OS Overview

Contents

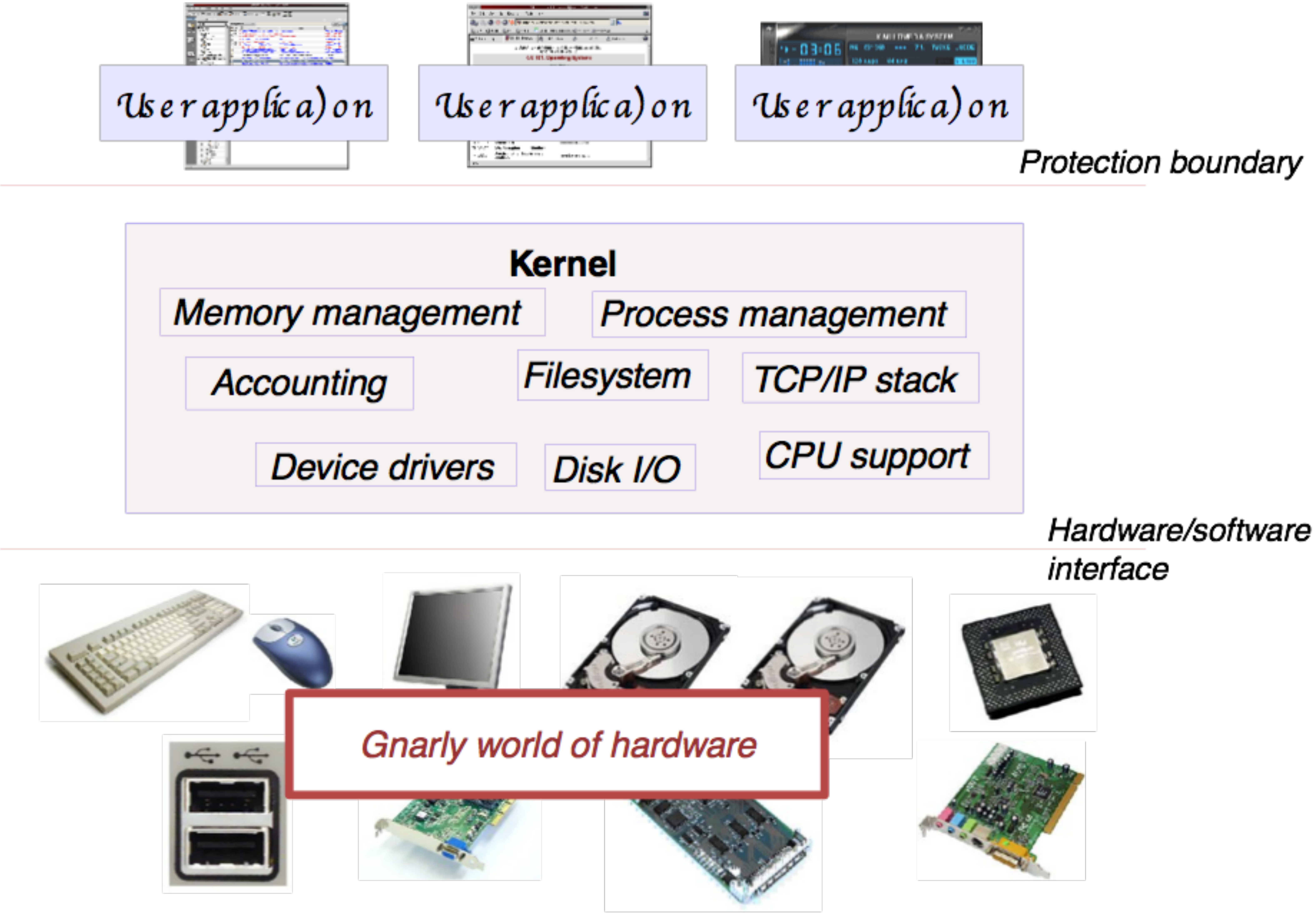
- What is an operating system?
- OS functions

What is an Operating System?

- A program that acts as an intermediary between a user of a computer and the computer hardware
- Operating system goals:
  - Execute user programs and simplify solving user problems.
  - Make the computer system convenient to use
  - Use the computer hardware efficiently

What is an operating system?

Software that provides an elaborate illusion to applications



User application

User application

User application

Protection boundary

Kernel

Memory management

Process management

Accounting

Filesystem

TCP/IP stack

Device drivers

Disk I/O

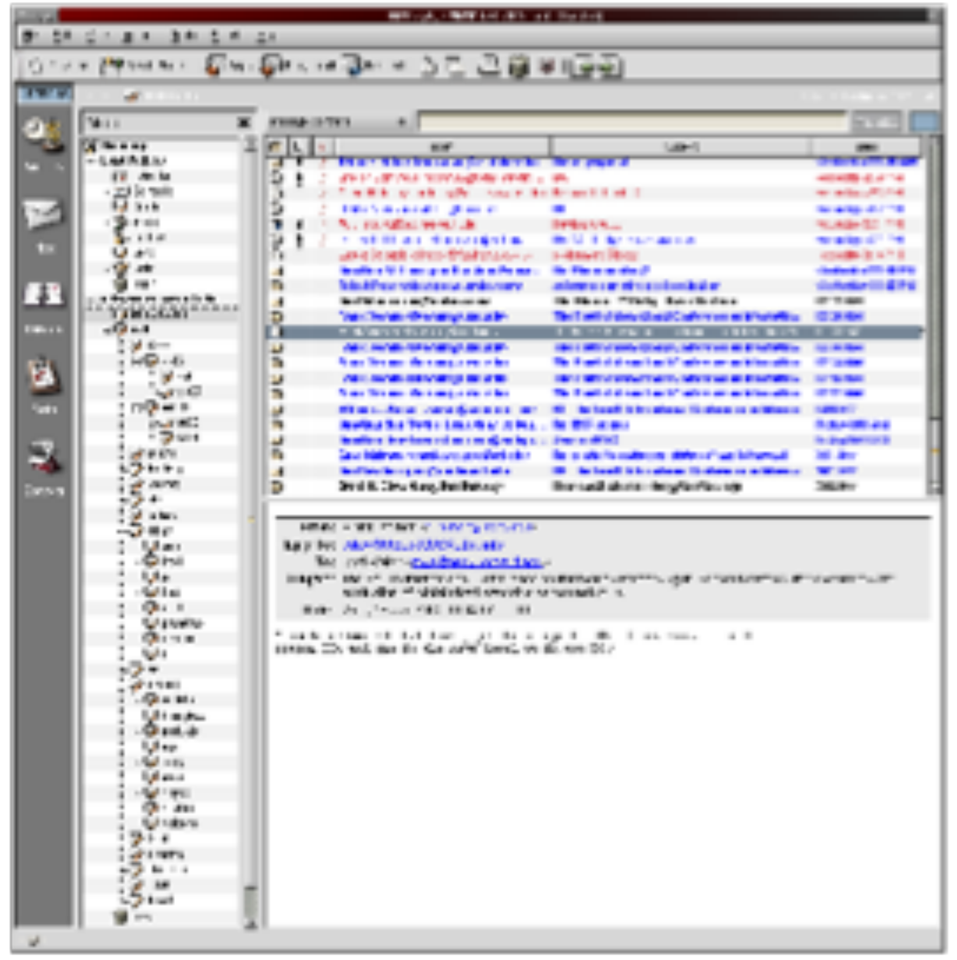
CPU support

Hardware/software interface

Gnarly world of hardware

One OS Function: Concurrency

Give every application the illusion of having its own CPU!



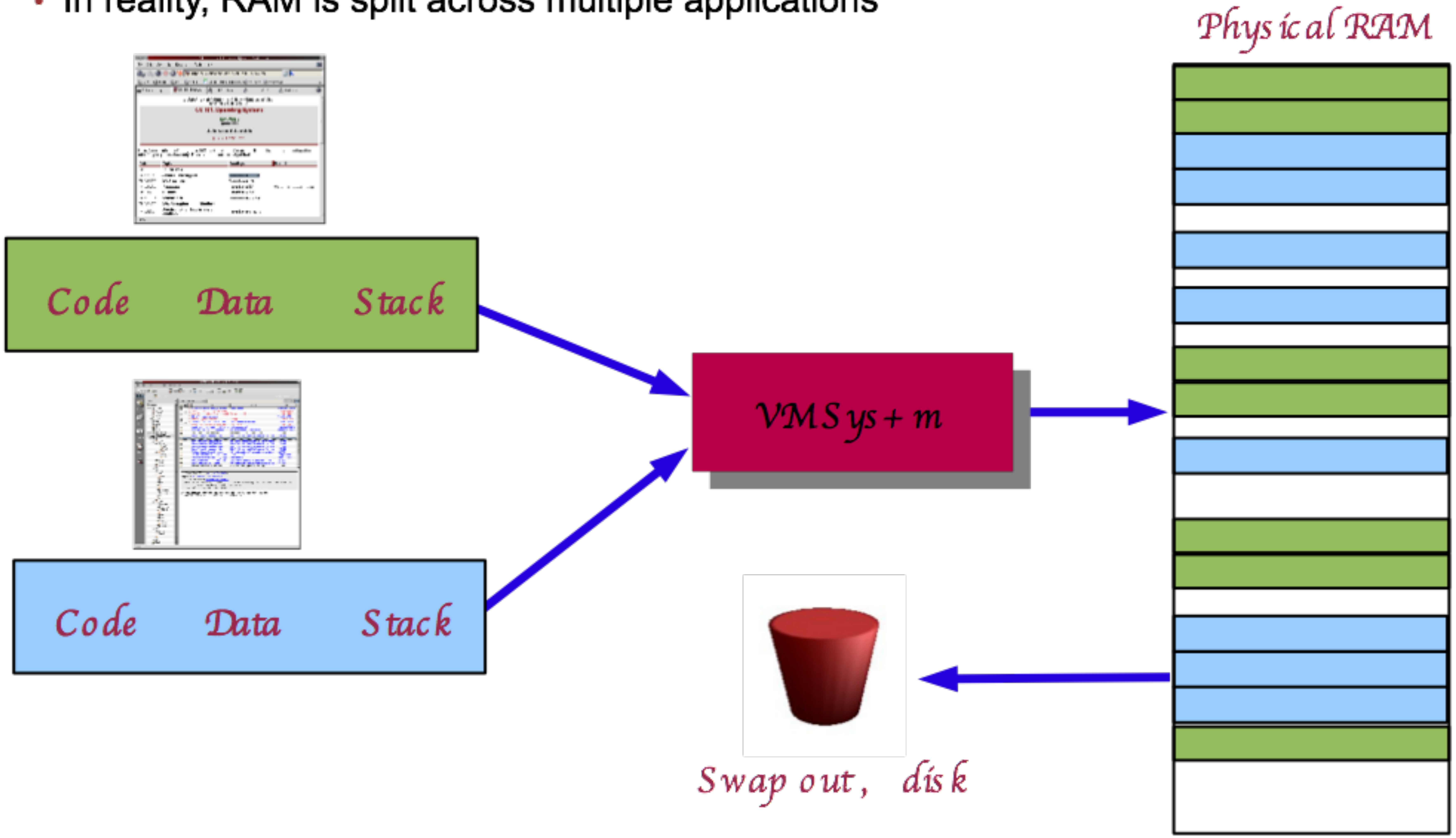
I think I have my own CPU

So do I

Another OS Function: Virtual Memory

Give every application the illusion of having infinite memory

- And, that it can access any memory address it likes!
- In reality, RAM is split across multiple applications



More OS Functions

Multiprocessor support

- Modern systems have multiple CPUs
- Can run multiple applications (or ***threads*** within applications) in parallel
- OS must ensure that memory and cache contents are consistent across CPUs

Filesystems

- Real disks have a hairy, sector-based access model
- User applications see flat files arranged in a hierarchical namespace

Network protocols

- Network interface hardware operates on the level of unreliable packets
- User apps see a (potentially reliable) byte-stream *socket*

Security and protection

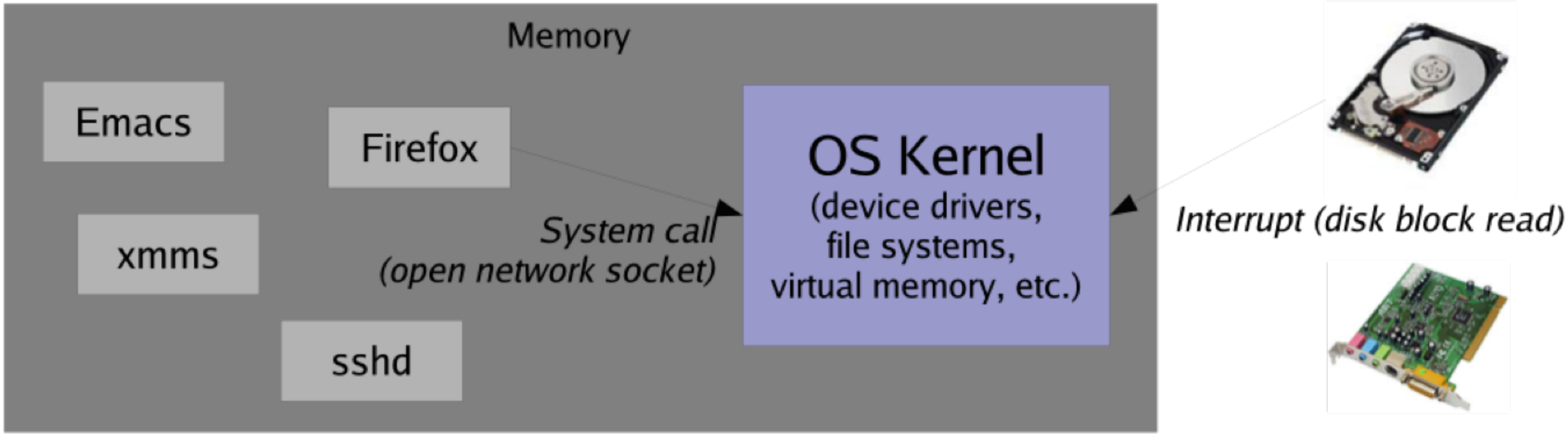
- Prevent multiple apps from interfering with each other and with normal system operation

Contents

- Interrupts and system calls
- User mode and kernel mode

Operating System basics

The OS kernel is just a bunch of code that sits around in memory, waiting to be executed



Memory

Emacs

Firefox

xmms

sshd

System call
(open network socket)

OS Kernel
(device drivers,
file systems,
virtual memory, etc.)

Interrupt (disk block read)

OS is triggered in two ways: *system calls* and *hardware interrupts*

System call: Direct “call” from a user program

- For example, open() to open a file, or exec() to run a new program

Hardware interrupt: Trigger from some hardware device

- For example, when a disk block has been read or written

Interrupts – a primer

An *interrupt* is a signal that causes the CPU to jump to a pre-defined instruction – called the *interrupt handler*

- Interrupt can be caused by hardware or software

Hardware interrupt examples

- Timer interrupt (periodic “tick” from a programmable timer)
- Device interrupts
  - *e.g., Disk will interrupt the CPU when an I/O operation has completed*

Software interrupt examples (also called *exceptions*)

- Division by zero error
- Access to a bad memory address
- Intentional software interrupt – e.g., x86 “INT” instruction
  - *Can be used to trap from user program into the OS kernel!*
  - *Why might this be useful?*

User mode vs. kernel mode

What makes the kernel different from user programs?

- Kernel can execute special *privileged instructions*

Examples of privileged instructions:

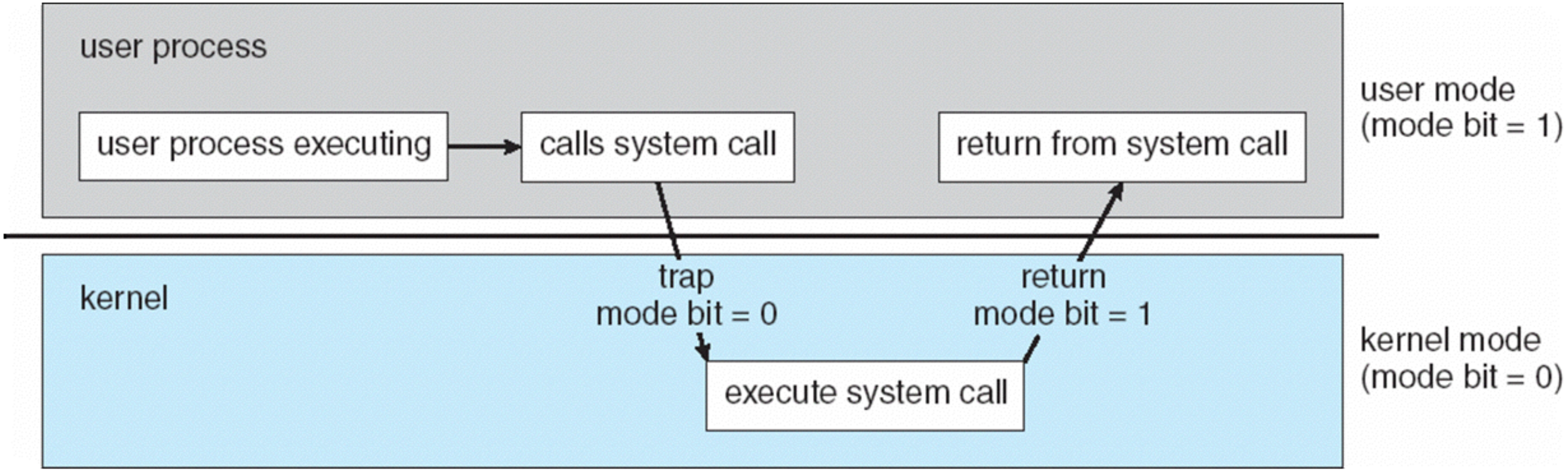
- Access I/O devices
  - *Poll for IO, perform DMA, catch hardware interrupt*
- Manipulate memory management
  - *Set up page tables, load/flush the TLB and CPU caches, etc.*
- Configure various “mode bits”
  - *Interrupt priority level, software trap vectors, etc.*
- Call halt instruction
  - *Put CPU into low-power or idle state until next interrupt*

These are enforced by the **CPU hardware itself**.

- CPU has at least two protection levels: *Kernel mode* and *user mode*
- CPU checks current protection level on each instruction
- What happens if user program tries to execute a privileged instruction?

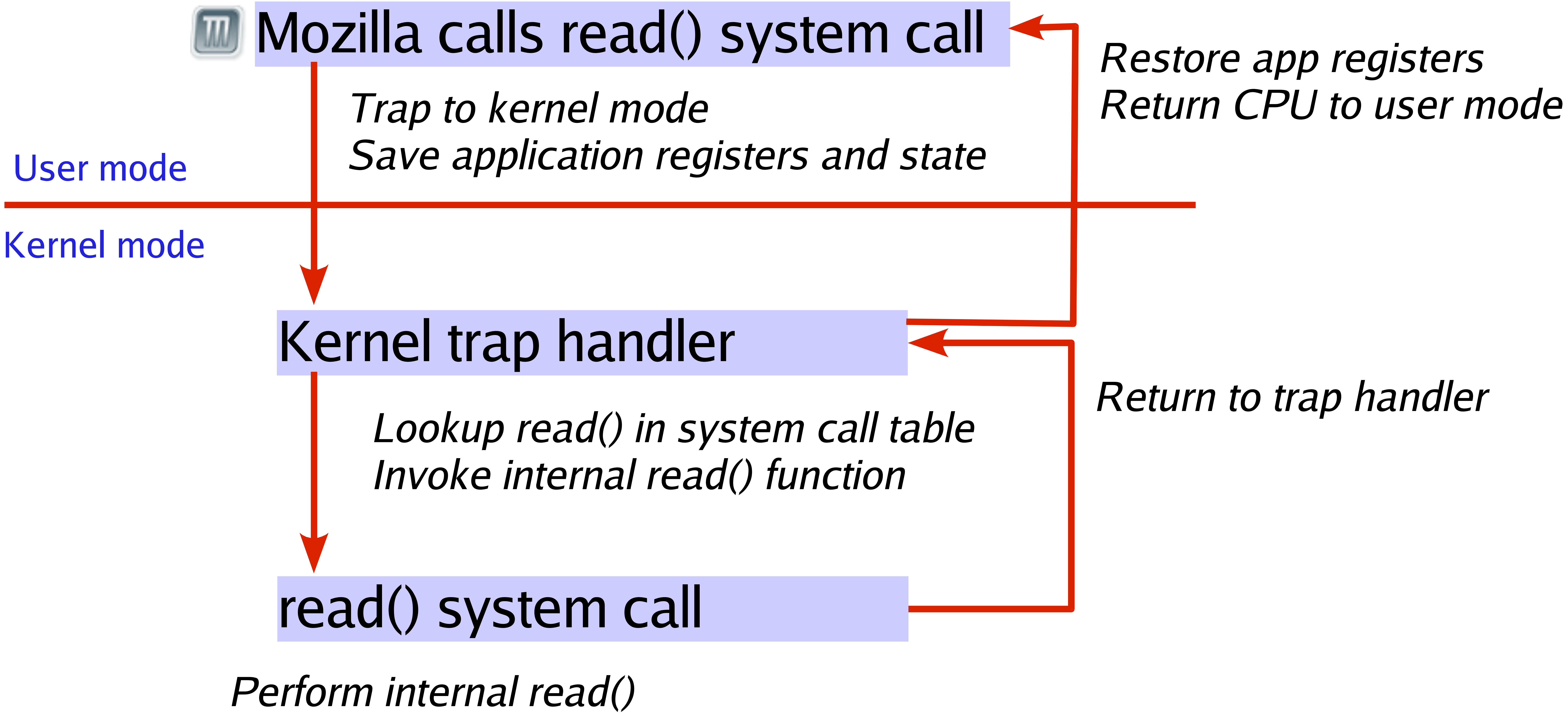
Adapted from Matt Welsh’s slides (Harvard University)

Transition from user to kernel mode



Silberschatz, Galvin and Gagne ©2009

Transition from user to kernel mode: example



Mozilla calls read() system call

Trap to kernel mode
Save application registers and state

User mode

Kernel mode

Kernel trap handler

Lookup read() in system call table
Invoke internal read() function

read() system call

Perform internal read()

Return to trap handler

Restore app registers
Return CPU to user mode

Adapted from Matt Welsh's slides (Harvard University)

Uniprogramming and Multiprogramming

Uniprogramming

- Only one program can run at a given time on the system
- Like old batch systems, MS-DOS, etc.

Multiprogramming (a.k.a. "multitasking")

- Multiple programs can run simultaneously
- Although only one program running **at any given instant**
  - *(Unless you have multiple CPUs!!!!)*

Tradeoffs

- Writing a uniprogramming OS is simpler
  - *Why?*
- But, multitasking OSs use resources more efficiently
  - *Why?*

Note on terminology:
Multitasking/multiprogramming refer to the number of programs running
Multiprocessing refers to the number of CPUs in the system

Adapted from Matt Welsh's slides (Harvard University

Process Management

An OS executes many kinds of applications

- Regular user programs
  - *Emacs, Mozilla, this OpenOffice program, etc...*
- Administrative servers
  - *Crond: Runs jobs at pre-scheduled times*
  - *Sshd: Manages incoming ssh connections*
  - *Lpd: Queues up jobs for the printer*

Each of these activities is encapsulated in a *process*

- A process consists of three main parts:

Processor state

- registers, program counter

OS resources

- open files, network sockets, etc.

Address space:

- The memory that a process accesses – its code, variables, stack, etc.

Adapted from Matt Welsh's slides (Harvard University)

Process Example

A **process** is an instance of a program being executed

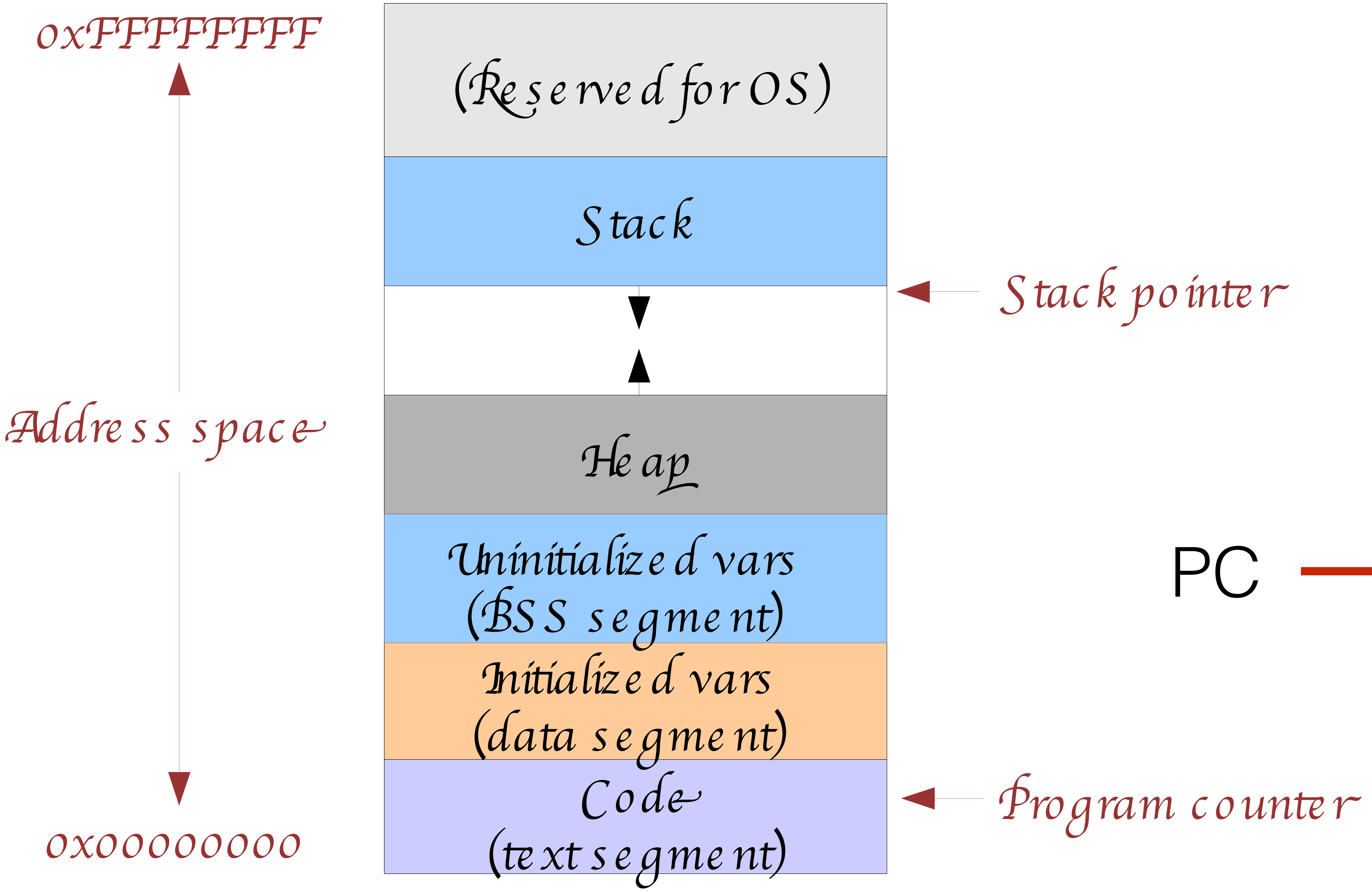
- Use “ps” to list processes on UNIX systems

```
 PID TTY      STAT   TIME COMMAND
 842 tty1     S      0:00 -bash
 867 tty1     S      0:00 xinit
 873 tty1     S      0:00 fvwm2
 887 tty1     S      0:00 xload
 888 tty1     S      0:02 /usr/local/j2sdk1.4.0/bin/java ApmView 896 243
1881 tty1     S      0:00 rxvt -fn fixed -cr red -fg white -bg #586570 -geometr
1883 pts/2    S      0:00 bash
1910 pts/0    S      0:00 /bin/sh /home/mdw/bin/ooffice arch.sxi
1911 pts/0    S      1:20 /usr/local/OpenOffice.org1.1.0/program/soffice.bin ar
1937 tty1     S      0:00 /bin/sh /home/mdw/bin/set-wlan-OFF
2310 pts/2    R      0:00 ps -Umdw -x
```

Adapted from Matt Welsh’s slides (Harvard University)

What is a process?

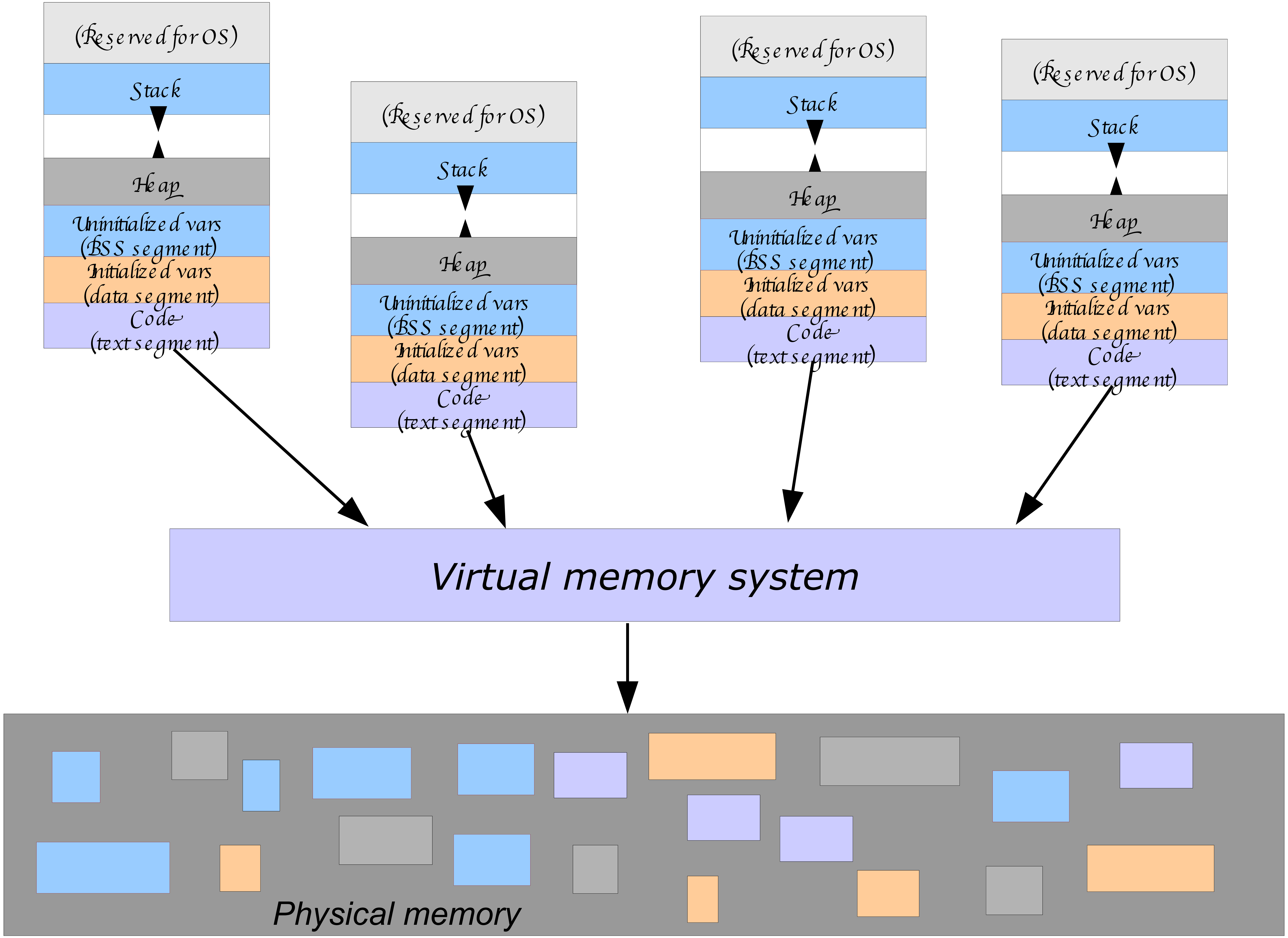
- A *process* is an abstraction of *a program in execution.*



PC →

```
/* Hello World program */

#include<stdio.h>

main()
{
    printf("Hello World");

}
```

Multiple processes



(Reserved for OS)
Stack
Heap
Uninitialized vars
(BSS segment)
Initialized vars
(data segment)
Code
(text segment)

(Reserved for OS)
Stack
Heap
Uninitialized vars
(BSS segment)
Initialized vars
(data segment)
Code
(text segment)

(Reserved for OS)
Stack
Heap
Uninitialized vars
(BSS segment)
Initialized vars
(data segment)
Code
(text segment)

(Reserved for OS)
Stack
Heap
Uninitialized vars
(BSS segment)
Initialized vars
(data segment)
Code
(text segment)

Virtual memory system

Physical memory

Figure by Matt Welsh, Harvard University.

Process Control Block (BCP)

The OS maintains a BCP for each process. It is a data structure with many fields.

Defined in:

`/include/linux/sched.h`

```
struct task_struct {
volatile long state;
unsigned long flags;
int sigpending;
mm_segment_t addr_limit;
struct exec_domain *exec_domain;
volatile long need_resched;
unsigned long ptrace;
int lock_depth;
unsigned int cpu;
int prio, static_prio;
struct list_head run_list;
prio_array_t *array;
unsigned long sleep_avg;
unsigned long last_run;
unsigned long policy;
unsigned long cpus_allowed;
unsigned int time_slice, first_time_slice;
atomic_t usage;
struct list_head tasks;
struct list_head ptrace_children;
struct list_head ptrace_list;
struct mm_struct *mm, *active_mm;
struct linux_binfmt *binfmt;
int exit_code, exit_signal;
int pdeath_signal;
unsigned long personality;
int did_exec:1;
unsigned task_dumpable:1;
pid_t pid;
pid_t pgrp;
pid_t tty_old_pgrp;
pid_t session;
pid_t tgid;
int leader;
struct task_struct *real_parent;
struct task_struct *parent;
struct list_head children;
struct list_head sibling;
struct task_struct *group_leader;
struct pid_link pids[PIDTYPE_MAX];
wait_queue_head_t wait_chldexit;
struct completion *vfork_done;
int *set_child_tid;
int *clear_child_tid;
unsigned long rt_priority;
unsigned long it_real_value, it_prof_value, it_virt_value;
unsigned long it_real_incr, it_prof_incr, it_virt_incr;
struct timer_list real_timer;
struct tms times;
struct tms group_times;
unsigned long start_time;
long per_cpu_utime[NR_CPUS], per_cpu_stime[NR_CPUS];
unsigned long min_flt, maj_flt, nswap, cmin_flt, cmaj_flt,
cnswap;
int swappable:1;
uid_t uid,euid,suid,fsuid;
gid_t gid,egid,sgid,fsgid;
int ngroups;
gid_t groups[NGROUPS];
kernel_cap_t   cap_effective, cap_inheritable, cap_permitted;
int keep_capabilities:1;
struct user_struct *user;
struct rlimit rlim[RLIM_NLIMITS];
unsigned short used_math;
char comm[16];
int link_count, total_link_count;
struct tty_struct *tty;
unsigned int locks;
struct sem_undo *semundo;
struct sem_queue *semsleeping;
struct thread_struct thread;
struct fs_struct *fs;
struct files_struct *files;
struct namespace *namespace;
struct signal_struct *signal;
struct sighand_struct *sighand;
sigset_t blocked, real_blocked;
struct sigpending pending;
unsigned long sas_ss_sp;
size_t sas_ss_size;
int (*notifier)(void *priv);
void *notifier_data;
sigset_t *notifier_mask;
void *tux_info;
void (*tux_exit)(void);
      u32 parent_exec_id;
      u32 self_exec_id;
spinlock_t alloc_lock;
        spinlock_t switch_lock;
void *journal_info;
unsigned long ptrace_message;
siginfo_t *last_siginfo;
};
```

Figure by Matt Welsh, Harvard University.

CPU Virtualization

```c
////////////////////////////////////////
// compilation:
//    gcc -Wall cpu.c -o cpu
////////////////////////////////////////

#include <stdio.h>
#include <stdlib.h>
#include "common.h"

int main(int argc, char *argv[])
{
    if (argc != 2) {
      fprintf(stderr, "usage: cpu <string>\n");
      exit(1);
    }
    char *str = argv[1];

    while (1) {
      printf("%s\n", str);
      Spin(1);
    }
    return 0;
}
```

```
prompt> ./cpu "A"
A
A
A
A
^C
prompt>
```

CPU Virtualization

```
prompt> ./cpu A & ; ./cpu B & ; ./cpu C & ; ./cpu D &
[1] 7353
[2] 7354
[3] 7355
[4] 7356
A
B
D
C
A
B
D
C
A
C
B
D
...
```

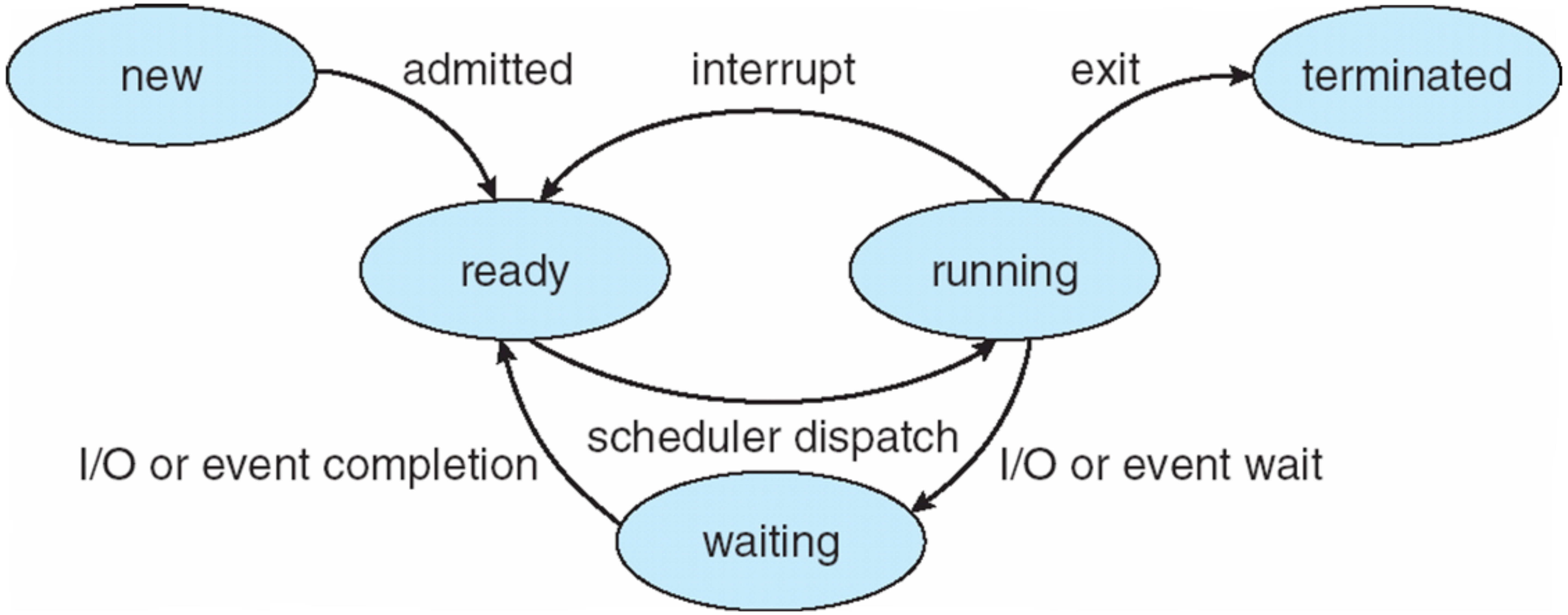
Memory Virtualization

```c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include "common.h"

int
main(int argc, char *argv[])
{
    if (argc != 2) {
      fprintf(stderr, "usage: mem <value>\n");
      exit(1);
    }
    int *p;                   // memory for pointer is on "stack"
    p = malloc(sizeof(int));  // malloc'd memory is on "heap"
    assert(p != NULL);
    // printf("(pid:%d) addr of main:     %llx\n", (int) getpid(), (unsigned long long) main);
    printf("(pid:%d) addr of p:        %llx\n", (int) getpid(), (unsigned long long) &p);
    printf("(pid:%d) addr stored in p: %llx\n", (int) getpid(), (unsigned long long) p);
    *p = atoi(argv[1]); // assign value to addr stored in p
    while (1) {
      Spin(1);
      *p = *p + 1;
      printf("(pid:%d) value of p: %d\n", getpid(), *p);
    }

    return 0;
}
```

Life cycle of a process

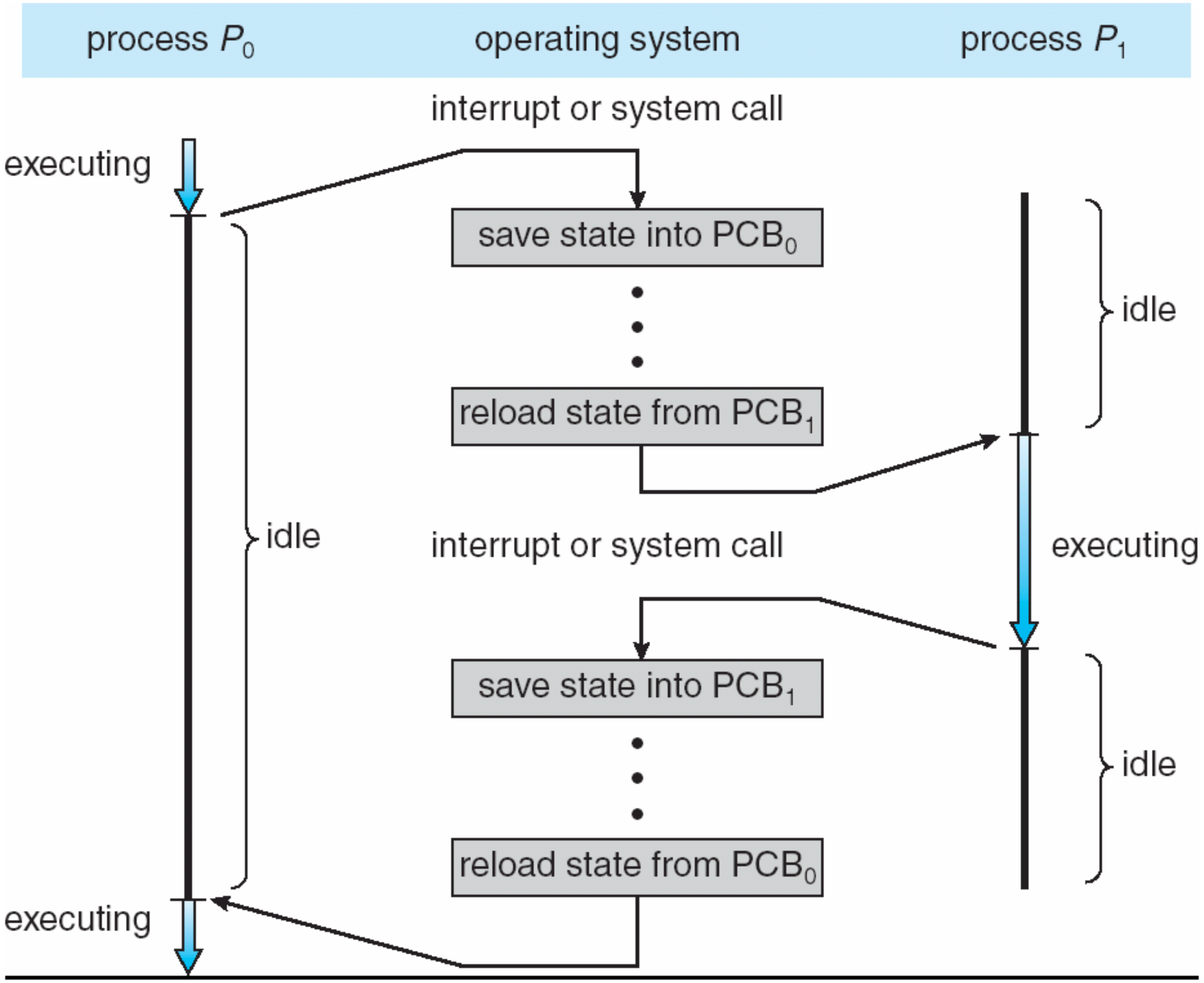
States of a process:

- **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



Adapted from Silberschatz, Galvin, and Gagne, 2009

CPU switch from process to process



process P_0

operating system

process P_1

interrupt or system call

executing

save state into PCB$_0$

idle

reload state from PCB$_1$

idle

interrupt or system call

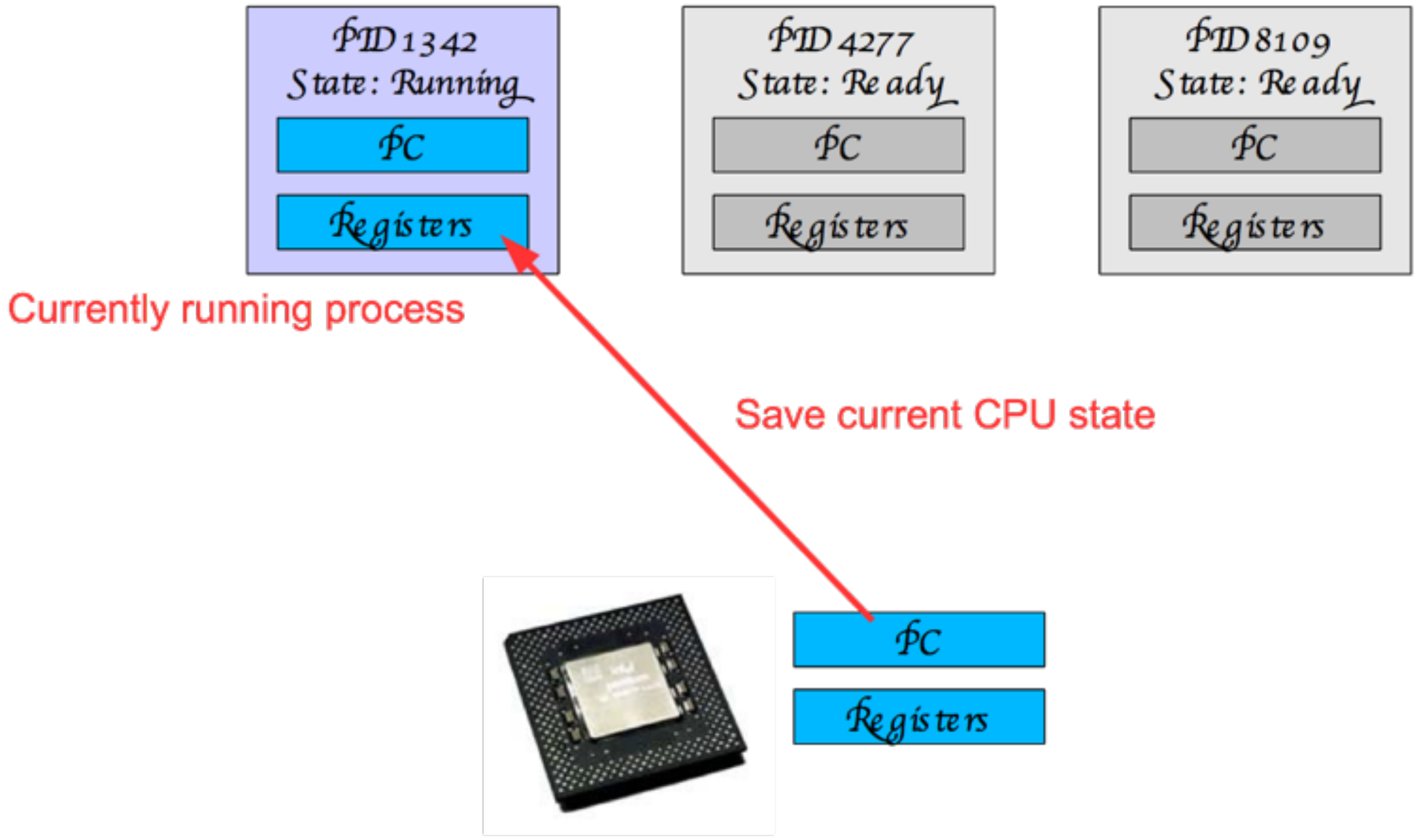
executing

save state into PCB$_1$

idle

reload state from PCB$_0$

executing



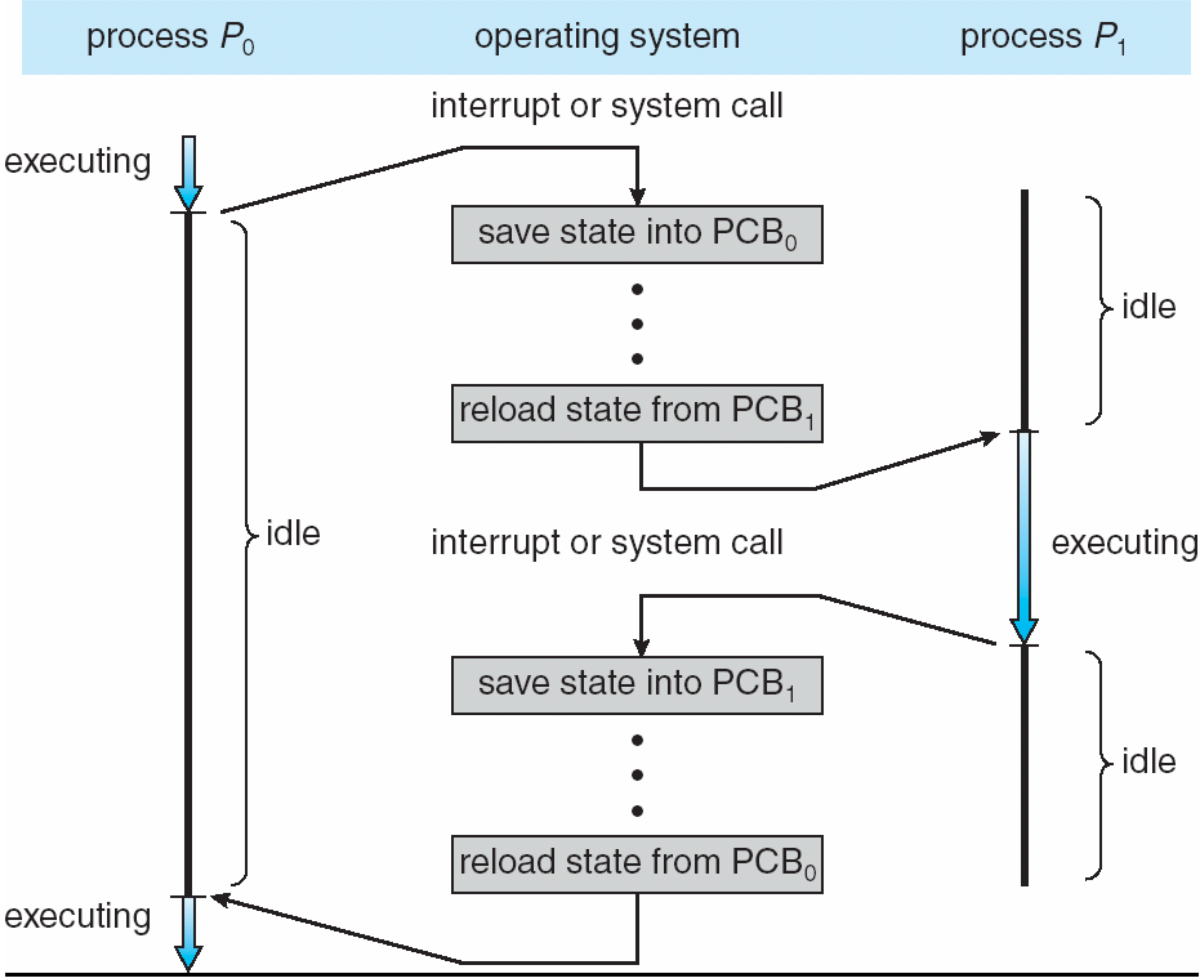
Currently running process

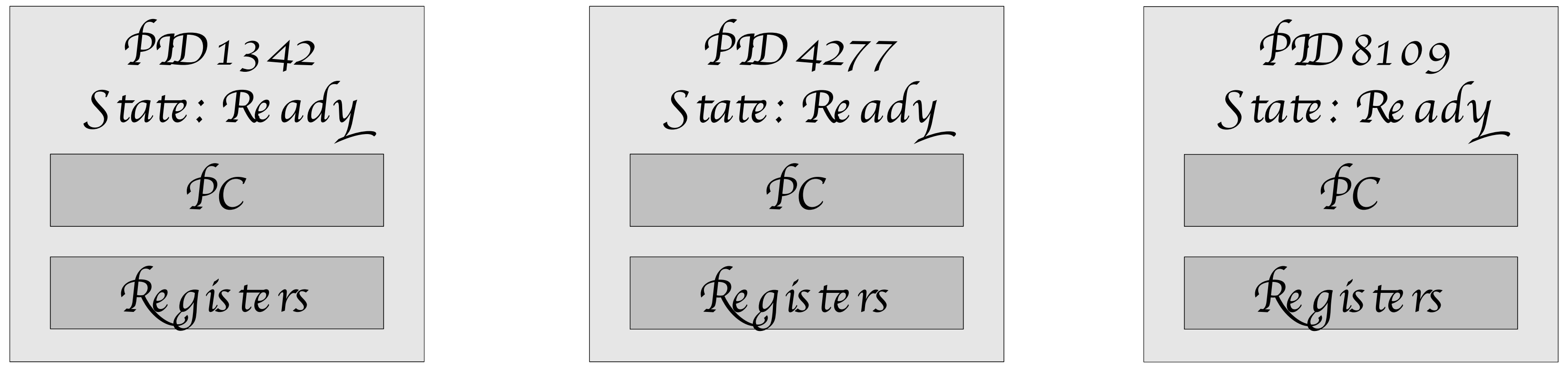
Save current CPU state

Adapted from Silberschatz, Galvin, and Gagne, 2009.

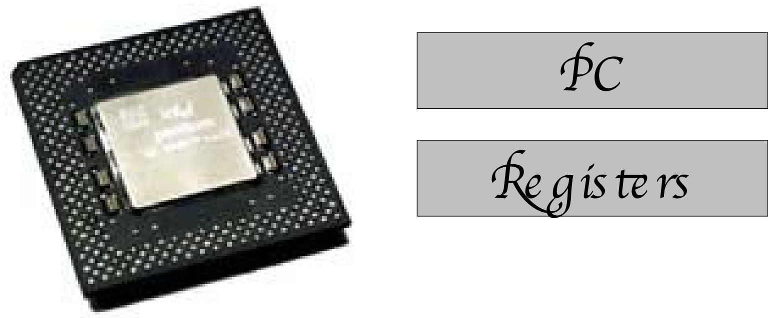
Figure by Matt Welsh, Harvard University.

CPU switch from process to process


process P_0
operating system
process P_1
interrupt or system call
executing
save state into PCB_0
reload state from PCB_1
idle
idle
executing
interrupt or system call
save state into PCB_1
reload state from PCB_0
idle
executing



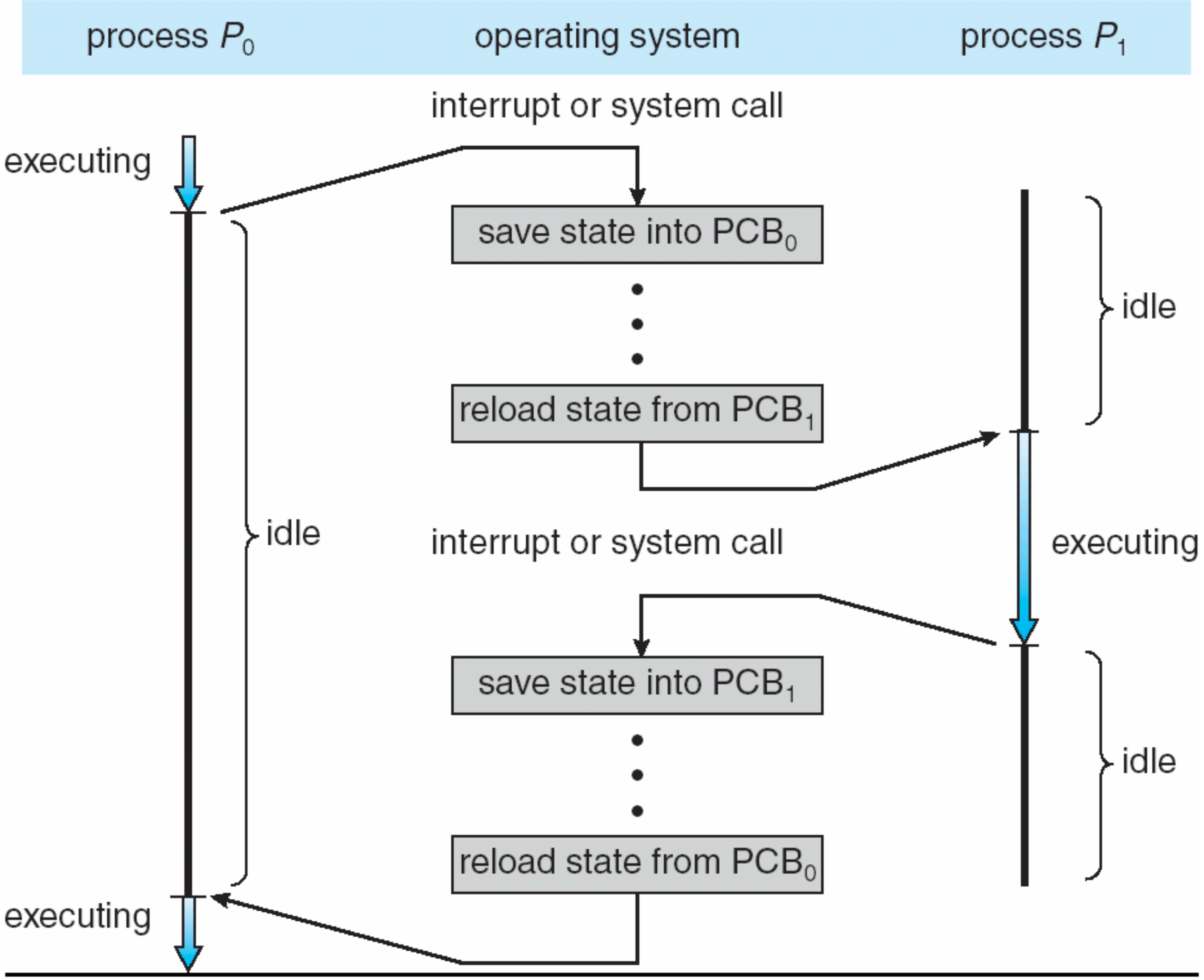
Suspend process



Adapted from Silberschatz, Galvin, and Gagne, 2009.

Figure by Matt Welsh, Harvard University.

CPU switch from process to process



process P_0

operating system

process P_1

interrupt or system call

executing

save state into PCB_0

idle

reload state from PCB_1

idle

executing

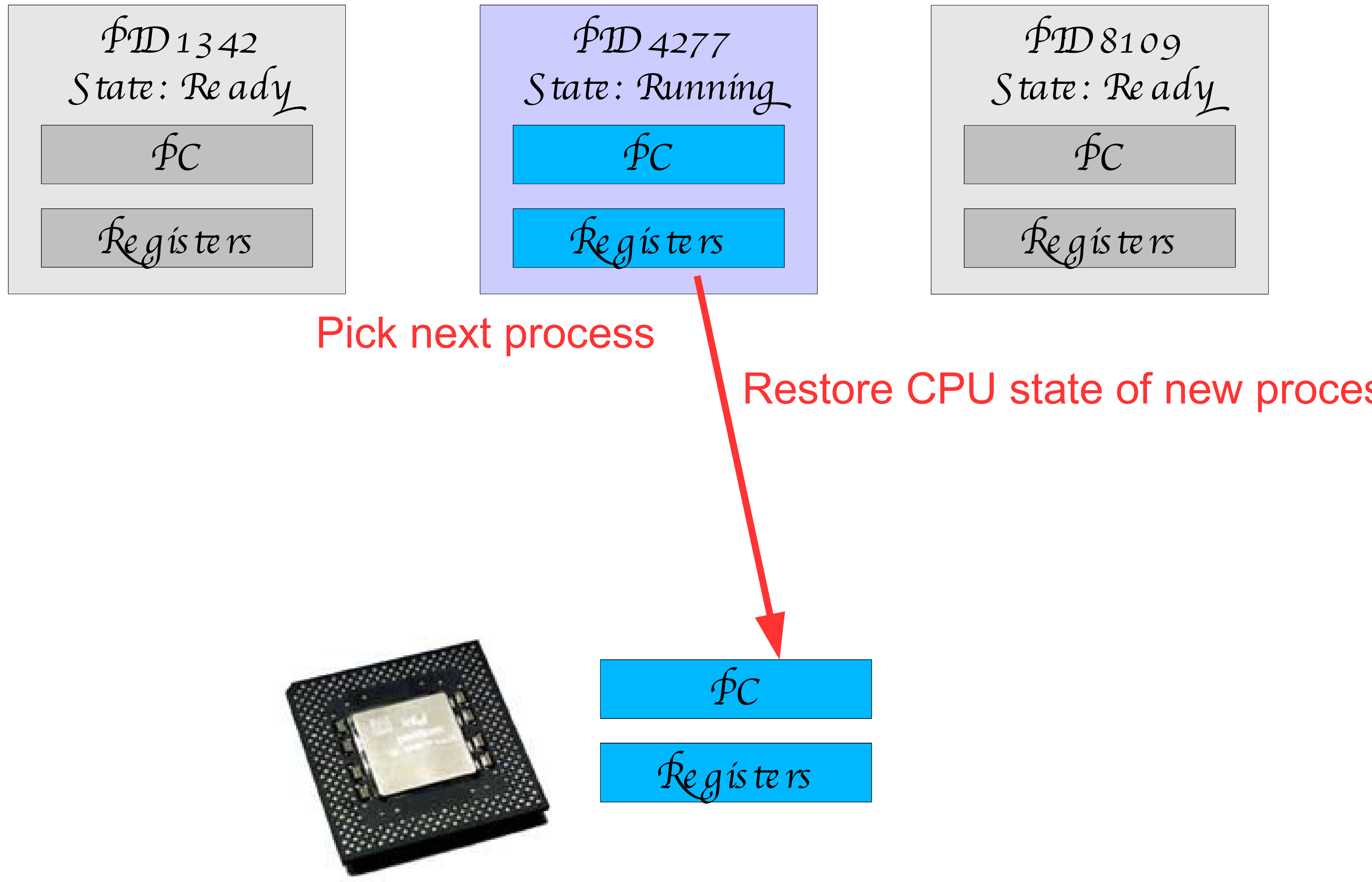
interrupt or system call

save state into PCB_1

idle

reload state from PCB_0

executing



PID 1342
State: Ready
PC
Registers

PID 4277
State: Running
PC
Registers

PID 8109
State: Ready
PC
Registers

Pick next process

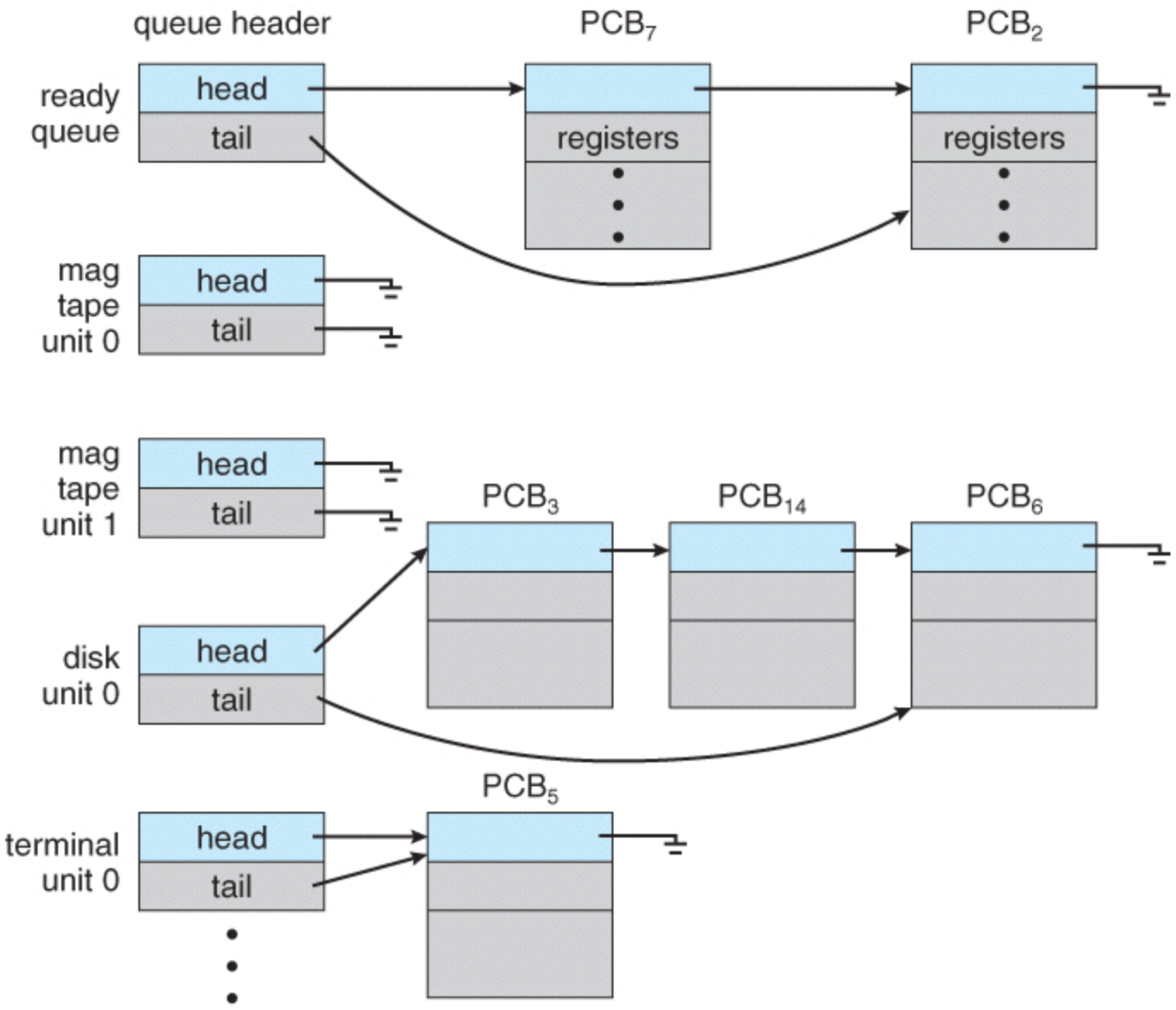
Restore CPU state of new proces

PC
Registers

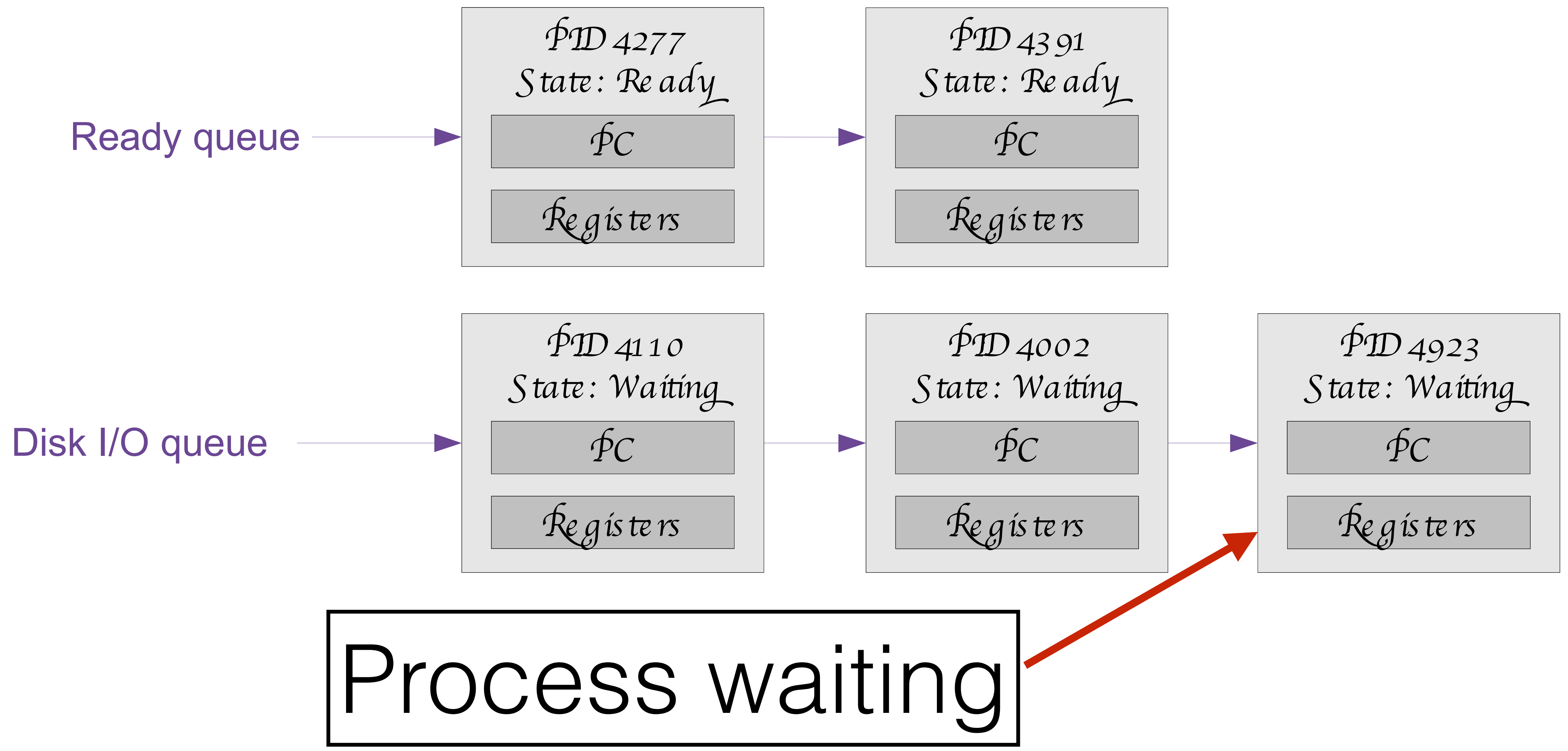
Adapted from Silberschatz, Galvin, and Gagne, 2009.

Figure by Matt Welsh, Harvard University.

Ready queue and various I/O queues

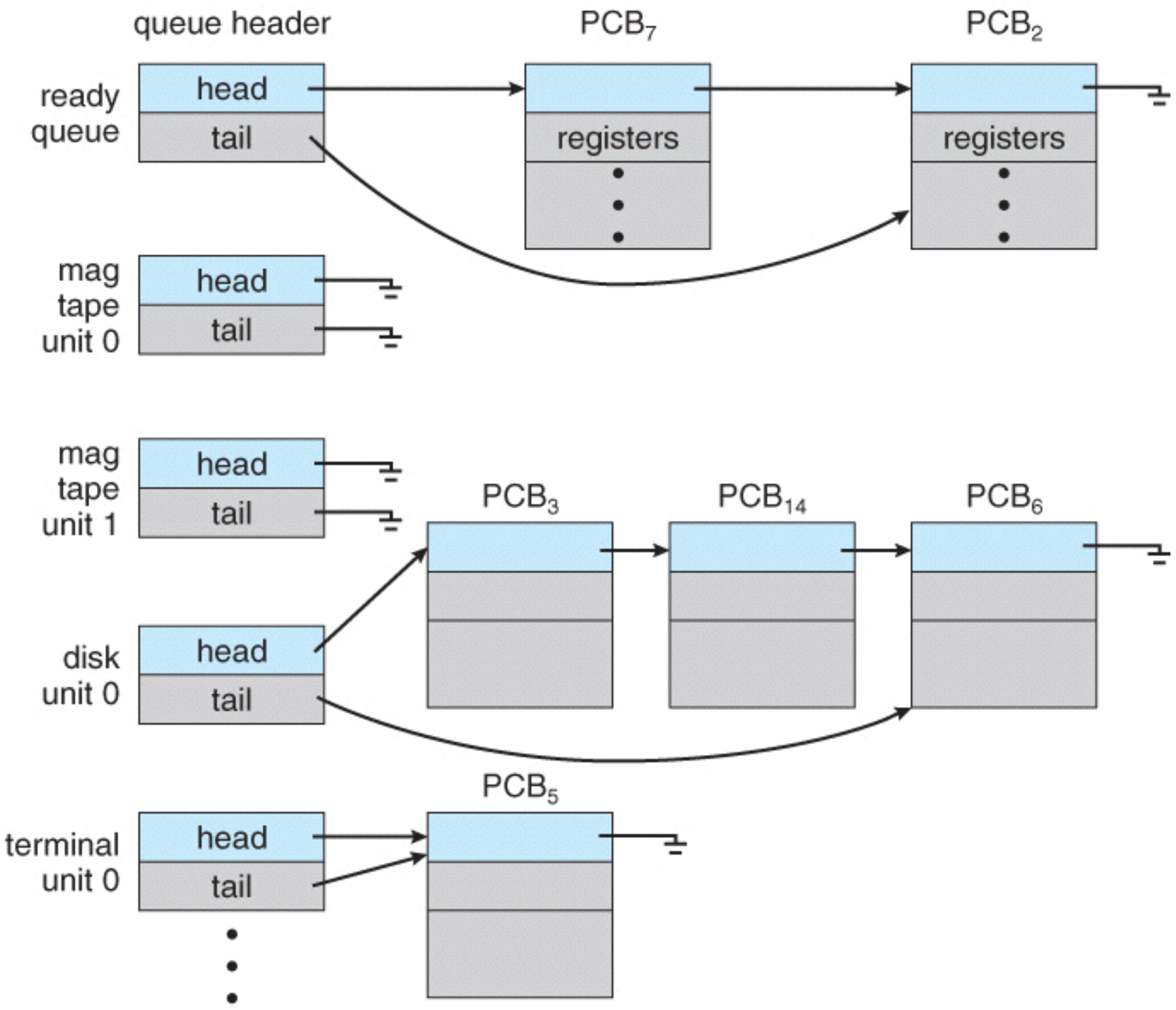


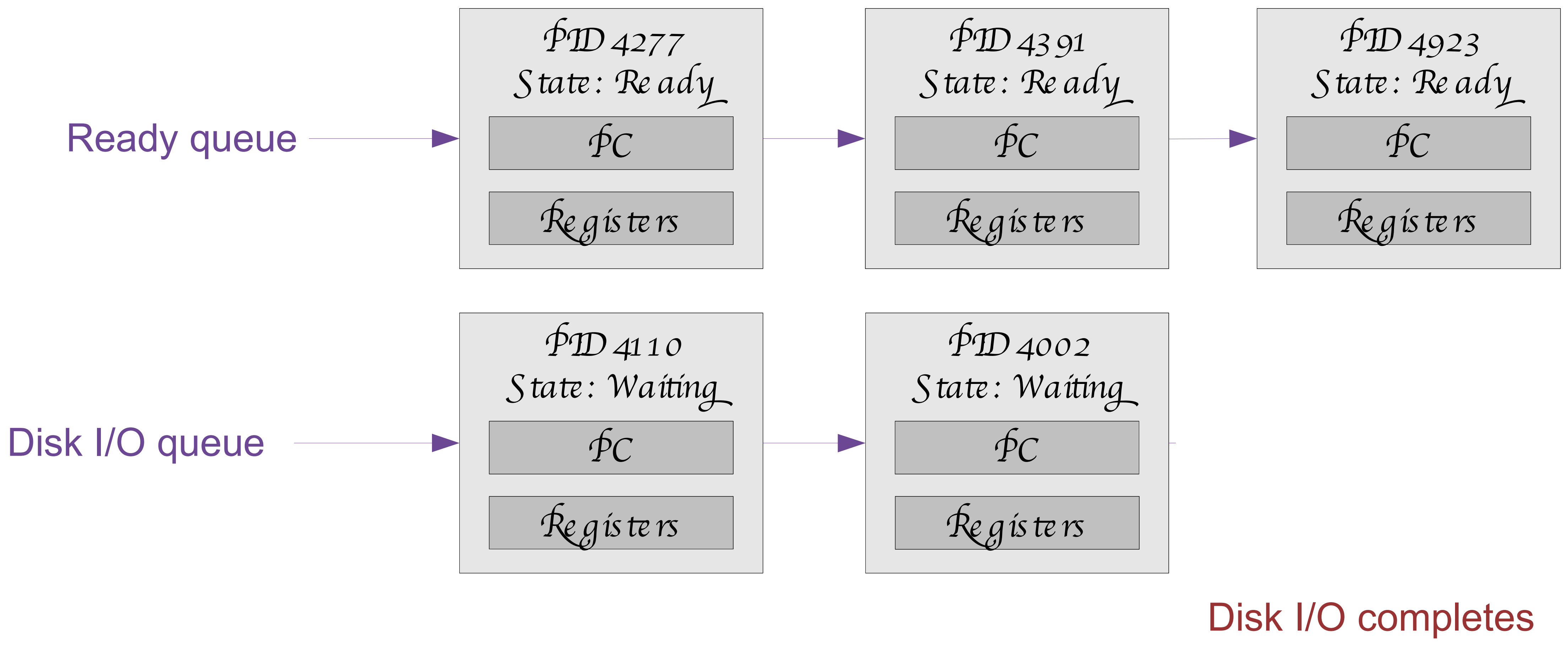
Adapted from Silberschatz, Galvin, and Gagne, 2009.



- OS maintains a set of queues
- Each PCB is queued on a state queue based on the process' current state.
- As processes change states, PCBs are unlinked from one queue and linked into another.

Ready queue and various I/O queues





- OS maintains a set of queues
- Each PCB is queued on a state queue based on the process' current state.
- As processes change states, PCBs are unlinked from one queue and linked into another.

Adapted from Silberschatz, Galvin, and Gagne, 2009.