Processes

CS 416: Operating Systems Design, Spring 2011

Department of Computer Science
Rutgers University

Von Neuman Model

Both text (program) and data reside in memory

Execution cycle

- Fetch instruction
- Decode instruction
- Execute instruction

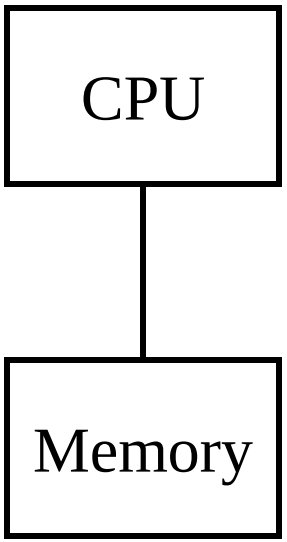


Image of Executing Program

100 mov (R1), R2

104 add R1,4,R1

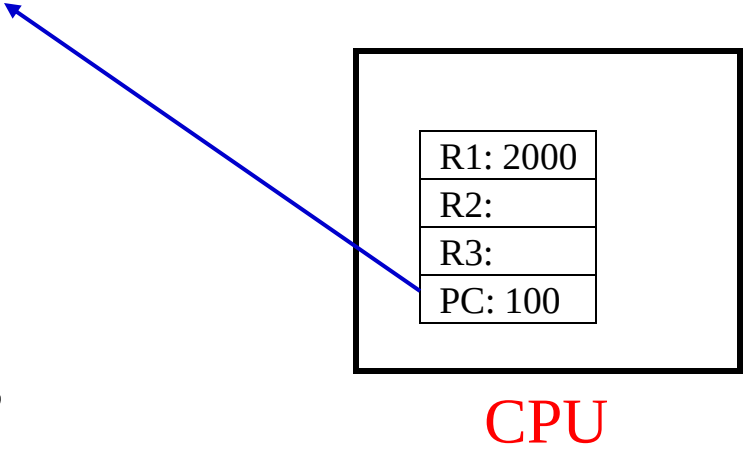
108 Mov (R1), R3

112 add R2,R3,R3

Memory

2000 4

2004 8



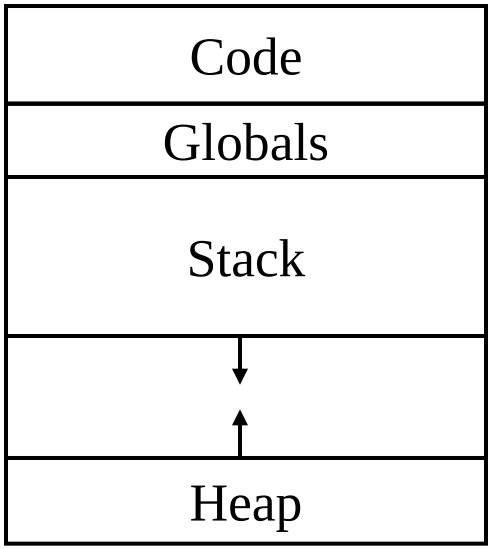
Higher-Level Languages

```
public class foo {

    static private int yv = 0;
    static private int nv = 0;

    public static void main() {
       foo foo_obj = new foo;
       foo_obj->cheat();
    }

    public cheat() {
       int tyv = yv;
       yv = yv + 1;
       if (tyv < 10) {
           cheat();
       }
    }
}
```

How to map a program like this to a Von Neuman machine?

- Where to keep yv, nv?
- What about foo_obj and tyv?
- How to do foo_obj->cheat()?

Run Time Storage Organization



Memory

Each variable must be assigned a storage class

Global (static) variables

- Allocated in globals region at compile-time

Method local variables and parameters

- Allocate dynamically on stack

Dynamically created objects (using new)

- Allocate from heap
- Objects live beyond invocation of a method
- Garbage collected when no longer “live”

Pointer to next instruction to be executed kept in special register called PC

- Variables also cached in registers

Process

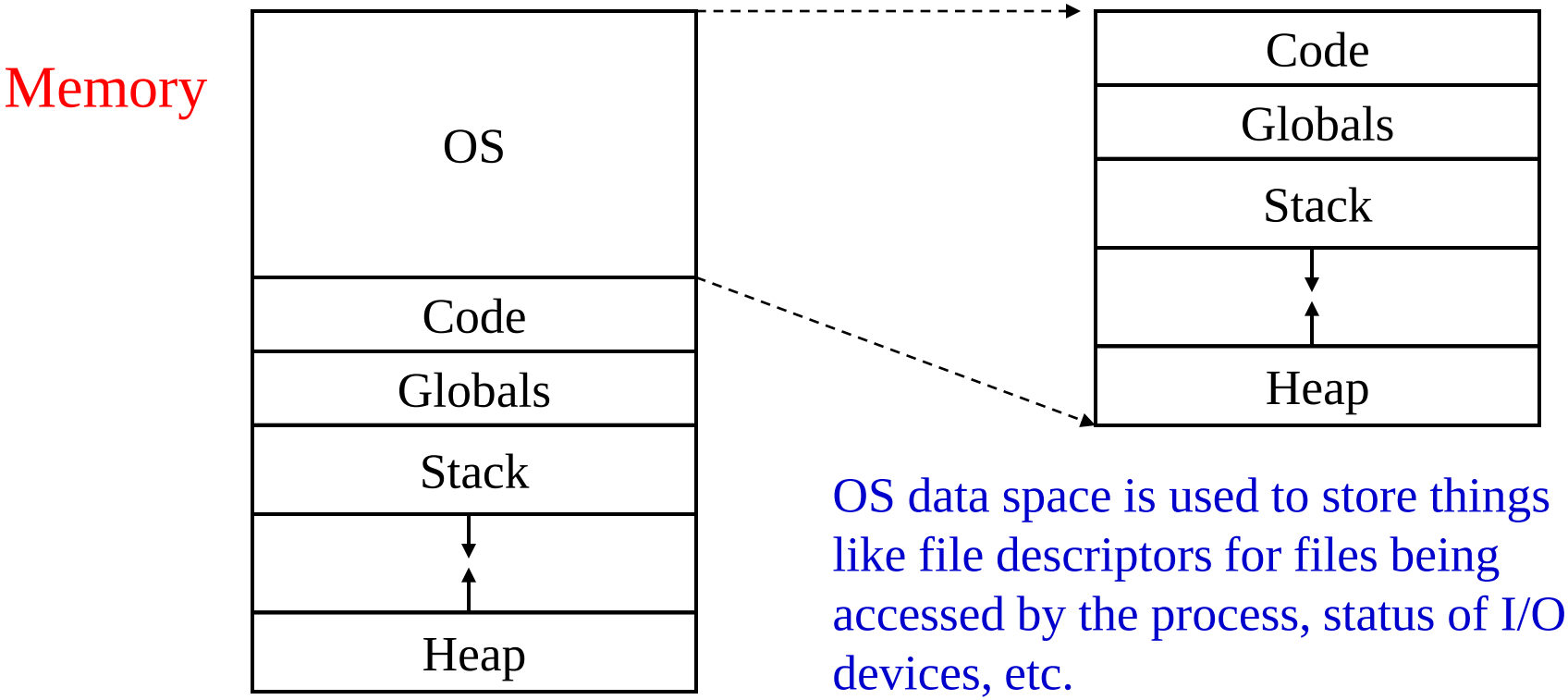
Process = system abstraction for the set of resources required for executing a program
= a running instance of a program
= memory image + registers' content (+ I/O state)

The stack + registers' content represent the *execution context* or *thread of control*

What About The OS?

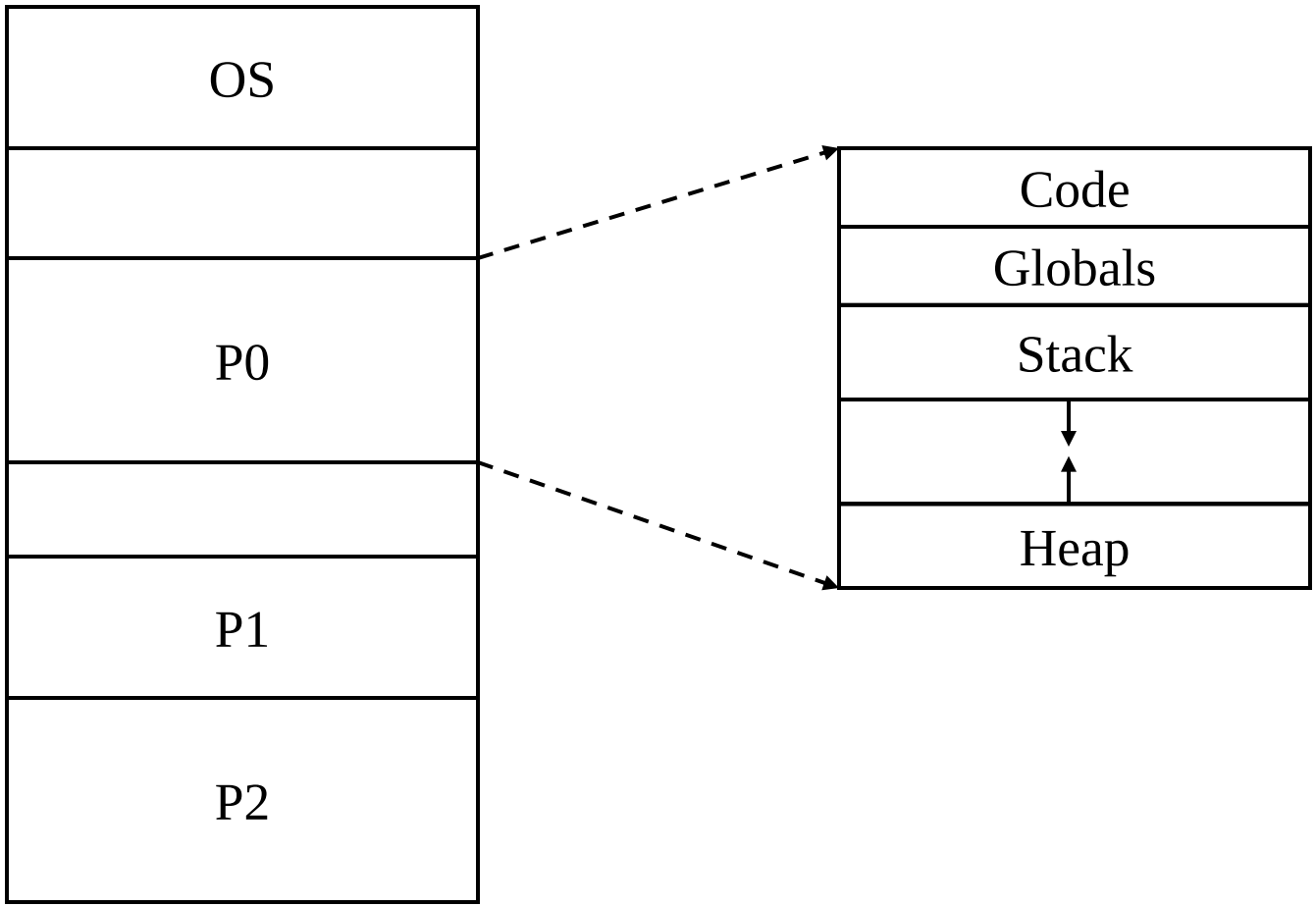
Recall that one of the function of an OS is to provide a virtual machine interface that makes programming the machine easier

So, a process memory image must also contain the OS



OS data space is used to store things like file descriptors for files being accessed by the process, status of I/O devices, etc.

What Happens When There Are More Than One Running Process?



Process Control Block

Each process has per-process state maintained by the OS

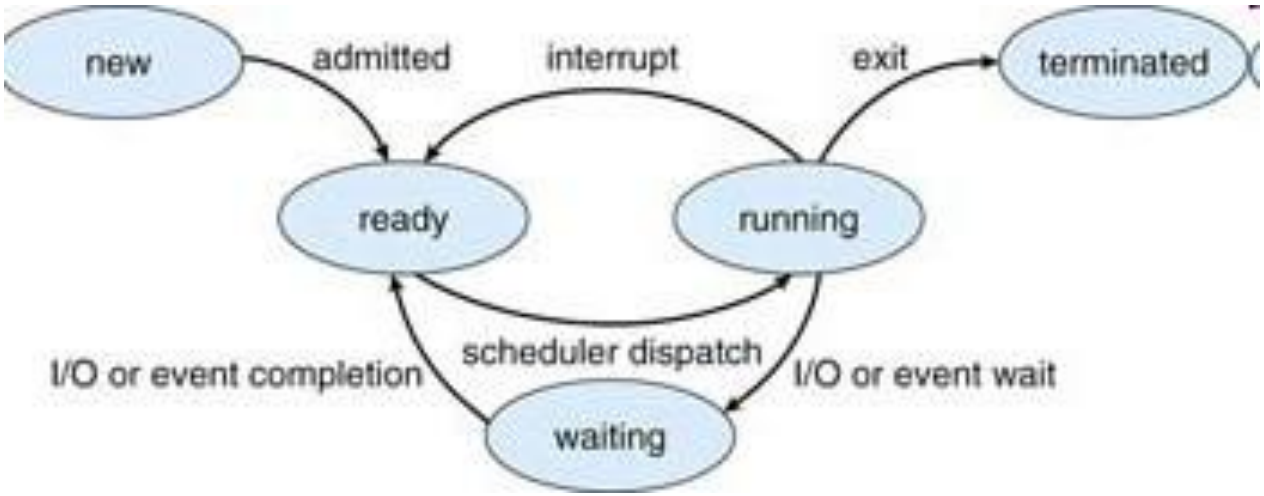
- Identification: process, parent process, user, group, etc.
- Execution contexts: threads
- Address space: virtual memory
- I/O state: file handles (file system), communication endpoints (network), etc.
- Accounting information

For each process, this state is maintained in a *process control block* (PCB)

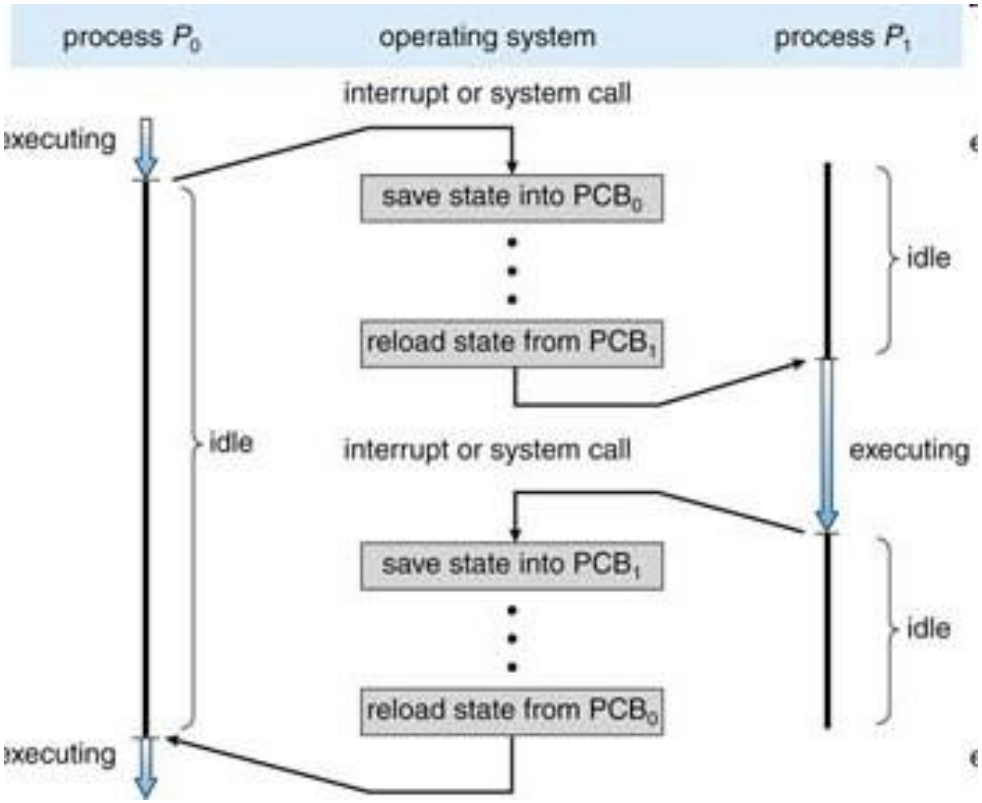
- This is just data in the OS data space
- Think of it as objects of a class

Process Control Block

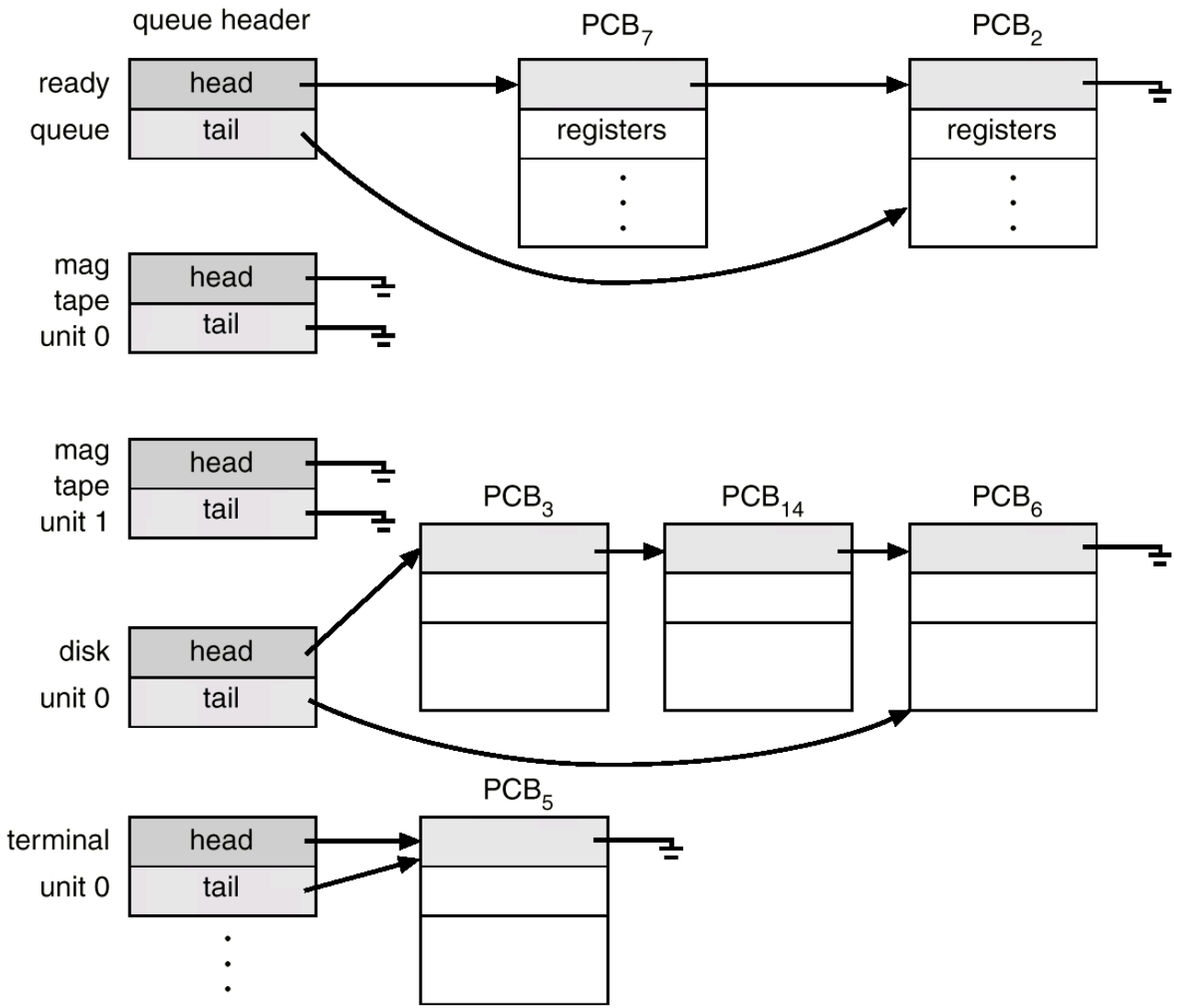
Process States



Switching Between Processes



Ready Queue And Various I/O Device Queues



Process Creation

How to create a process? System call.

In UNIX, a process can create another process using the fork() system call

```
int pid = fork();           /* this is in C */
```

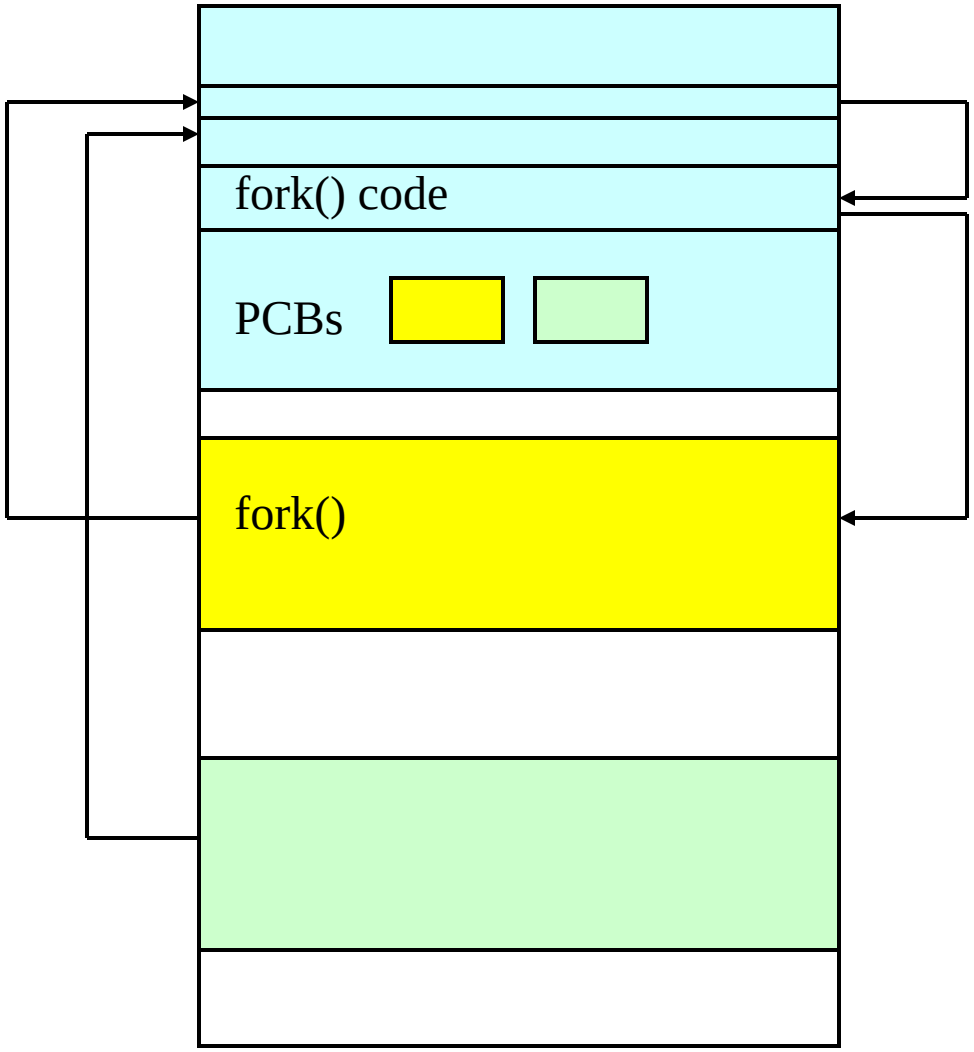
The creating process is called the parent and the new process is called the child

The child process is created as a copy of the parent process (process image and process control structure) except for the identification and scheduling state

- Parent and child processes run in two different address spaces
- By default, there's no memory sharing
- Process creation is expensive because of this copying

The exec() call is provided for the newly created process to run a different program than that of the parent

Process Creation



Example of Process Creation Using Fork

The UNIX shell is command-line interpreter whose basic purpose is for user to run applications on a UNIX system

cmd arg1 arg2 ... argn

```
while(TRUE) {
  get_command(command, parameters)

  if(fork() != 0) {      /* parent */
    wait(&status);
  } else {               /* child */
    exec(command, parameters)
  }
}
```

Process Death (or Murder)

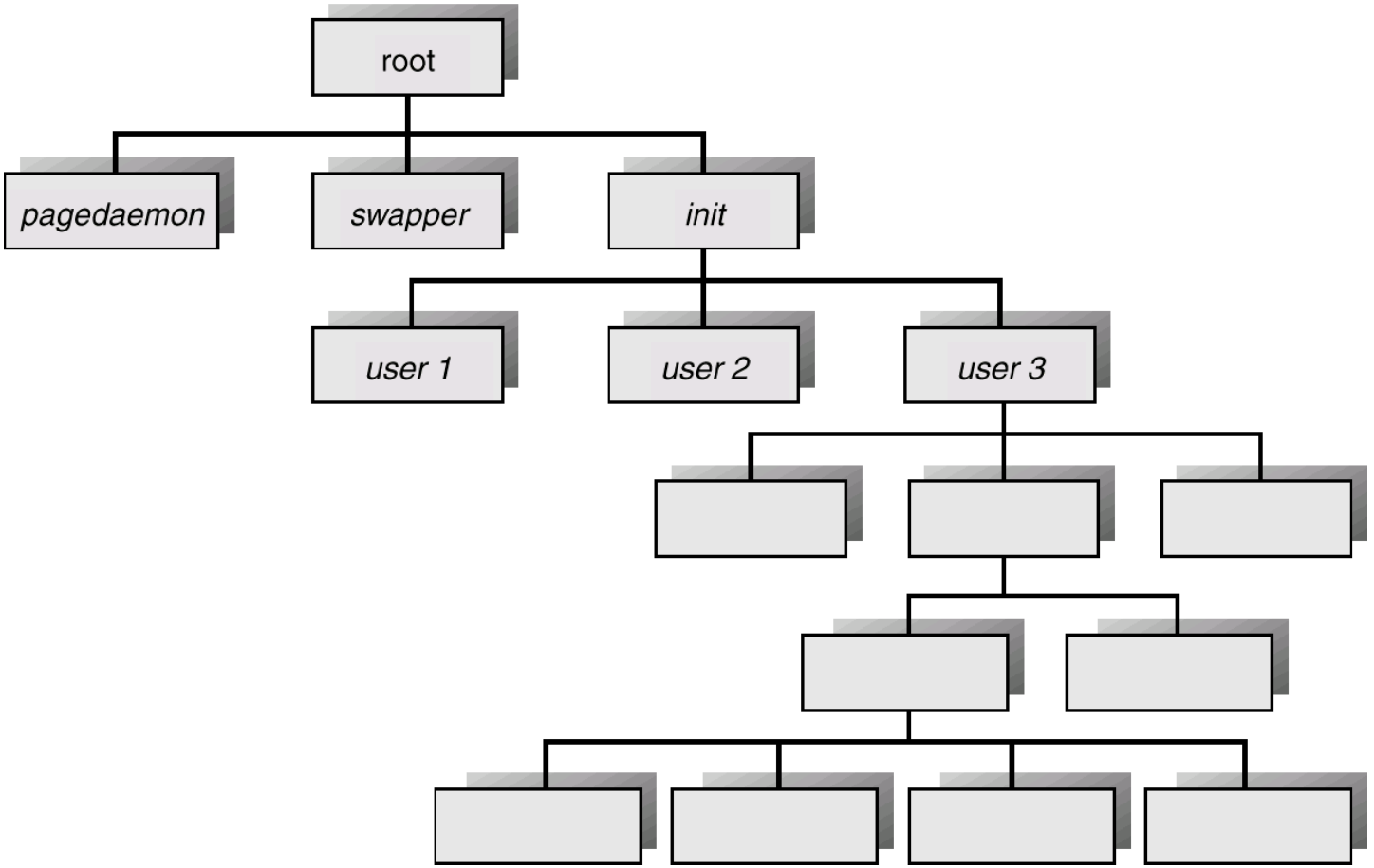
One process can wait for another process to finish using the wait() system call

- Can wait for a child to finish as shown in the example
- Can also wait for an arbitrary process if know its PID

Can kill another process using the kill() system call

- What all happens when kill() is invoked?
- What if the victim process doesn't want to die?

A Tree of Processes On A Typical UNIX System



Signals

User program can invoke OS services by using system calls

What if the program wants the OS to notify it *asynchronously* when some event occurs?

Signals

UNIX mechanism for OS to notify a user program when an event of interest occurs

Potentially interesting events are predefined: e.g., segmentation violation, message arrival, kill, etc.

When interested in "handling" a particular event (signal), a process indicates its interest to the OS and gives the OS a procedure that should be invoked in the upcall

How does a process "indicate" its interest in handling a signal?

Signals (Cont'd)

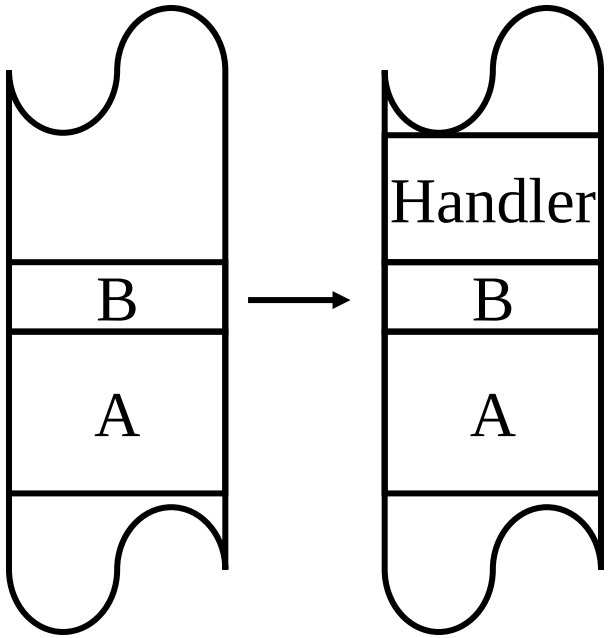
When an event of interest occurs:

The kernel handles the event first, then modifies the process's stack to look as if the process's code made a procedure call to the signal handler.

Puts an activation record on the user-level stack corresponding to the event handler

When the user process is scheduled next it executes the handler first

From the handler the user process returns to where it was when the event occurred



Process: Summary

An "instantiation" of a program

System abstraction: the set of resources required for executing a program

- Execution context(s)
- Address space
- File handles, communication endpoints, etc.

Historically, all of the above "lumped" into a single abstraction

More recently, split into several abstractions

- Threads, address space, protection domain, etc.

OS process management:

- Supports user creation of processes and interprocess communication (IPC)
- Allocates resources to processes according to specific policies
- Interleaves the execution of multiple processes to increase system utilization