CSE4001: Operating Systems Concepts

Processes and limited direct execution

What is a process?

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

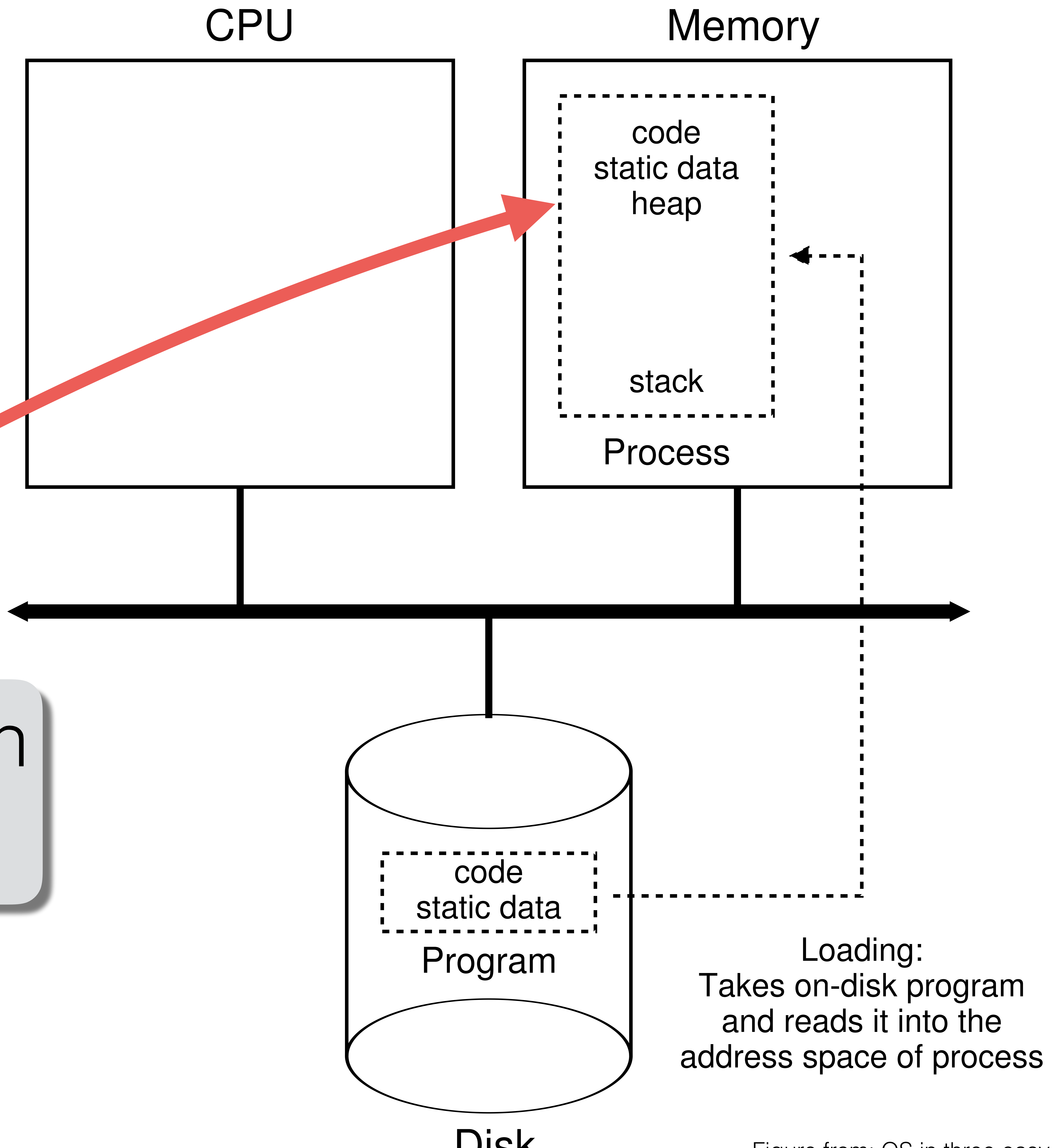


Figure from: OS in three easy pieces

What is a process?

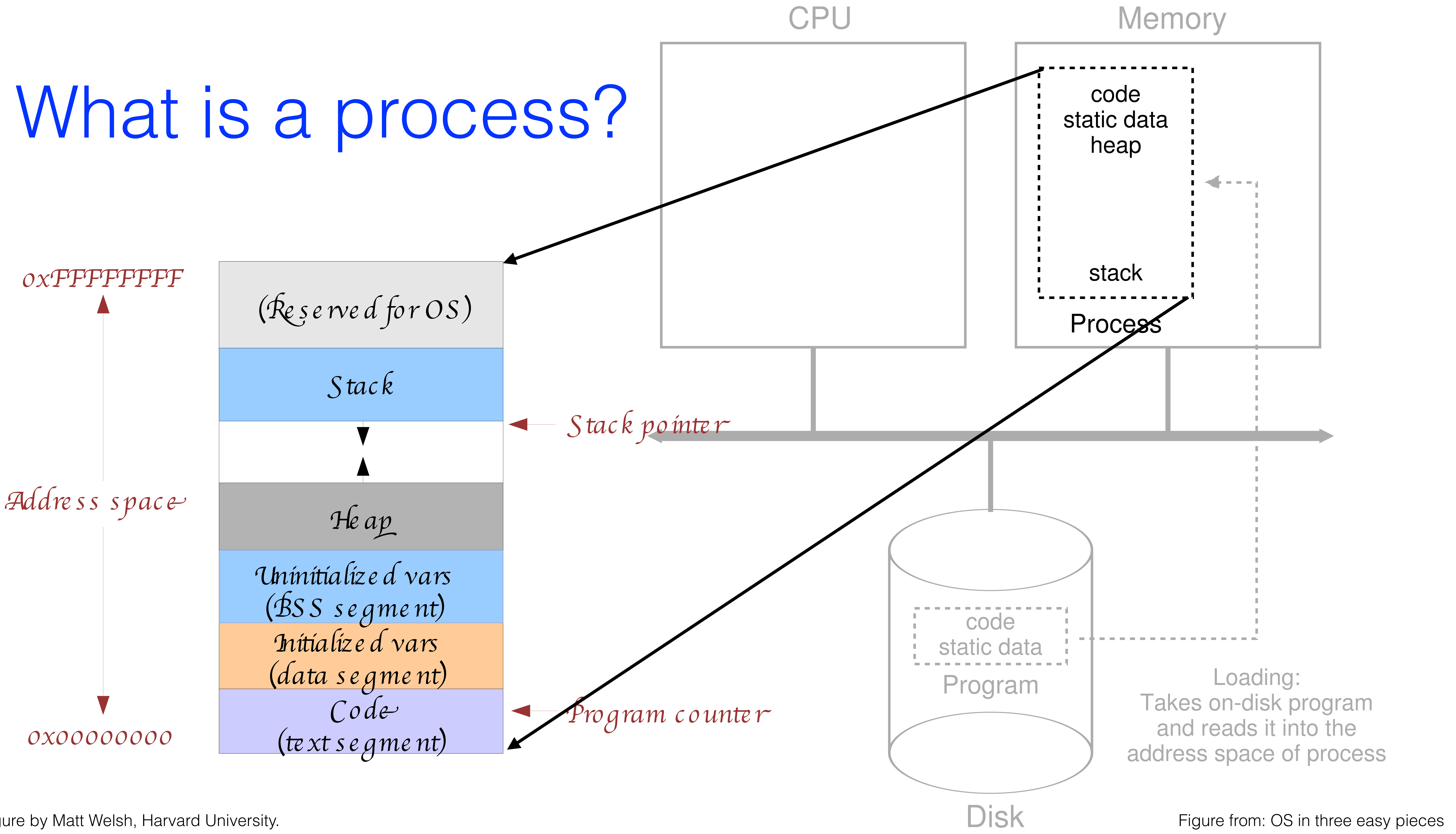
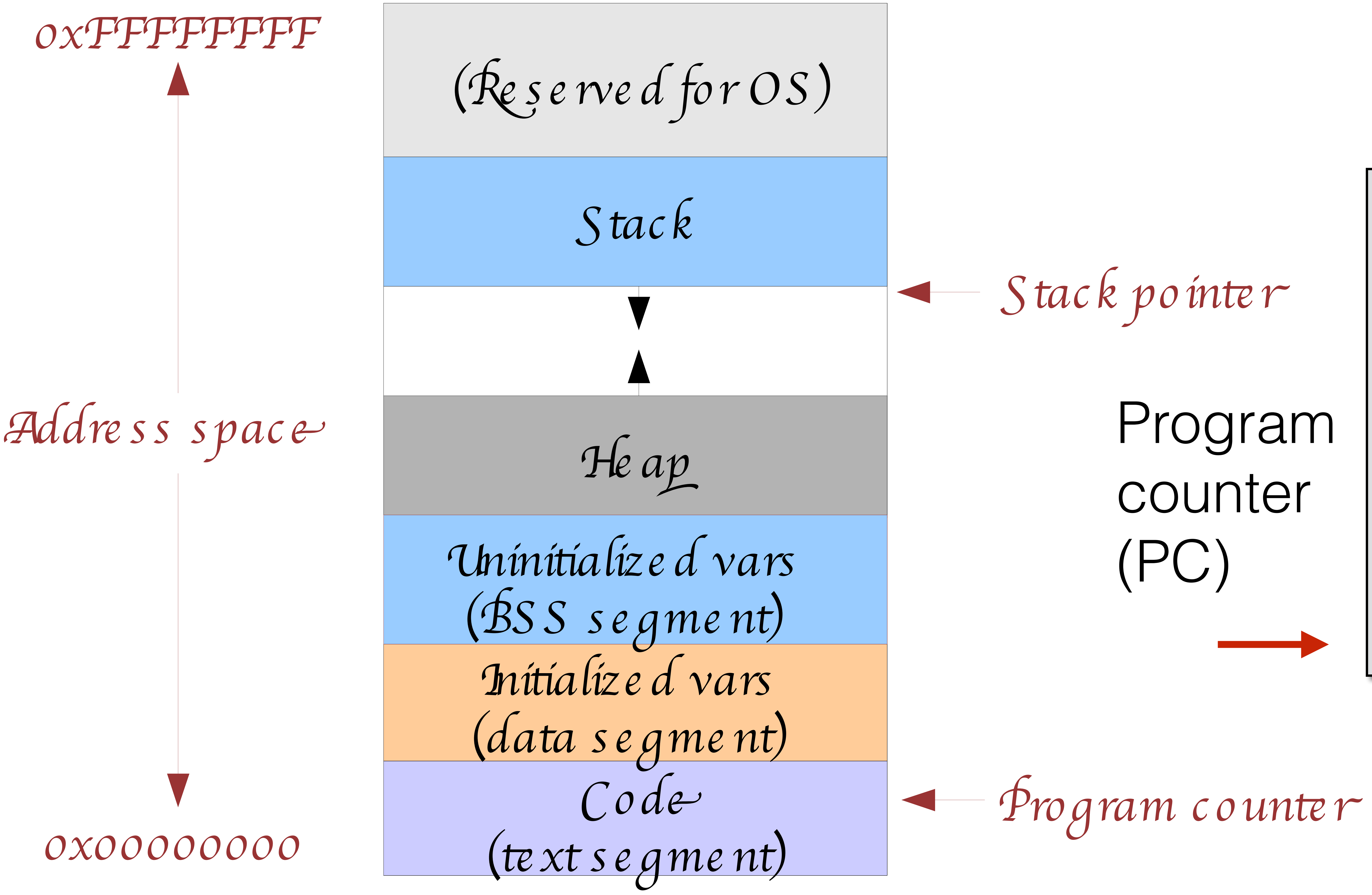


Figure by Matt Welsh, Harvard University.

Figure from: OS in three easy pieces

What is a process?




```
/* Hello World program */

#include<stdio.h>

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

}
```

Figure by Matt Welsh, Harvard University.

The OS view of a process

- ➔ Process state (ready, running, blocked, ...)
- ➔ The **address space** (how many possible addresses)
- ➔ The **code** of the running program
- ➔ The **data** of the running program
- ➔ An execution **stack** encapsulating the state of procedure calls
- ➔ The **program counter (PC)** indicating the address of the next instruction.
- ➔ A set of general-purpose **registers** with current values
- ➔ A set of operating system **resources**
  - ◆ open files, network connections, signals, etc.
- ➔ CPU scheduling info: process **priority**
- ➔ Each process is identified by its **process ID (PID)**

All these information is stored in a construct called
Process Control Block (PCB)

Adapted from original slide by Larry Zhang

Process Control Block (PCB)

The OS maintains a PCB 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;        Execution state
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;   Memory mgmt info
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;   Process ID
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;   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;                                Accounting info
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;   User ID
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;   CPU state
struct fs_struct *fs;
struct files_struct *files;   Open 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.

Example of simple PCB: 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;
};
```

[CK+08] "The xv6 Operating System"
Russ Cox, Frans Kaashoek, Robert Morris, Nickolai Zeldovich
From: http://pdos.csail.mit.edu/6.828/2008/index.html
The coolest real and little OS in the world. Download and play with it to learn more about the details of how operating systems actually work.

Figure from: OS in three easy pieces

Example of simple PCB: The xv6 `proc` structure

```
// the different states a process can be in
enum proc_state { UNUSED, EMBRYO, SLEEPING,
                  RUNNABLE, RUNNING, ZOMBIE };

// 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
};
```

Figure from: OS in three easy pieces

PCB in OS161

kern/src/kern/include/thread.h

```
/* States a thread can be in. */
typedef enum {
        S_RUN,          /* running */
        S_READY,        /* ready to run */
        S_SLEEP,        /* sleeping */
        S_ZOMBIE,       /* zombie; exited but not yet deleted */
} threadstate_t;

/* Thread structure. */
struct thread {
        /*
         * These go up front so they're easy to get to even if the
         * debugger is messed up.
         */
        char *t_name;                   /* Name of this thread */
        const char *t_wchan_name;       /* Name of wait channel, if sleeping */
        threadstate_t t_state;          /* State this thread is in */

        /*
         * Thread subsystem internal fields.
         */
        struct thread_machdep t_machdep; /* Any machine-dependent goo */
        struct threadlistnode t_listnode; /* Link for run/sleep/zombie lists */
        void *t_stack;                  /* Kernel-level stack */
        struct switchframe *t_context;  /* Saved register context (on stack) */
        struct cpu *t_cpu;              /* CPU thread runs on */
```

Running multiple processes concurrently

- Thousands of processes may be running
- Users do not need to worry about CPU availability
- Modern operating systems use a technique called time sharing.

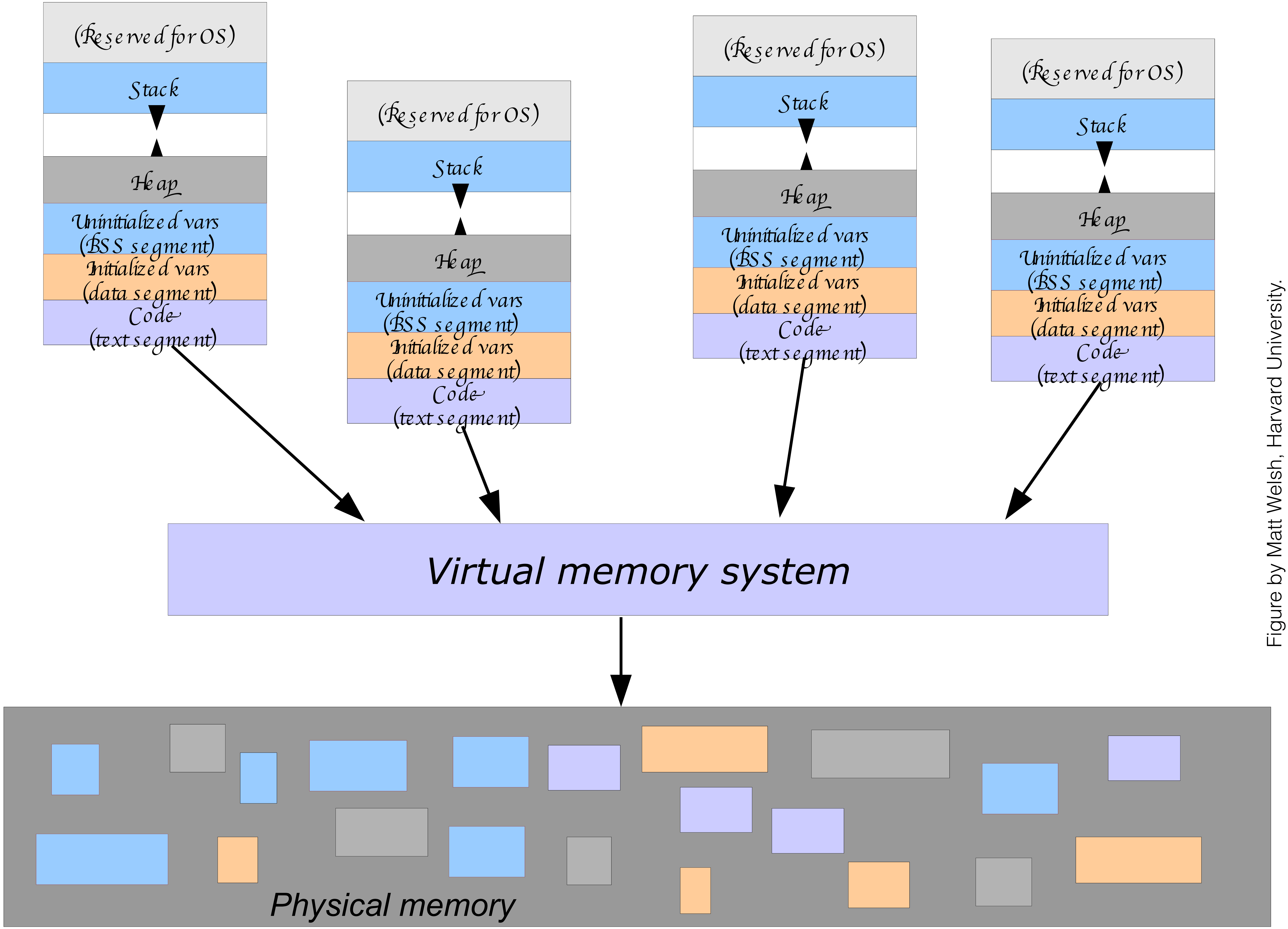


Figure by Matt Welsh, Harvard University.

Process Example

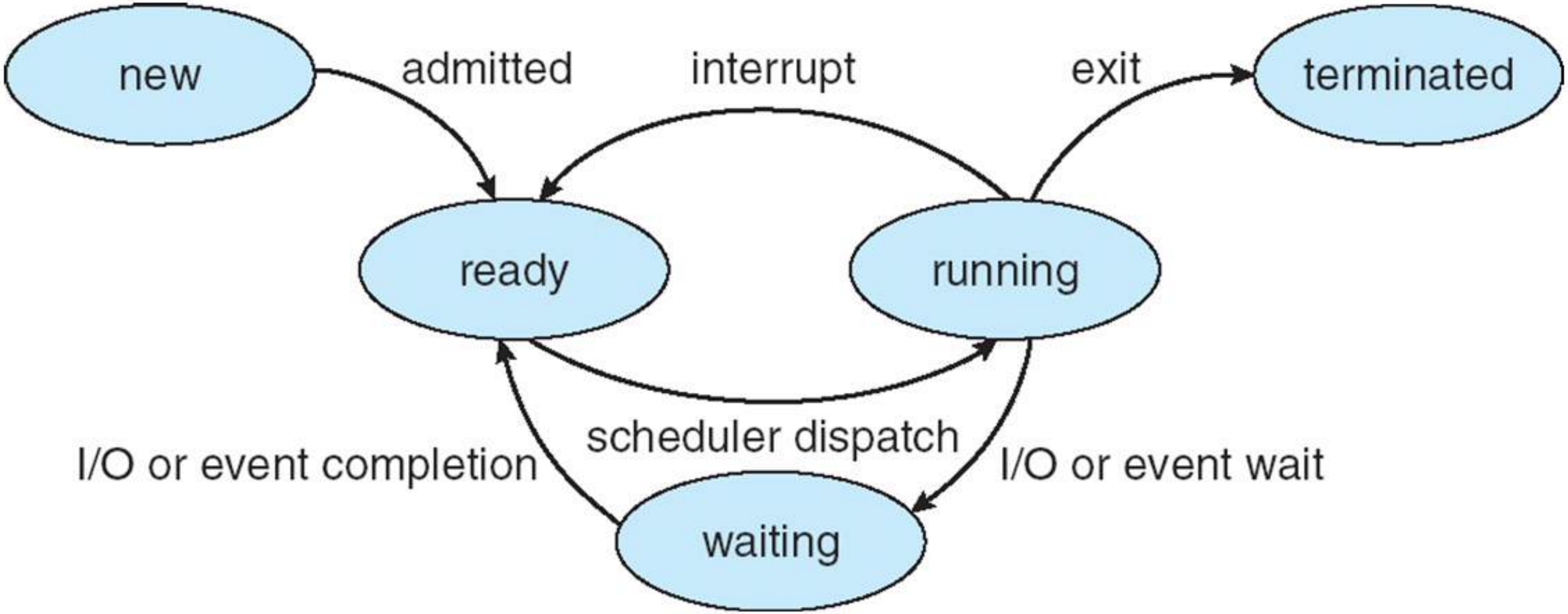
A **process** is an instance of a program being executed

- 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)

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.

Life cycle of a process

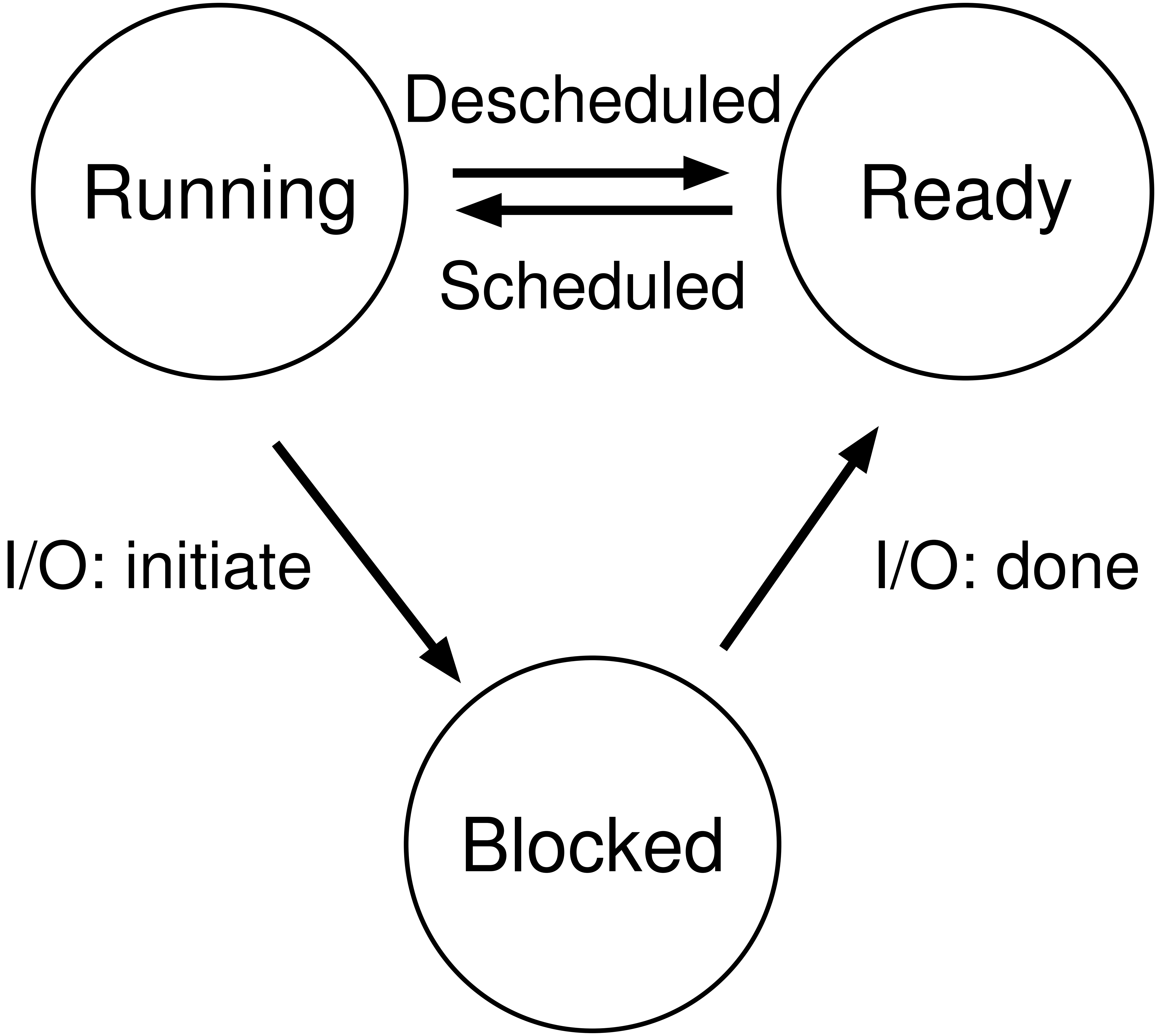
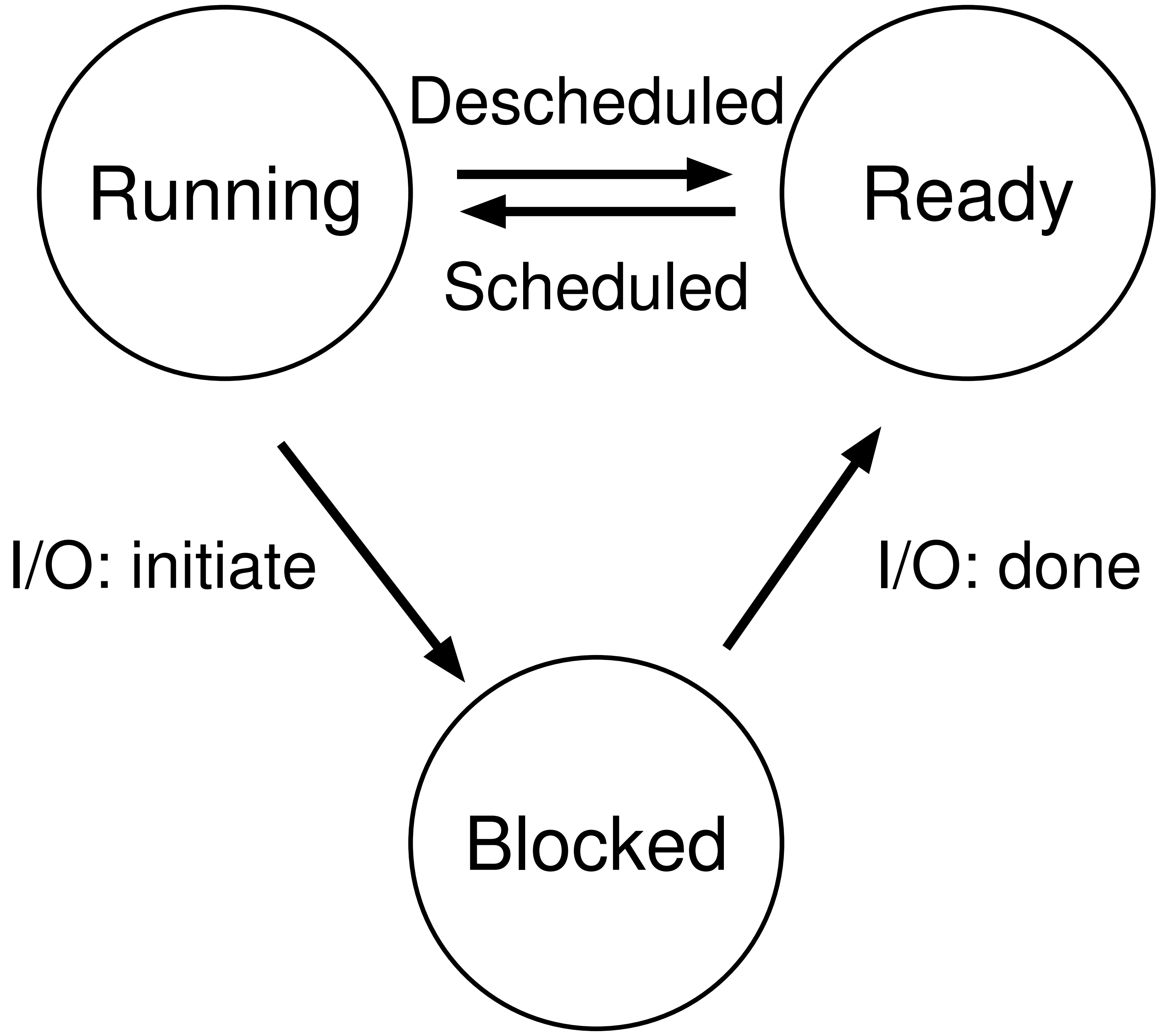


Figure from: OS in three easy pieces

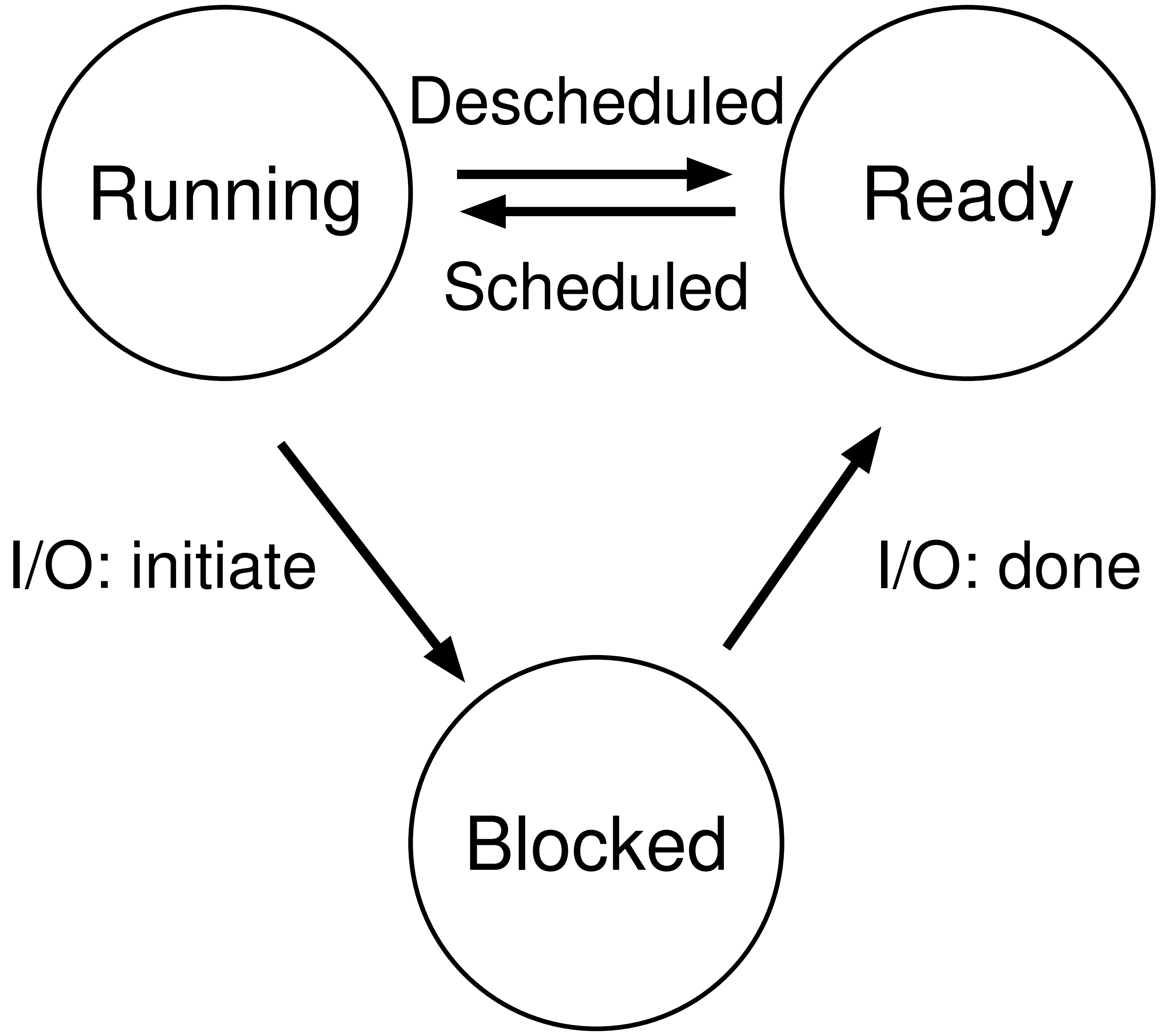
Example: two running processes, no I/O



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

Figure from: OS in three easy pieces

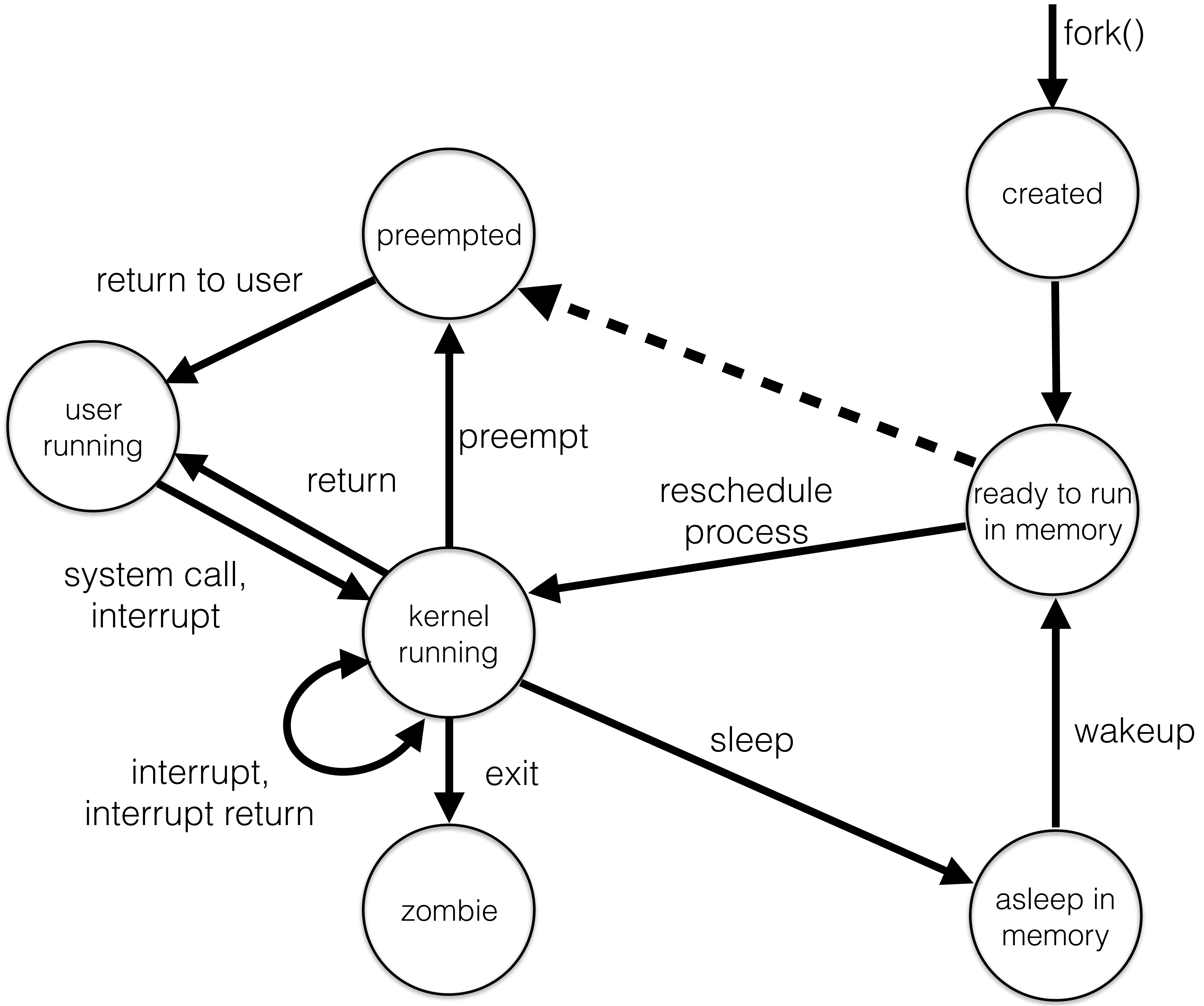
Example: two running processes, with I/O



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

Figure from: OS in three easy pieces

Process States (Unix)



Created: Process is newly created but it is not ready to run yet.

Preempted: Process is returning from kernel to user mode, but the kernel preempts it and does a process switch to schedule another process.

Zombie: Process is no longer exists, but it leaves a record for its parent process to collect.

Figure adapted from Stallings' book

Process States (Unix)

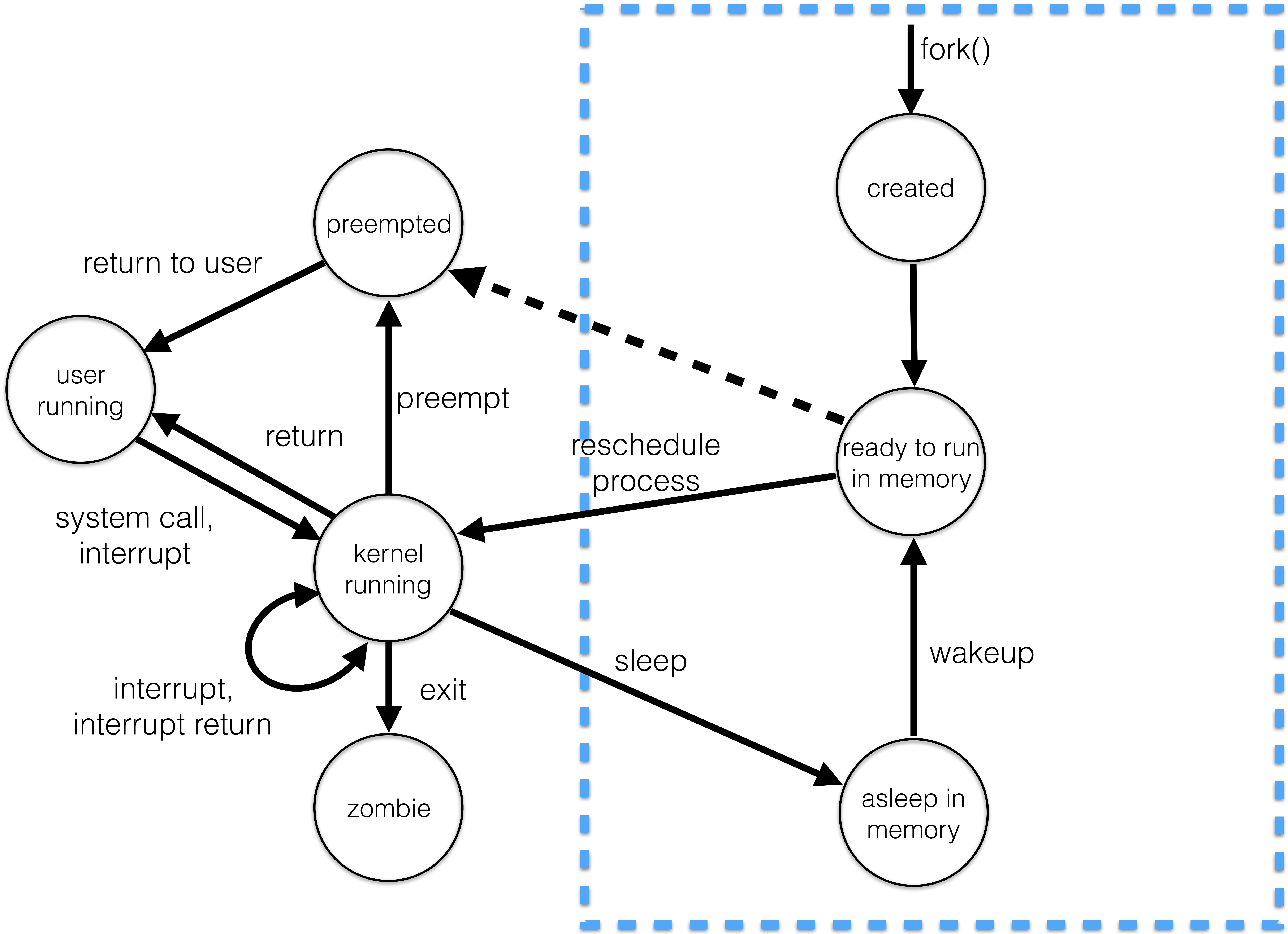


Figure adapted from Stallings' book

Process States (Unix)

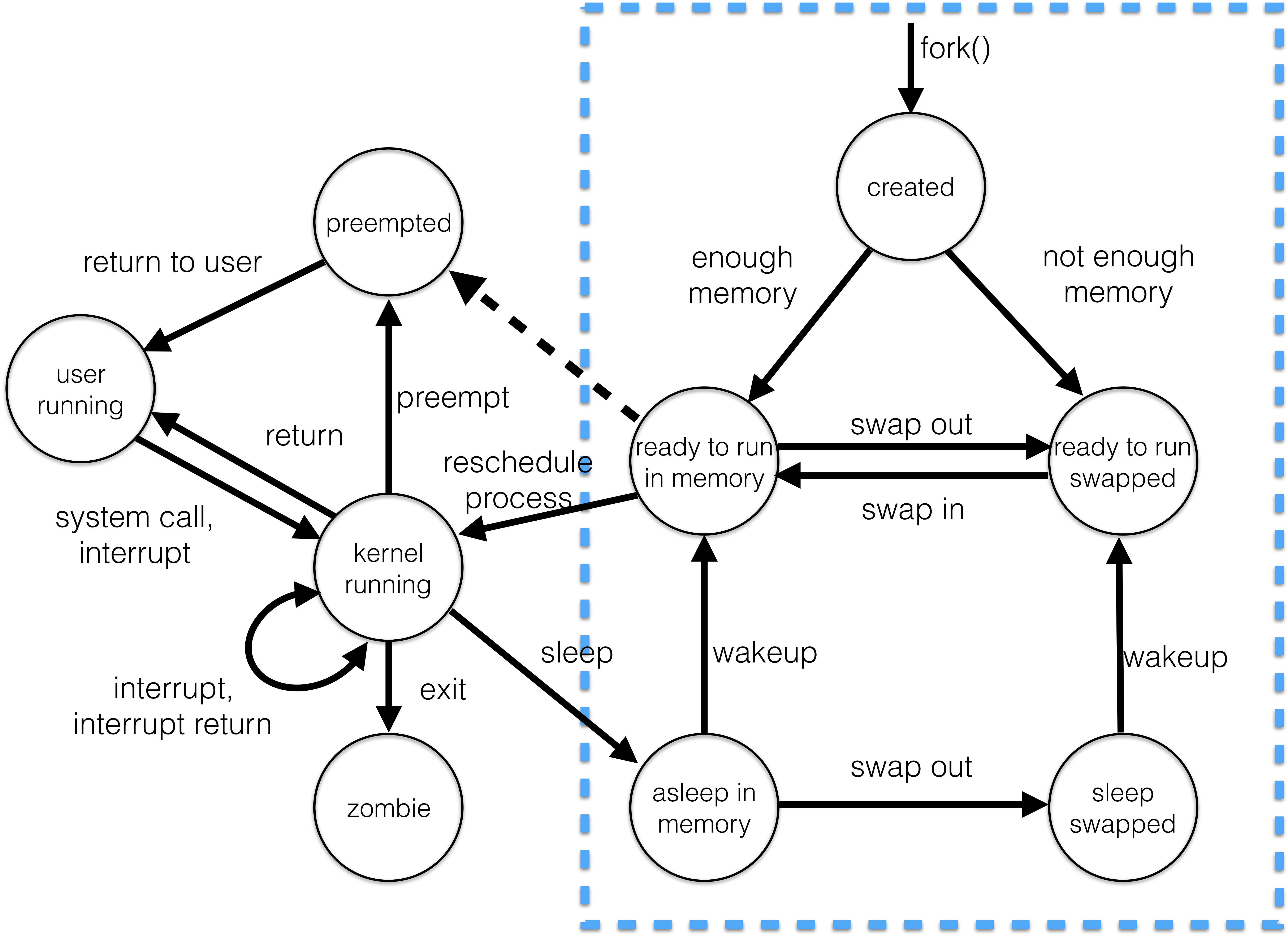
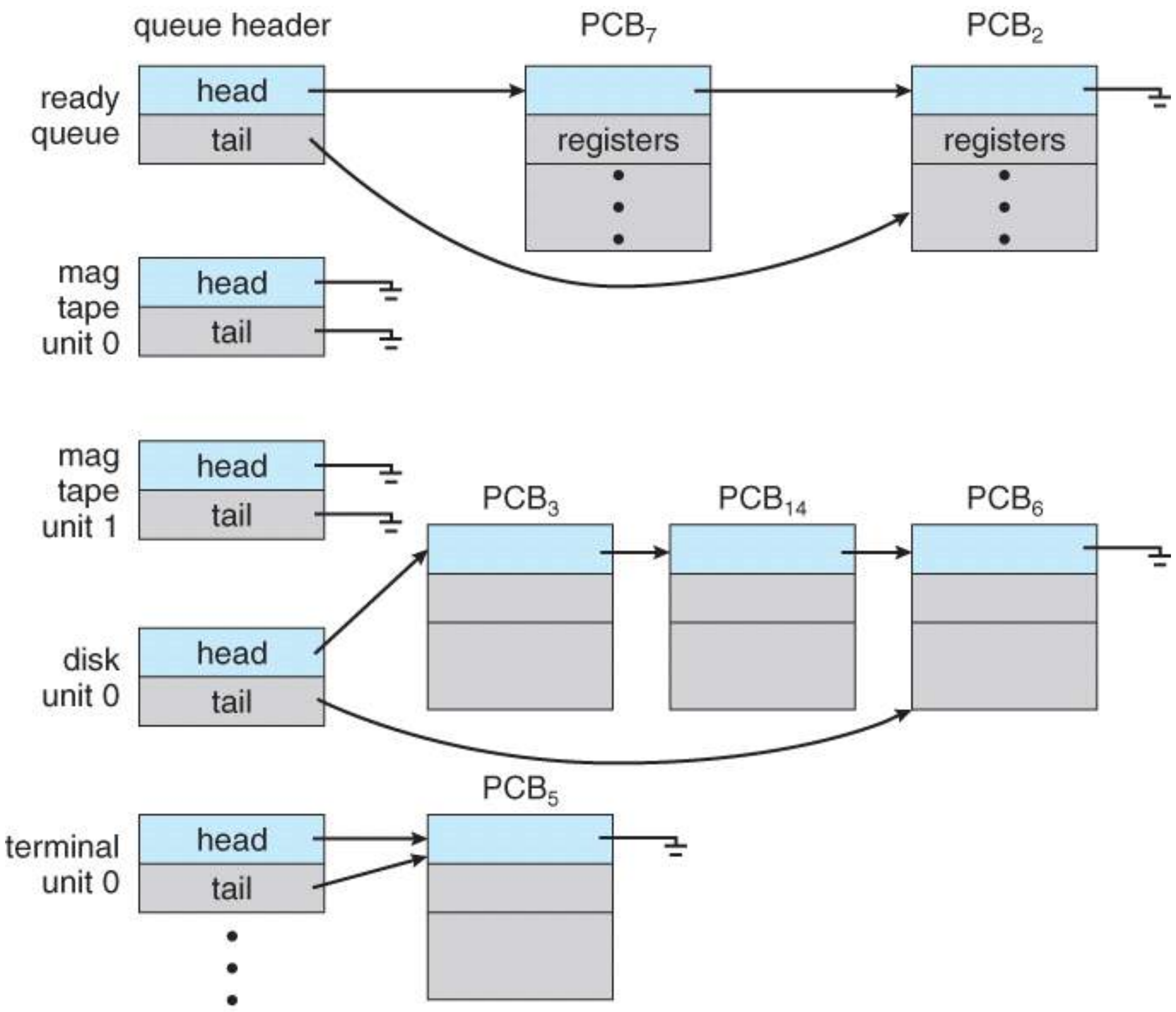
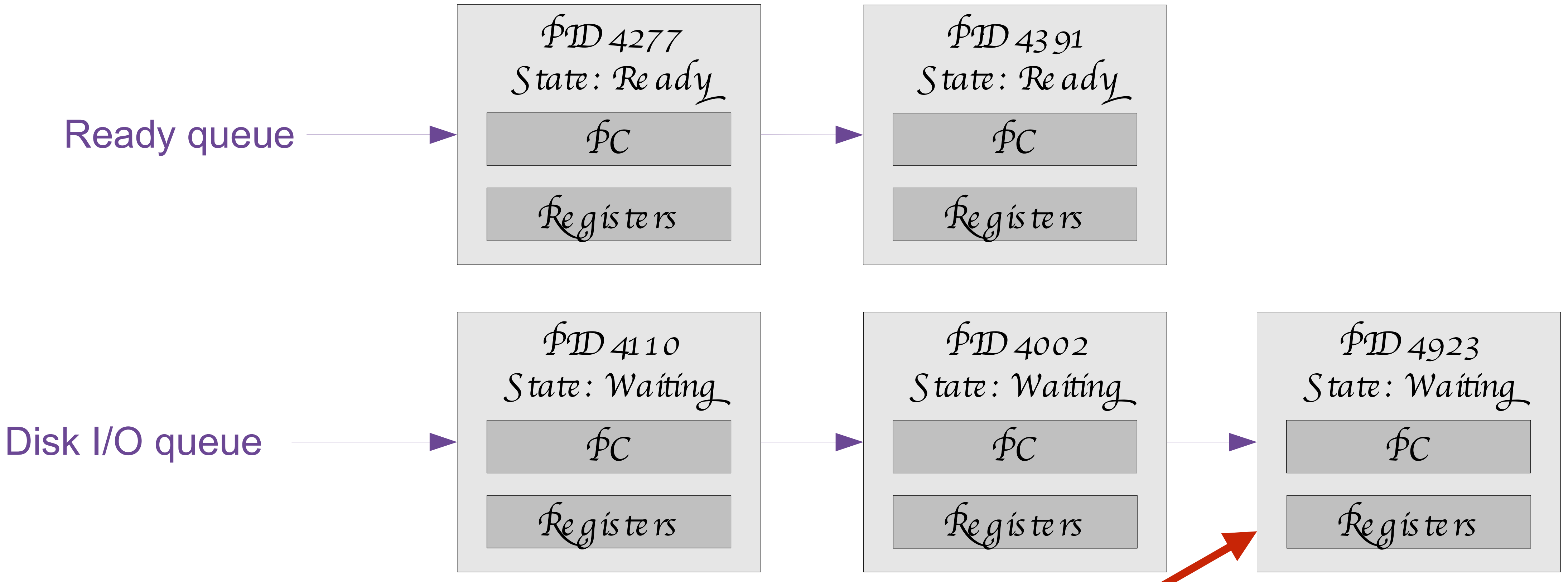


Figure adapted from Stallings' book

Ready queue and various I/O queues



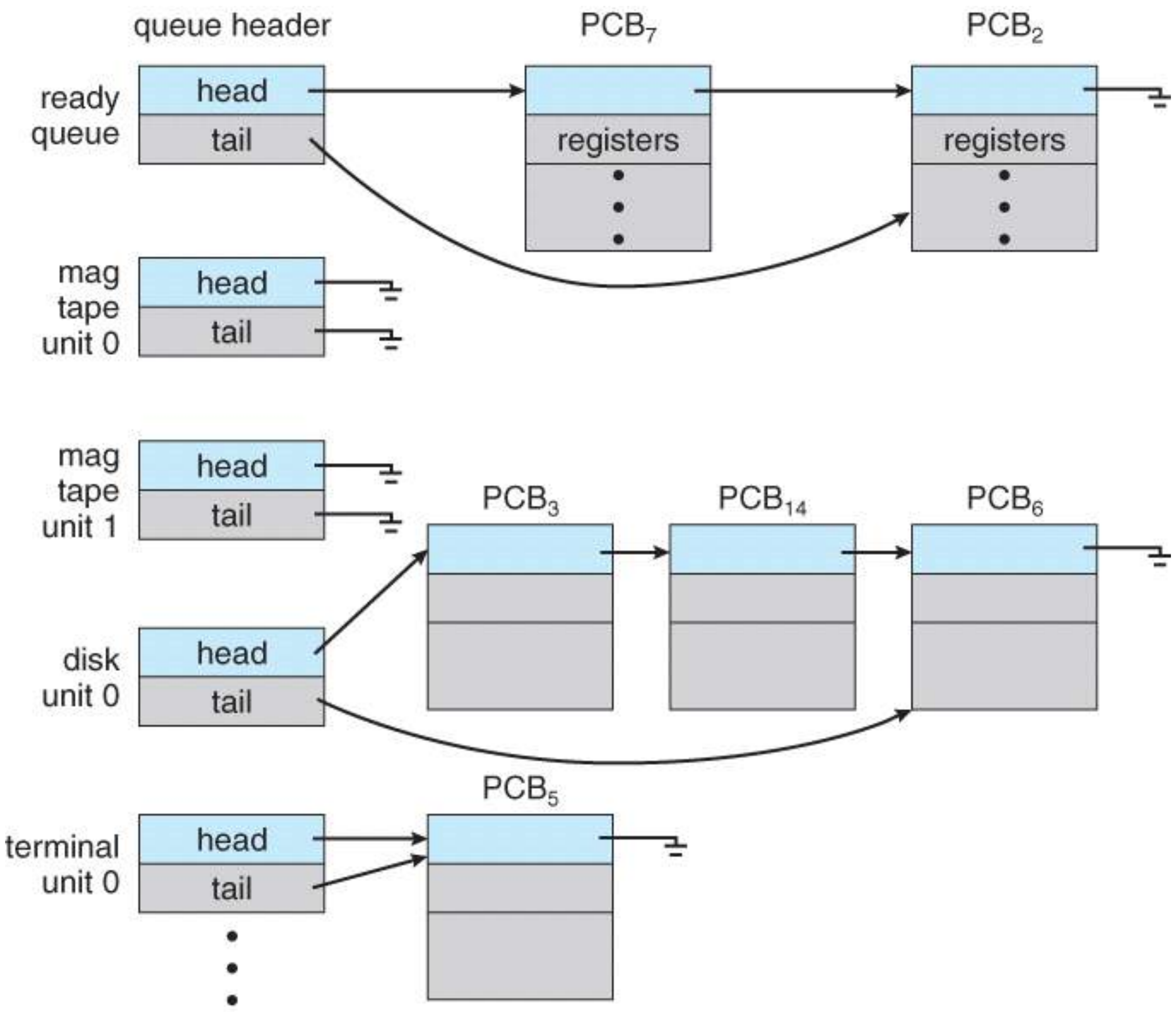


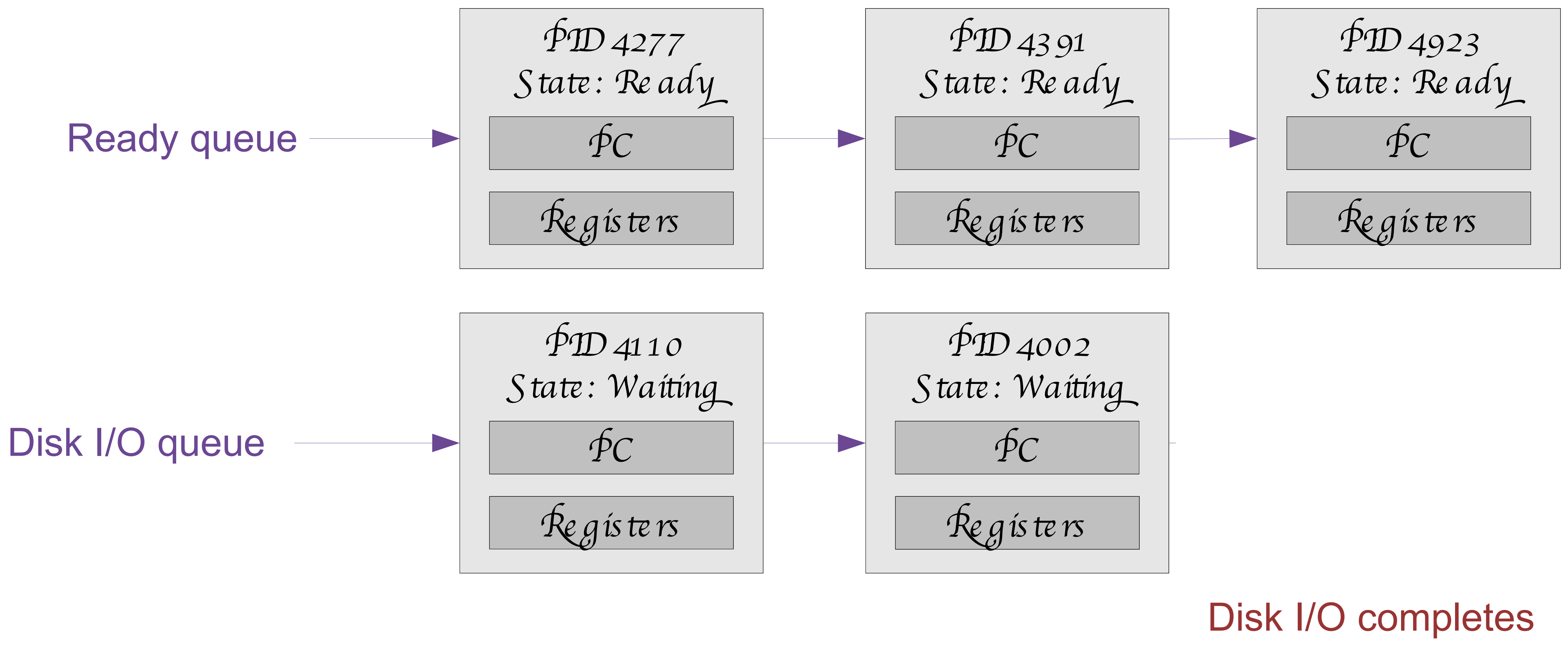
Process waiting

- 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.

Ready queue and various I/O queues



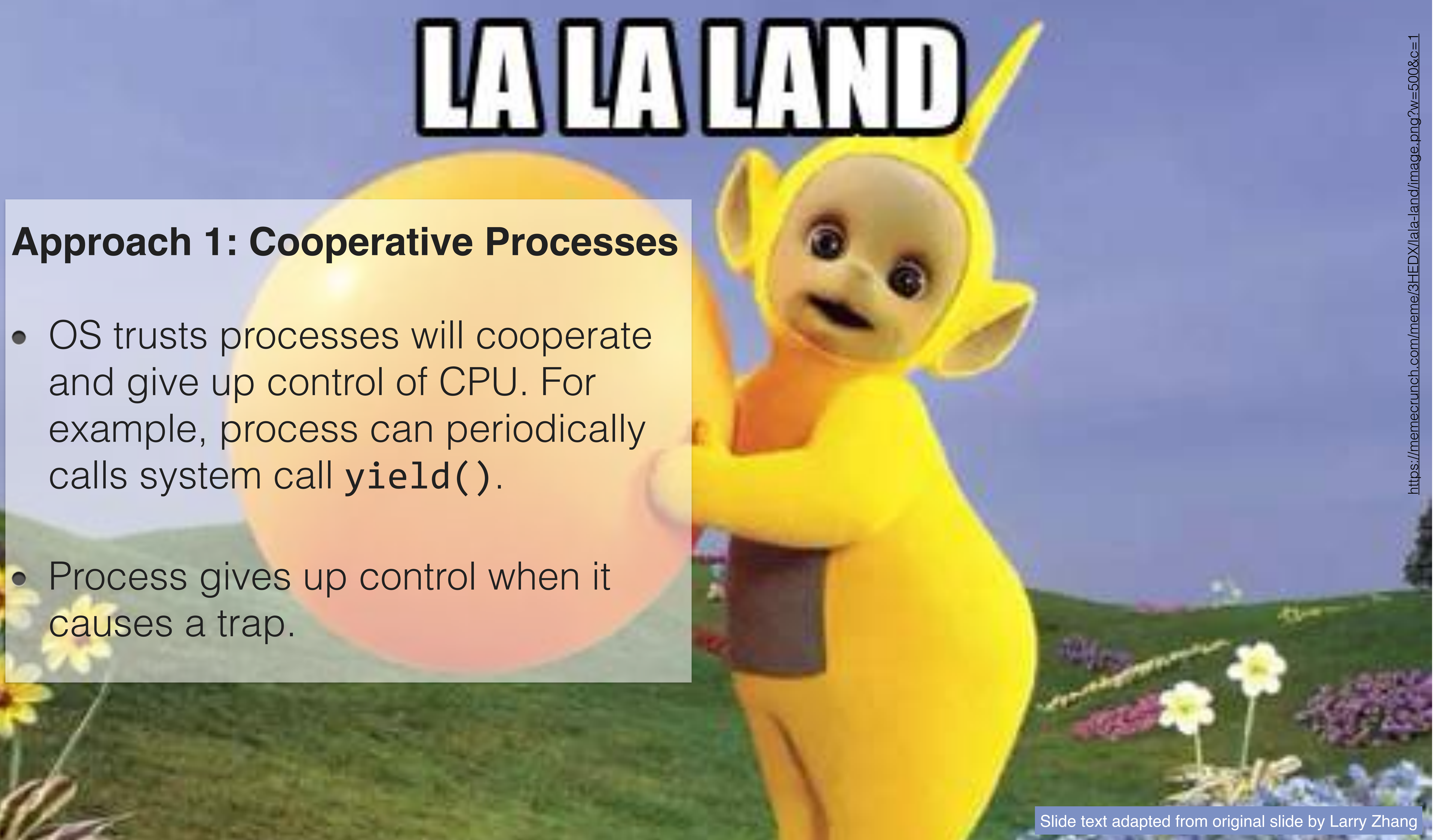


- OS maintains a set of queues
- Each PCB is queued on a state queue based on the process' current state.
- As processes change states, PCBs are unlinked from one queue and linked into another.

Adapted from Silberschatz, Galvin, and Gagne, 2009.

Main Question:

How can OS **regain control** of the CPU from a process so that it can switch to another process?



Approach 1: Cooperative Processes

- OS trusts processes will cooperate and give up control of CPU. For example, process can periodically calls system call `yield()`.

- Process gives up control when it causes a trap.

https://memecrunch.com/meme/3HEDX/lala-land/image.png?w=500&c=1

Slide text adapted from original slide by Larry Zhang

Approach 2: Non-Cooperative Processes

- OS takes control periodically (e.g., timer interrupt).
- Timer can be programmed to raise an interrupt periodically.
- When interrupt is raised, OS *Interrupt Handler* runs, and OS regains control.

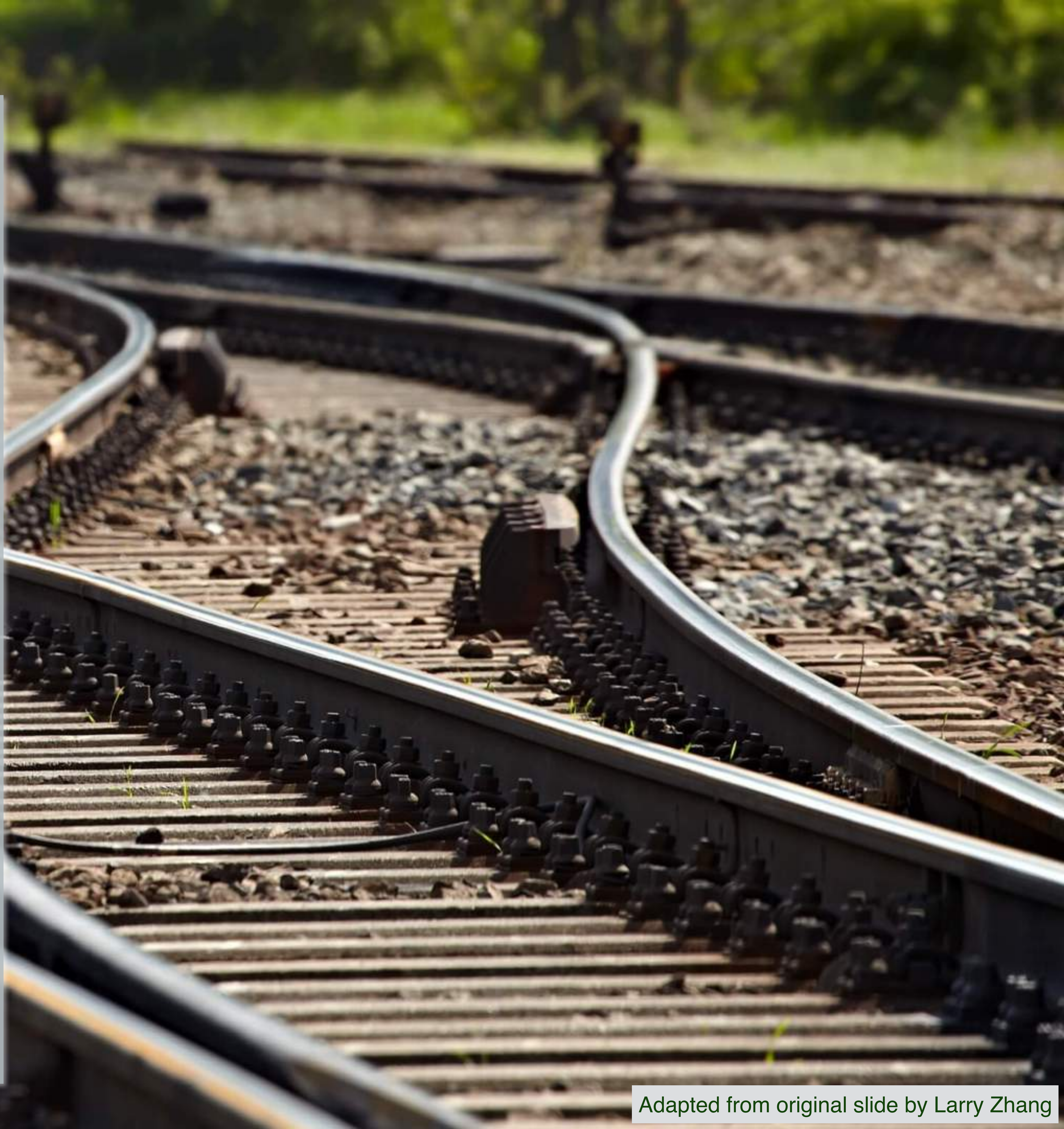
Slide text adapted from original slide by Larry Zhang

Now, OS has control. How to switch to another process?

- OS decides the process to which to switch (i.e., scheduler decides).

- OS executes a piece of assembly code (i.e., context switch).

- **Context switch:**
  1. Save register values of current process to kernel stack.
  2. Restore register values of the next process from its kernel stack.



CPU switch from process to process

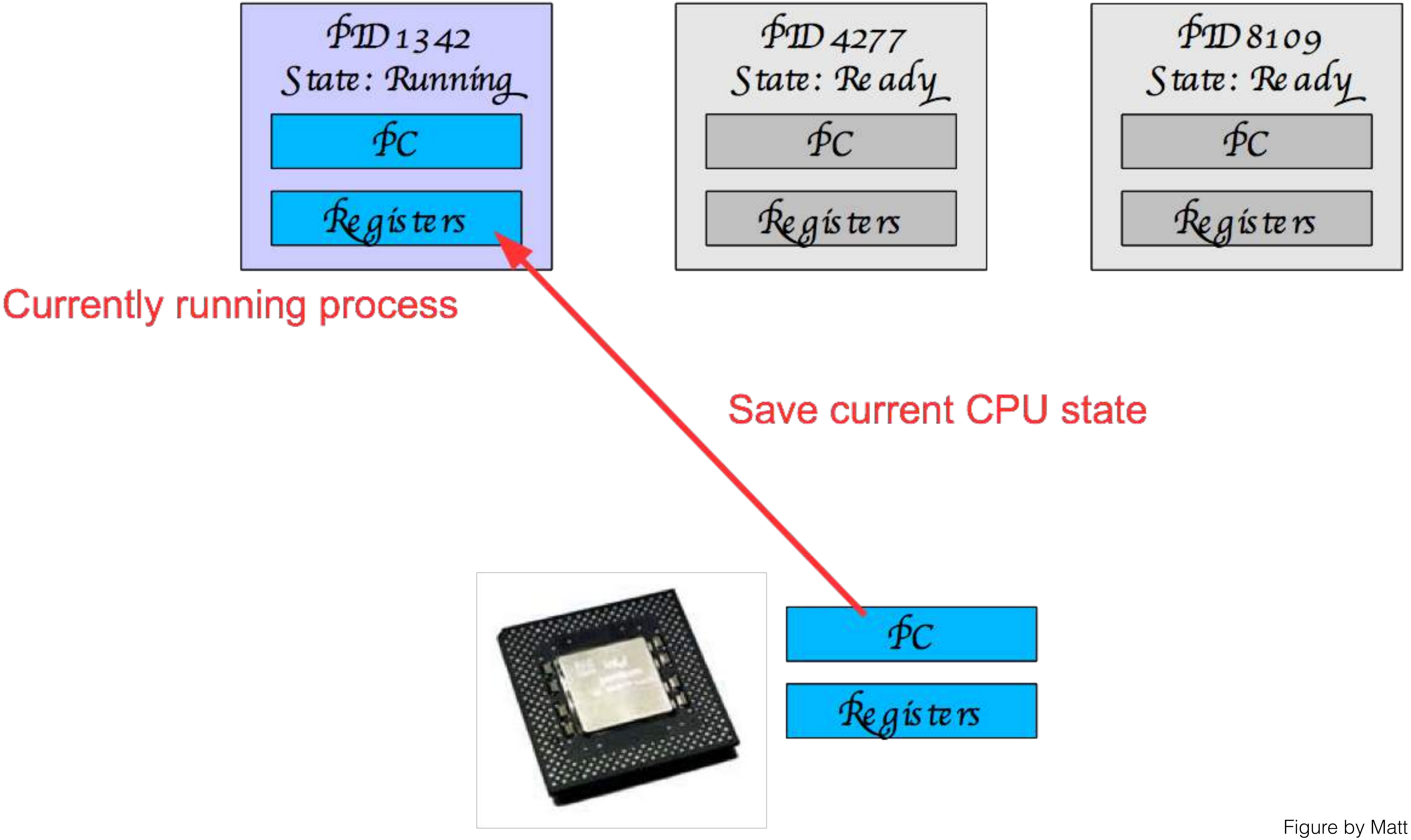


Figure by Matt Welsh, Harvard University.

CPU switch from process to process

PID 1342
State: Ready
PC
Registers

PID 4277
State: Ready
PC
Registers

PID 8109
State: Ready
PC
Registers

Suspend process

PC
Registers

Figure by Matt Welsh, Harvard University.

CPU switch from process to process

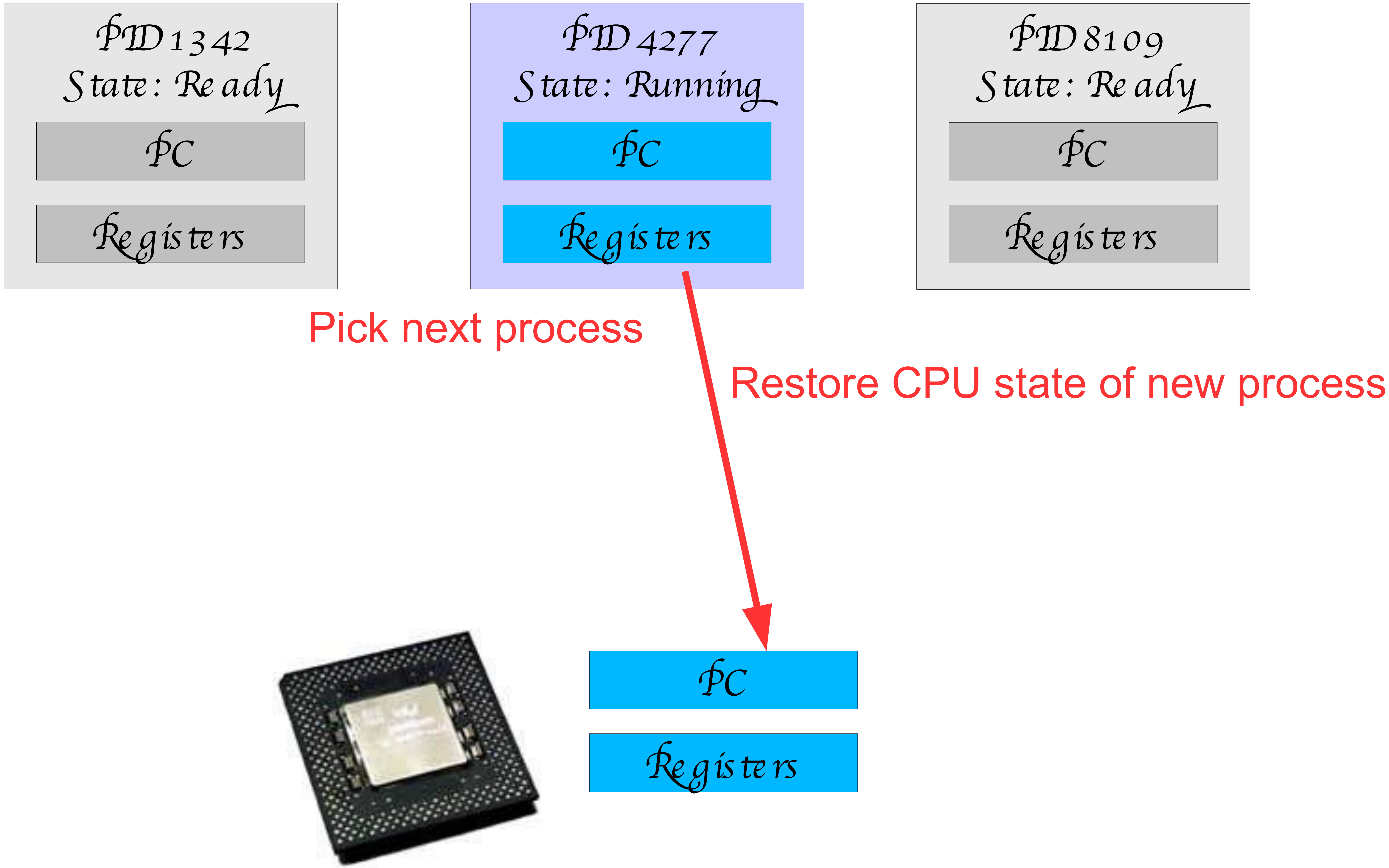


Figure by Matt Welsh, Harvard University.

Context switch in OS/161

kern/thread/thread.c

```
 * The current thread is queued appropriately and its state is changed
 * to NEWSTATE; another thread to run is selected and switched to.
 *
 * If NEWSTATE is S_SLEEP, the thread is queued on the wait channel
 * WC. Otherwise WC should be NULL.
 */
static
void
thread_switch(threadstate_t newstate, struct wchan *wc)
{
        struct thread *cur, *next;
        int spl;

        DEBUGASSERT(curcpu->c_curthread == curthread);
        DEBUGASSERT(curthread->t_cpu == curcpu->c_self);

        /* Explicitly disable interrupts on this processor */
        spl = splhigh();

        cur = curthread;

```

Context switch in OS/161

kern/thread/thread.c

```c
        } while (next == NULL);
        curcpu->c_isidle = false;

        /*
         * Note that curcpu->c_curthread may be the same variable as
         * curthread and it may not be, depending on how curthread and
         * curcpu are defined by the MD code. We'll assign both and
         * assume the compiler will optimize one away if they're the
         * same.
         */
        curcpu->c_curthread = next;
        curthread = next;

        /* do the switch (in assembler in switch.S) */
        switchframe_switch(&cur->t_context, &next->t_context);

        /*
         * When we get to this point we are either running in the next
         * thread, or have come back to the same thread again,
         * depending on how you look at it. That is,
```

Context switch in OS/161

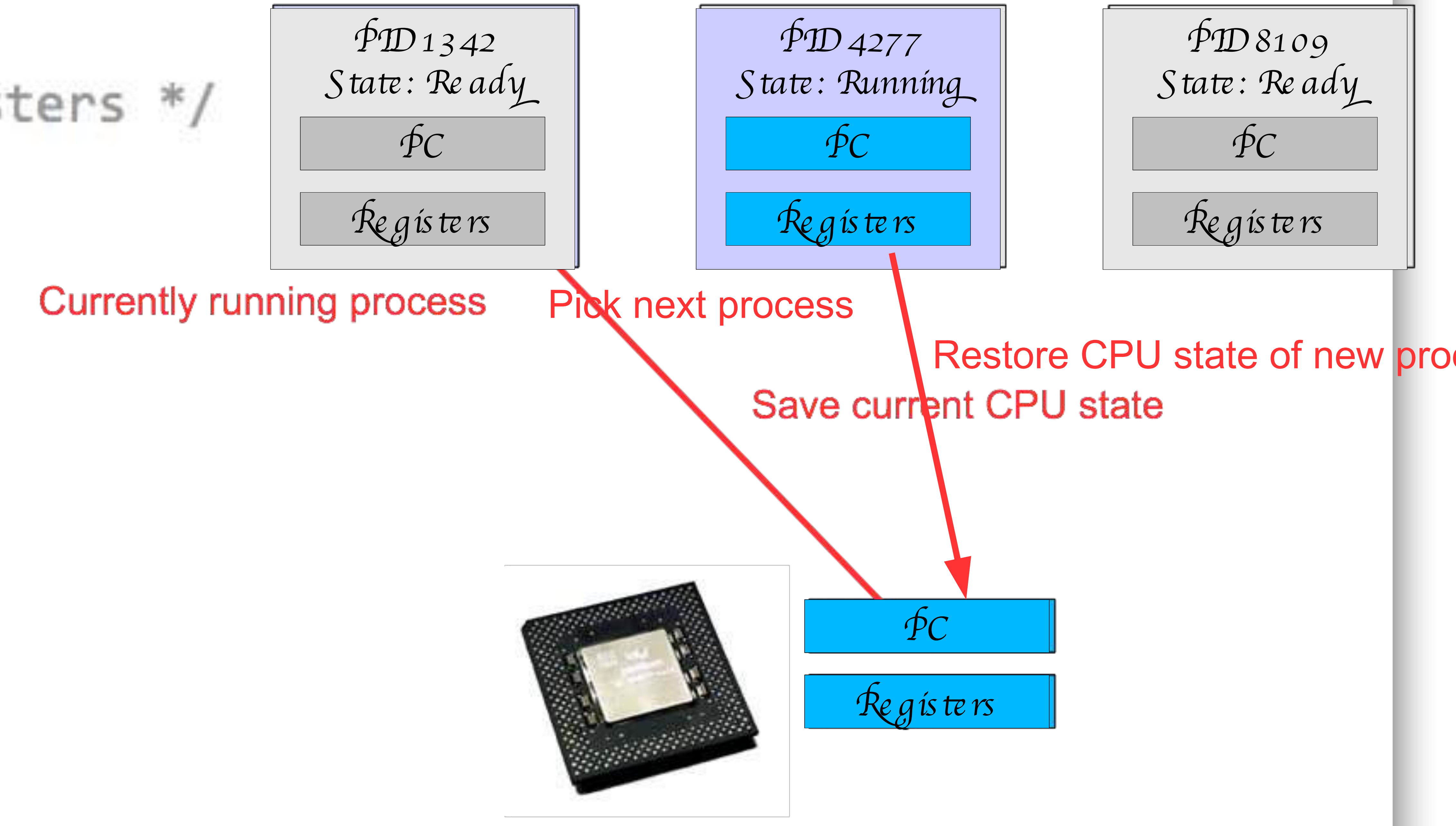
`src/kern/arch/mips/thread/switch.S`

```
/* Allocate stack space for saving 10 registers. 10*4 = 40 */
addi sp, sp, -40

/* Save the registers */
sw   ra, 36(sp)
sw   gp, 32(sp)
sw   s8, 28(sp)
sw   s6, 24(sp)
sw   s5, 20(sp)
sw   s4, 16(sp)
sw   s3, 12(sp)
sw   s2, 8(sp)
sw   s1, 4(sp)
sw   s0, 0(sp)

/* Store the old stack pointer in the old thread */
sw   sp, 0(a0)

/* Get the new stack pointer from the new thread */
lw   sp, 0(a1)
nop              /* delay slot for load */
```



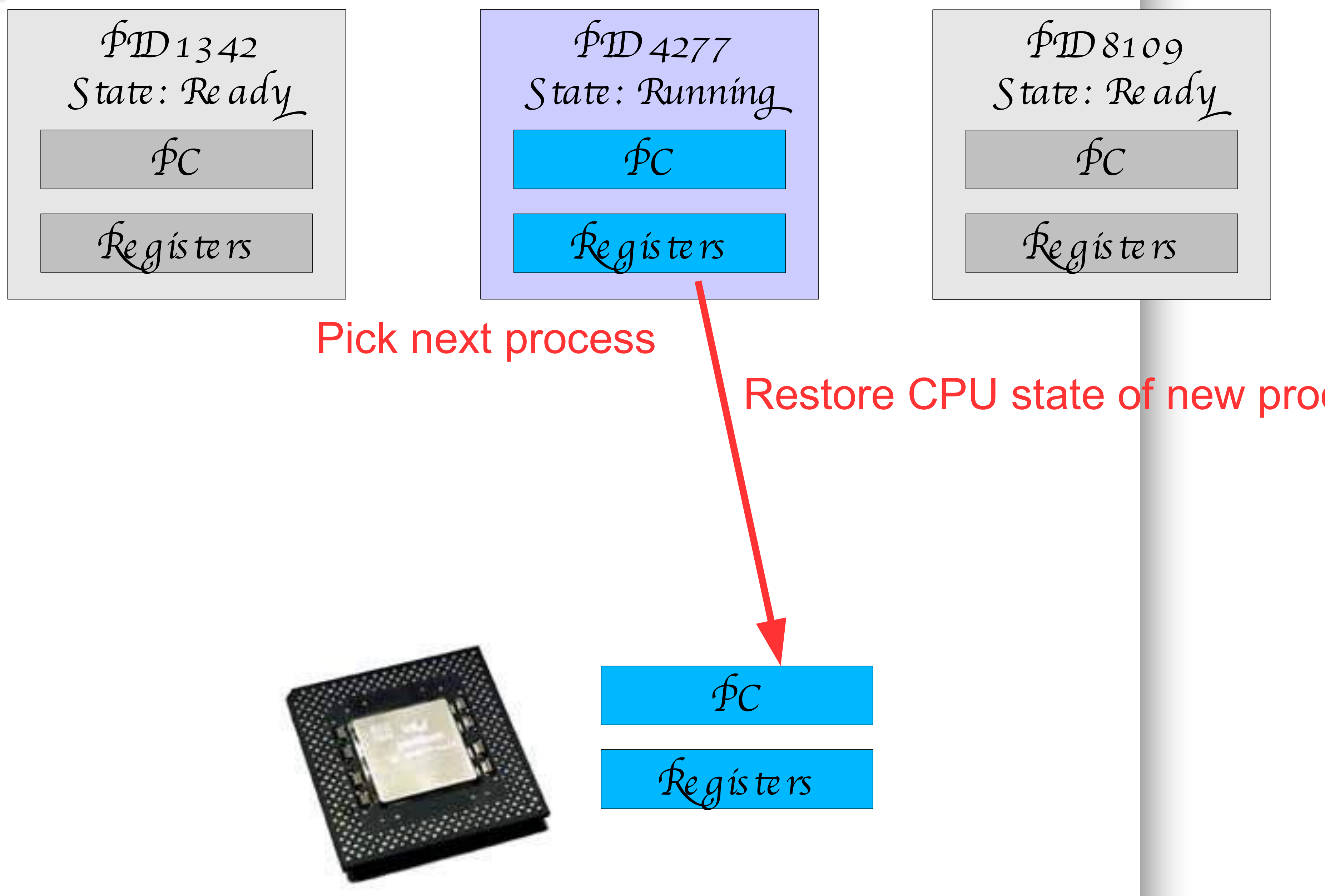
sw saves a word from a register into RAM.

Context switch in OS/161

`src/kern/arch/mips/thread/switch.S`

```
/* Now, restore the registers */
lw    s0, 0(sp)
lw    s1, 4(sp)
lw    s2, 8(sp)
lw    s3, 12(sp)
lw    s4, 16(sp)
lw    s5, 20(sp)
lw    s6, 24(sp)
lw    s8, 28(sp)
lw    gp, 32(sp)
lw    ra, 36(sp)
nop                     /* delay slot for load */

/* and return. */
j ra
addi sp, sp, 40         /* in delay slot */
.end switchframe_switch
```



lw loads a word memory into a register.

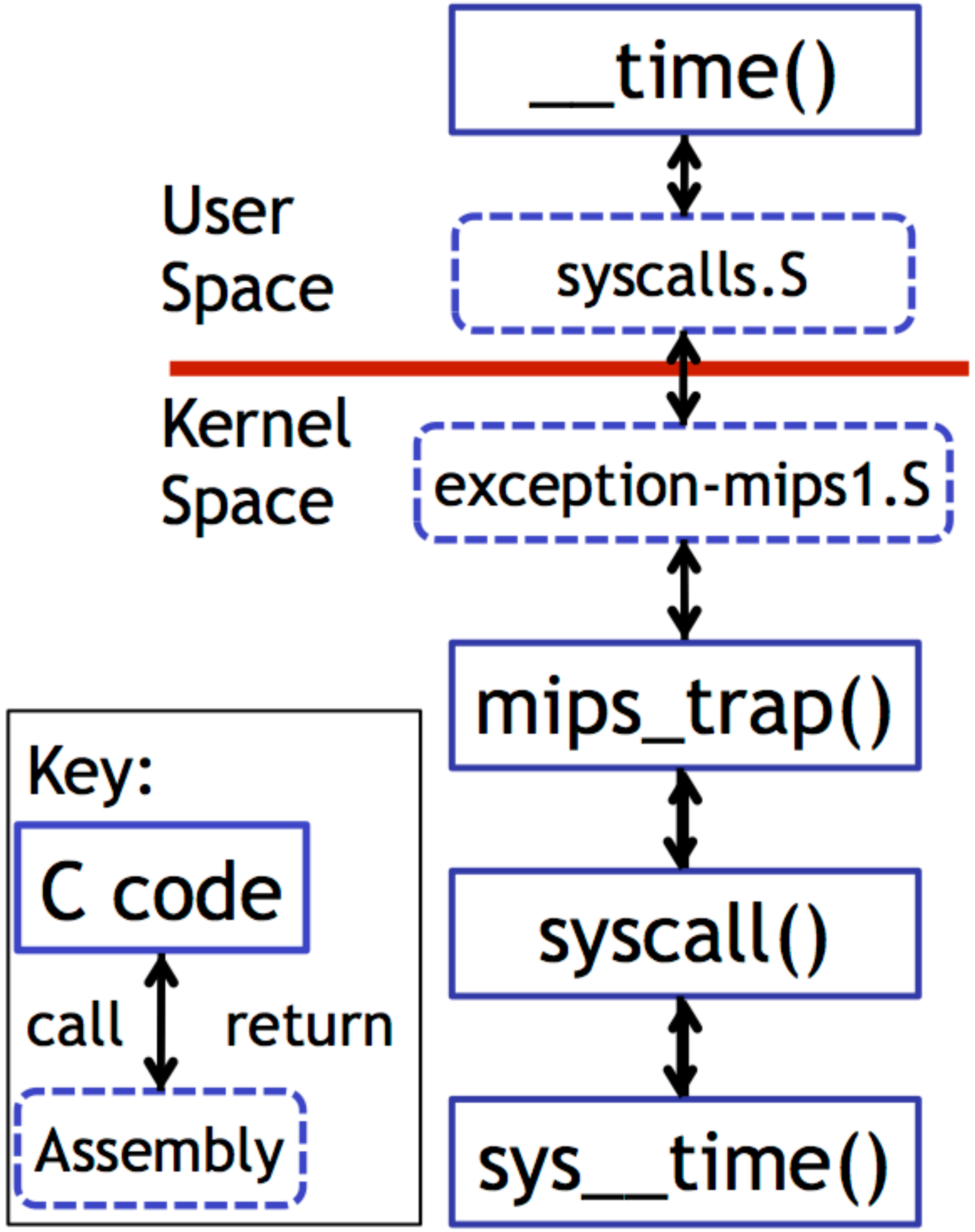
Summary

The challenge: **efficiently** virtualize CPU with **control.**

Mechanism: **time sharing** with **limited direct execution.**

- ➔ Process can perform restrict operations without messing around with hardware
  - ◆ System calls
- ➔ OS can switch from one process to another
  - ◆ Cooperative vs noncooperative
  - ◆ Timer interrupt
  - ◆ Context switch

OS161 System Call Example: time



/src/user/bin/sh/sh.c (or sbin and testbin pgms)

(Invokes system call)

/src/user/lib/libc/arch/mips/syscalls-mips.S

(auto generated!)

(Sets up syscall number and traps to kernel)

/src/kern/arch/mips/locore/exception-mips1.S

(asm code to save registers)

/src/kern/arch/mips/locore/trap.c

(trap handler, determines the trap is for a syscall)

/src/kern/arch/mips/syscall/syscall.c

(syscall handler, determines which syscall is requested)

/src/kern/syscall/time_syscalls.c (time handler)

Creating a New System Call in OS161

- Define the new system call code in src/kern/include/kern/syscall.h
- Define the prototypes for the new syscall
  - Kernel space: src/kern/include/syscall.h
  - User space: src/user/include/unistd.h
- Write the source code for it in src/kern/syscall/new_syscall.c
  - Be sure to include this new file in src/kern/conf/conf.kern! (so it's included in the build path)
- If necessary, define any new error codes in src/kern/include/kern/errno.h
- Add a case in the handler switch statement in src/kern/arch/mips/syscall/syscall.c
- Create a test program in src/testbin
- Rebuild both kernel and user level programs