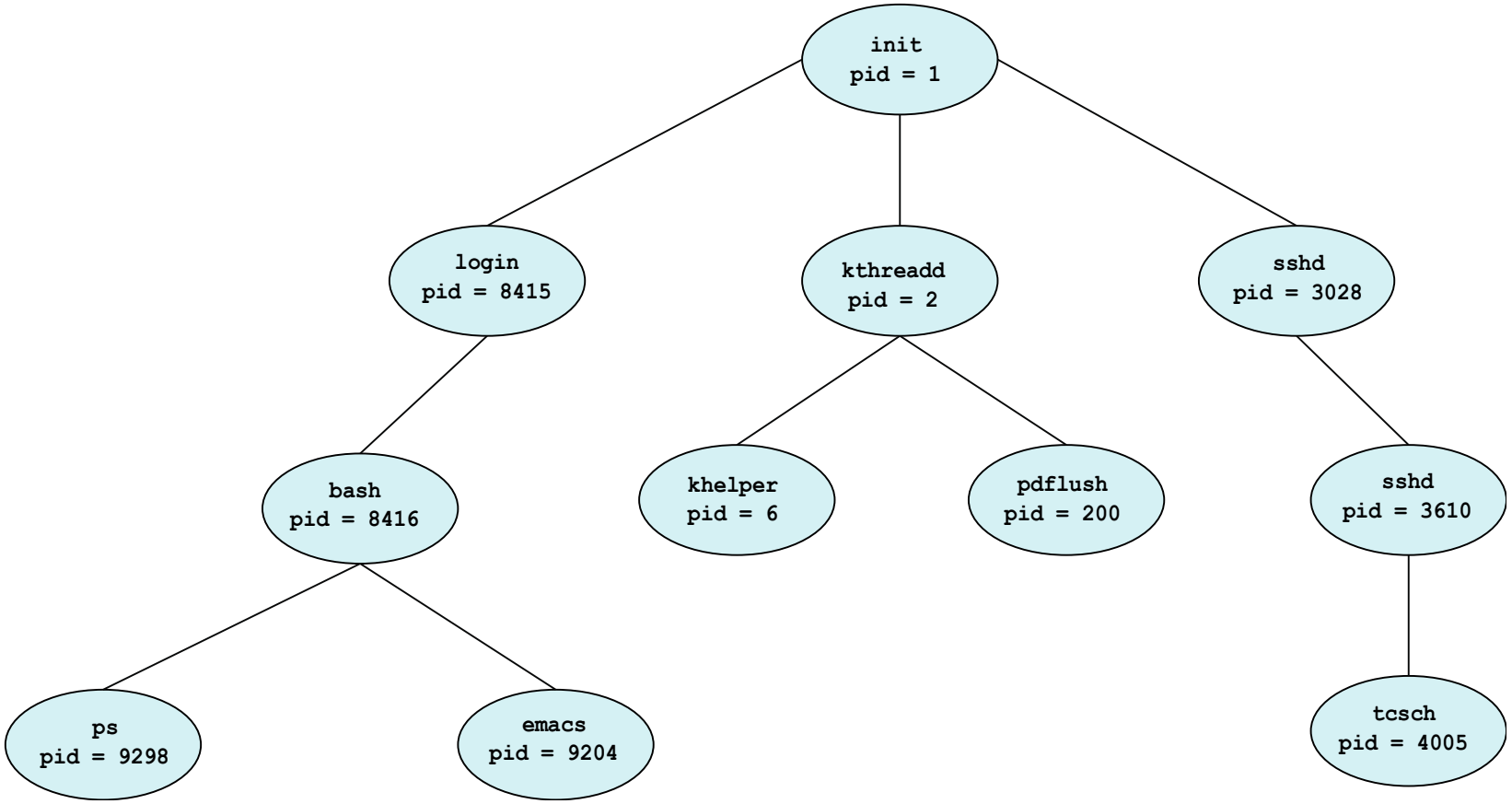
Process hierarchies

- **Parent creates a child process,**
  - special system calls for communicating with and waiting for child processes
  - each process is assigned a unique identifying number or process ID (PID)
- **Child processes can create their own child processes**
  - Forms a hierarchy
  - UNIX calls this a "process group"
  - Windows has no concept of process hierarchy

Process creation in UNIX

- **All processes have a unique process id**
  - *getpid(), getppid()* system calls allow processes to get their information
- **Process creation**
  - *fork()* system call creates a copy of a process and returns in both processes, but with a different return value
  - *exec()* replaces an address space with a new program
- **Process termination, signaling**
  - *signal(), kill()* system calls allow a process to be terminated or have specific signals sent to it

A Tree of Processes in Linux



Example: process creation in UNIX

csh (pid = 22)

```
…

chpid = fork()
if (chpid == 0)
{
   // child…
   …
   exec();
   }
else {
   // parent
   wait();
… }
```

Process creation in UNIX example

csh (pid = 22)

```
…
chpid = fork()
if (chpid == 0)
{
   // child…
   …
   exec();
   }
else {
   // parent
   wait();
… }
```

csh (pid = 24)

```
…
chpid = fork()
if (chpid == 0)
{
   // child…
   …
   exec();
   }
else {
   // parent
   wait();
… }
```

Process creation in UNIX example

csh (pid = 22)

```
…
chpid = fork()
if (chpid == 0)
{
   // child…
   …
   exec();
   }
else {
   // parent
   wait();
   }
…
```

csh (pid = 24)

```
…
chpid = fork()
if (chpid == 0)
{
   // child…
   …
   exec();
   }
else {
   // parent
   wait();
   }
…
```

Process creation in UNIX example

csh (pid = 22)

```
…
chpid = fork()
if (chpid == 0)
{
   // child…
   …
   exec();
   }
else {
   // parent
   wait();
   }
…
```

csh (pid = 24)

```
…
chpid = fork()
if (chpid == 0)
{
   // child…
   …
   exec();
   }
else {
   // parent
   wait();
   }
…
```

Process creation in UNIX example

csh (pid = 22)

```
…

chpid = fork()
if (chpid == 0) {
   // child…
   …
   exec();
   }
else {
   // parent
   wait();
   }
…
```

ls (pid = 24)

```
//ls program

main(){

   //look up dir

   …

}
```

How do processes terminate?

Conditions which terminate processes

- Normal exit (voluntary)
- Error exit (voluntary)
- Fatal error (involuntary)
- Killed by another process (involuntary)

Process Termination

- In some systems, terminating a parent process will also terminate all its child processes
- We call this case cascading termination
- Otherwise, if cascading termination is not enforced, a child process where its parent is terminated will be called an Orphan process
- Linux allows orphan processes, and assigns the *init* process to be the parent of all orphan processes
- If a child process that has terminated, but its parent hasn't yet called wait(), then we call it a Zombie process

What other process state does the OS manage?

Process management	**Memory management**	**File management**
Registers	Pointer to text segment	Root directory
Program counter	Pointer to data segment	Working directory
Program status word	Pointer to stack segment	File descriptors
Stack pointer		User ID
Process state		Group ID
Priority		
Scheduling parameters		
Process ID		
Parent process		
Process group		
Signals		
Time when process started		
CPU time used		
Children's CPU time		
Time of next alarm		

Example fields of a process table entry