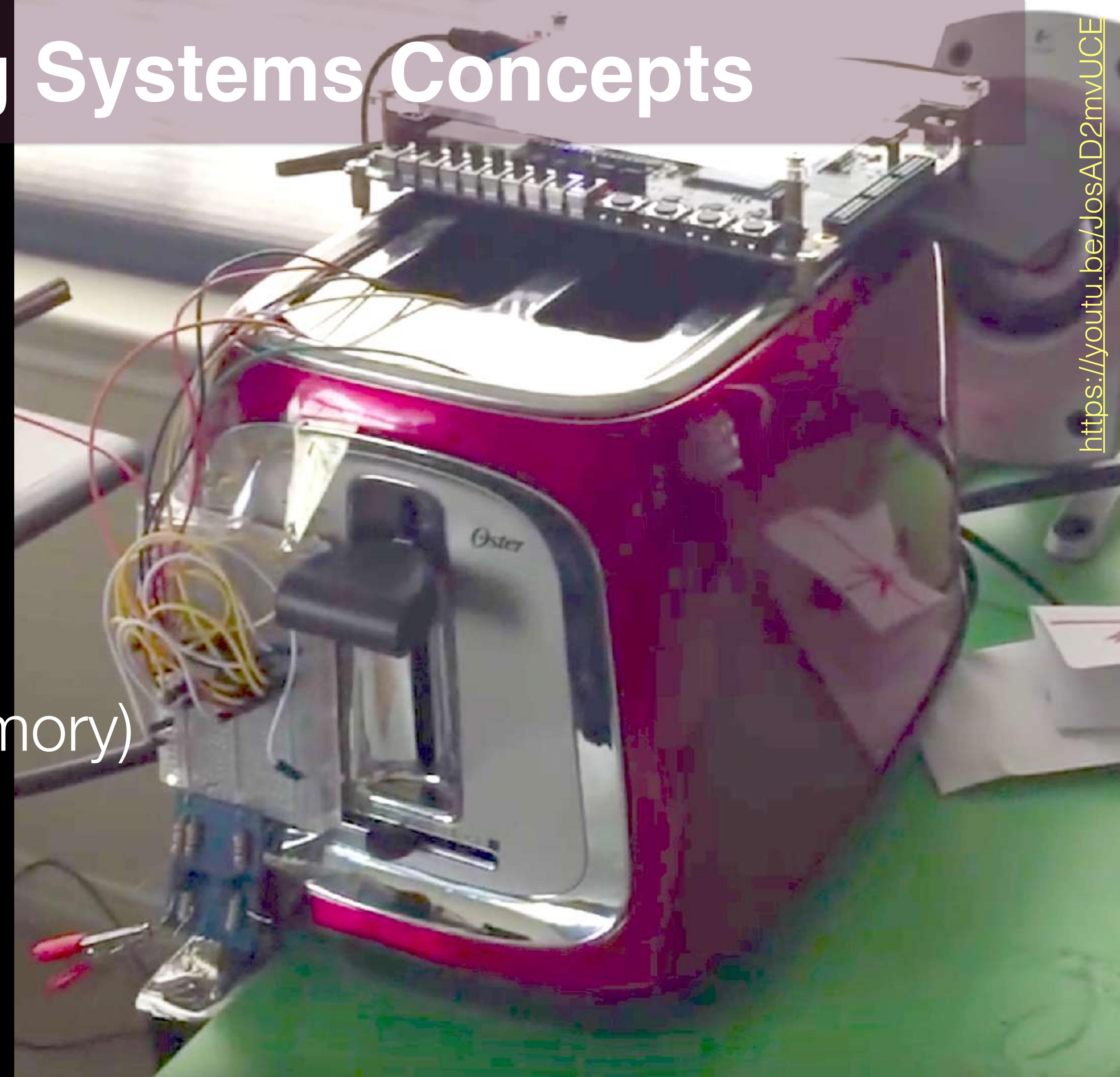
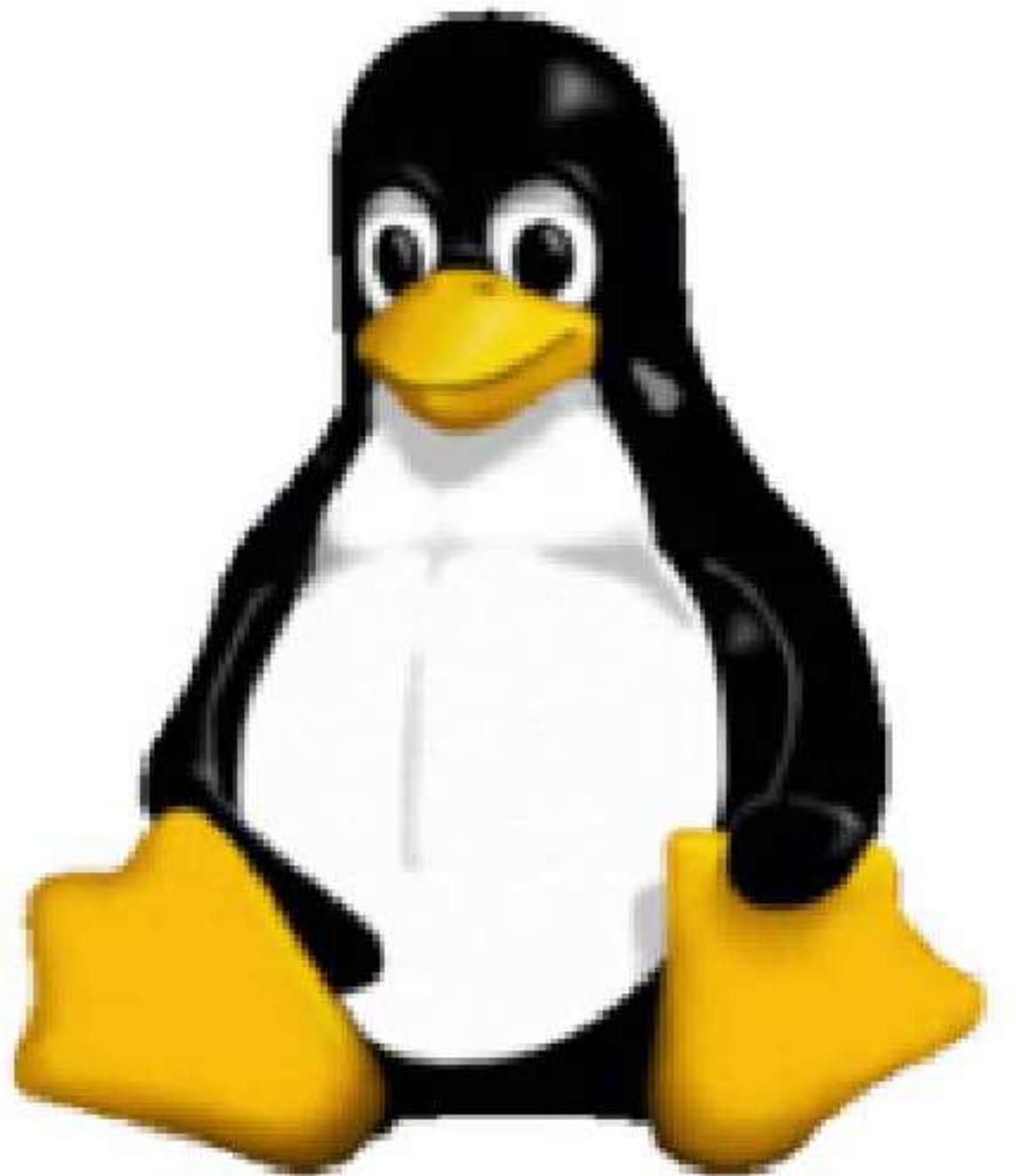


CSE 4001 Operating Systems Concepts

Topics

- OS Structures
- Processes and Threads
- CPU Scheduling
- Process Synchronization
- Memory (and Virtual Memory)
- File-Systems
- I/O Systems





OneCore to rule them all: How Windows Everywhere finally happened

Microsoft promised developers that Windows would run anywhere. This summer, it finally will.

PETER BRIGHT - 5/20/2016, 7:00 AM



Everywhere Windows 10 can be. And on the server, too, though there it gets a different branding.

One OS to rule them all, One OS to find them, One OS to bring them all and in the darkness bind them

restart your computer. If this screen appears again, follow these steps:

Check to make sure any new hardware or software is properly installed. If this is a new installation, ask your hardware or software manufacturer for any Windows updates you might need.

If problems continue, disable or remove any newly installed hardware or software. Disable BIOS memory options such as caching or shadowing. If you need to use Safe Mode to remove or disable components, restart your computer, press F8 to select Advanced Startup Options, and then select Safe Mode.

Technical information:

*** STOP: 0x000000E2 (0x00000000, 0x00000000, 0x00000000, 0x00000000)

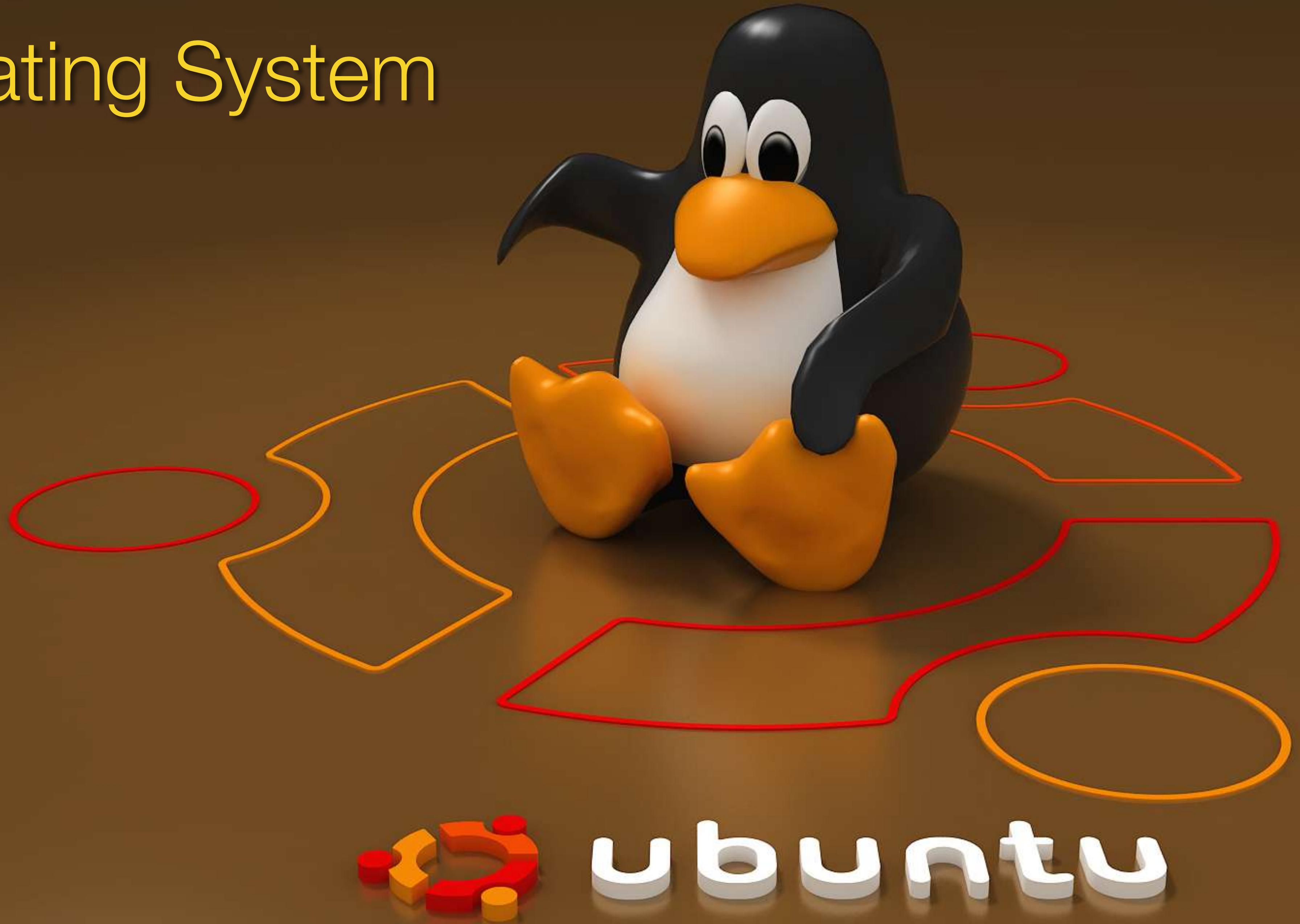
Beginning dump of physical memory

Physical memory dump complete.

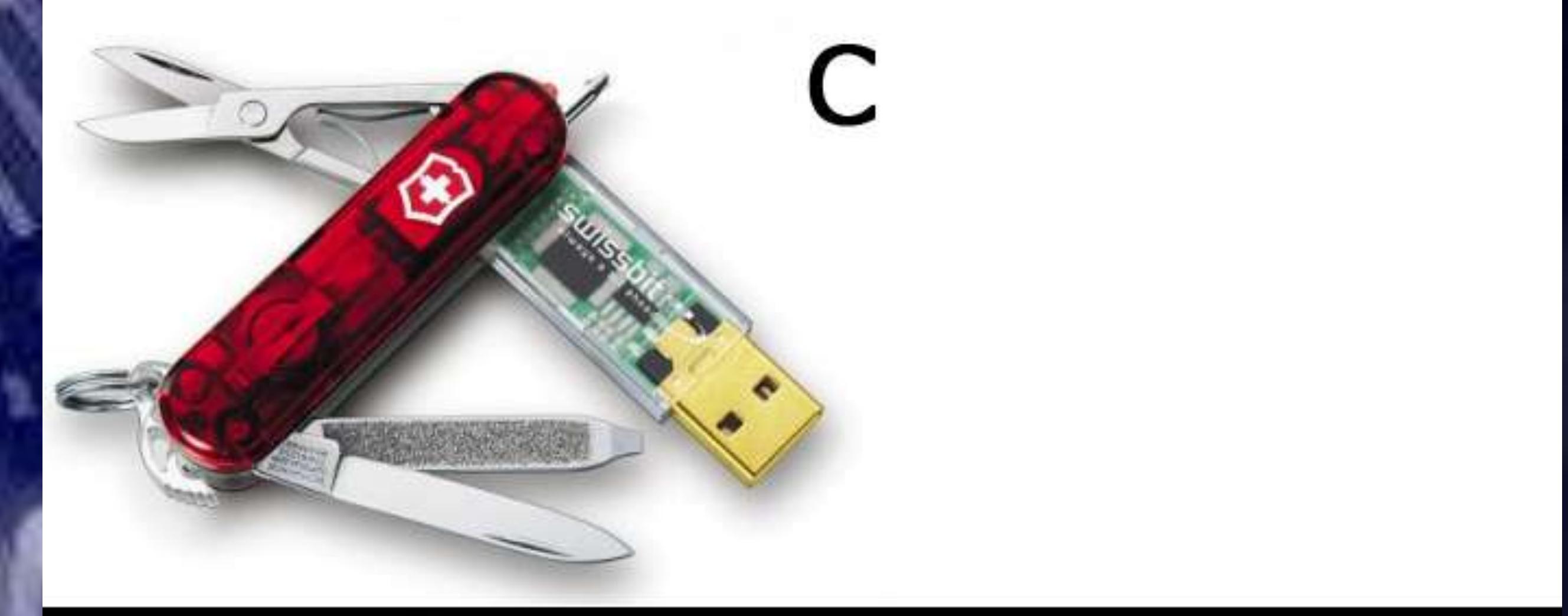
Contact your system administrator or technical support group for further assistance.



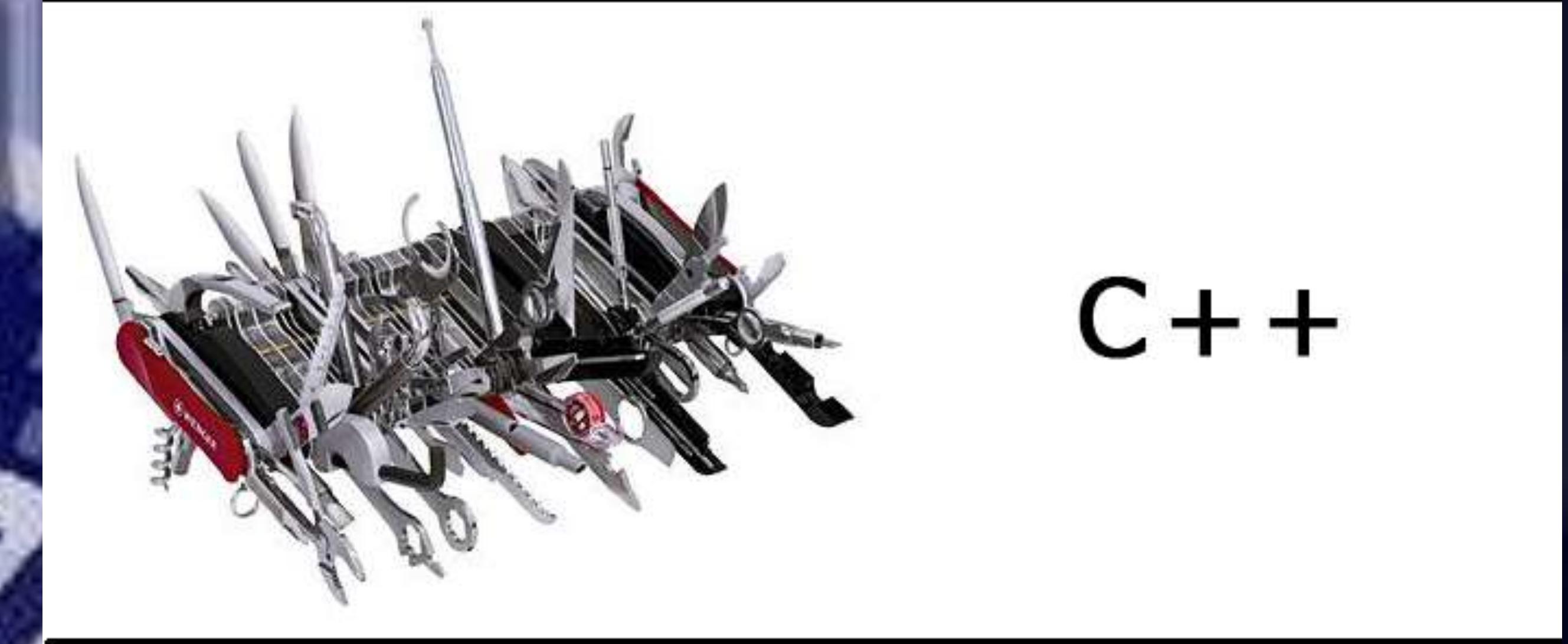
Operating System



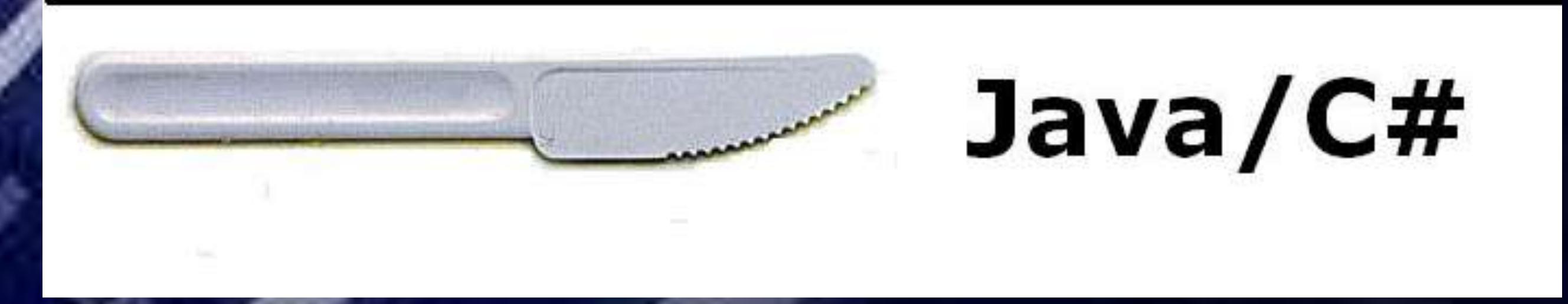
Programming language



C



C++



Java/C#

Assignments and Exams

| Type | % of Total Grade |
|---------------|------------------|
| Midterm exams | 30% |
| Final exam | 40% |
| Assignments | 25% |
| Quizzes | 5% |

Office Hours

Tuesdays and Thursdays:
from 2.00pm to 3.00pm
Harris Center for Science
and Engineering

208

DR. E. RIBEIRO
COMPUTER
SCIENCES

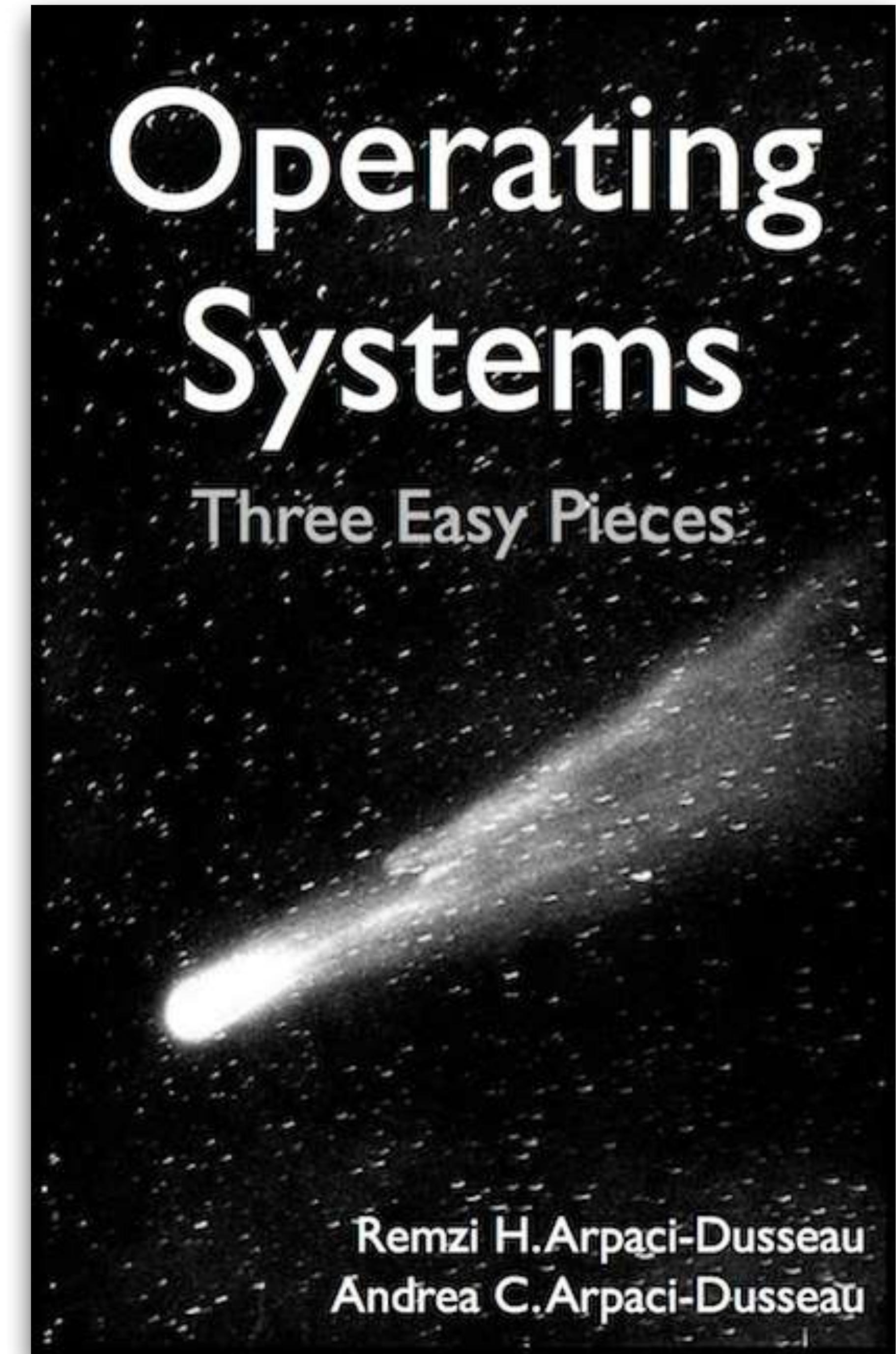
Book (required)

Operating Systems: Three Easy Pieces

Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau

Book and materials available from:

<http://pages.cs.wisc.edu/~remzi/OSTEP/>



Operating Systems: Three Easy Pieces

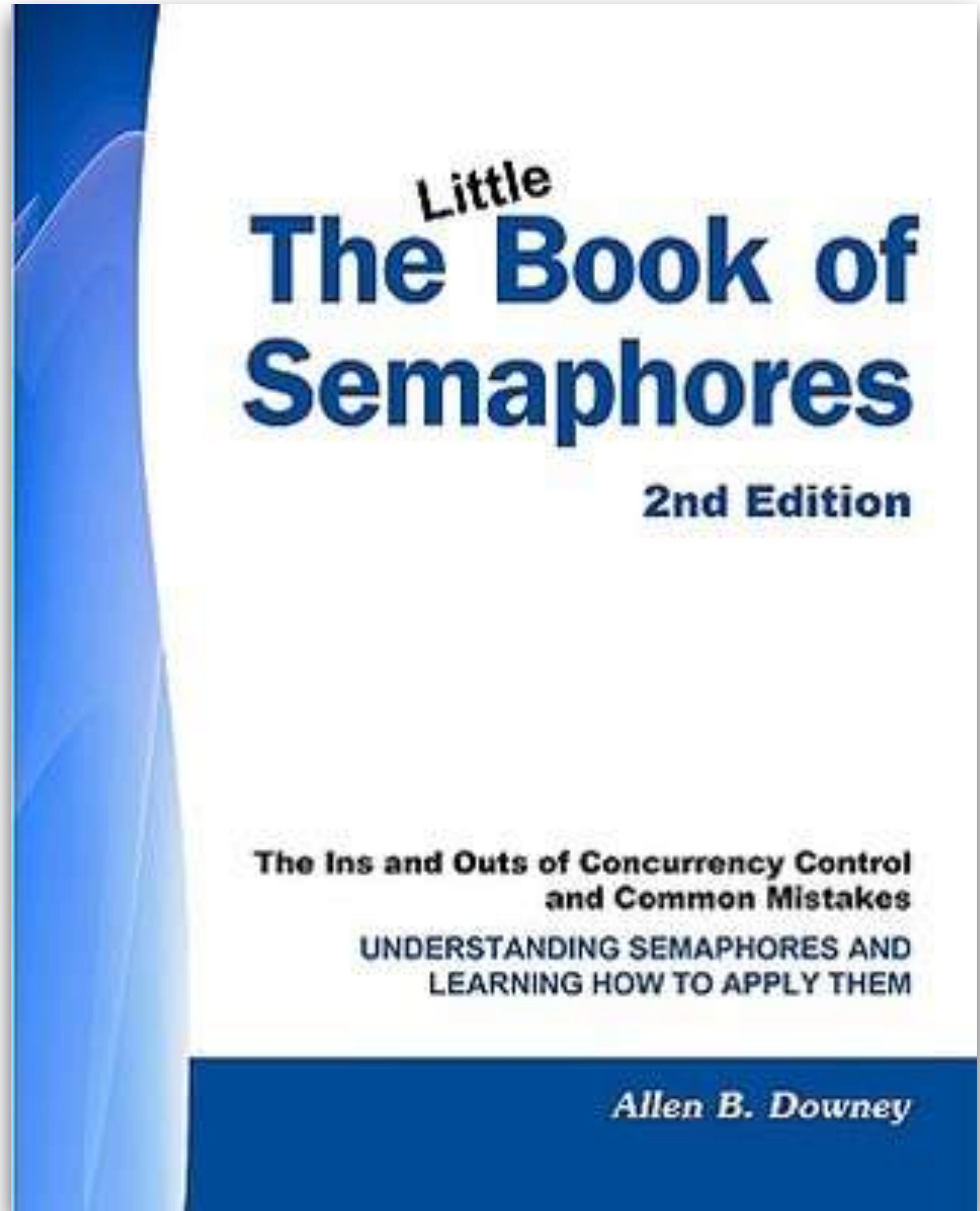
Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau

| Intro | Virtualization | | Concurrency | Persistence | Appendices |
|--|--|--|--|---|----------------------------------|
| Preface | 3 Dialogue | 12 Dialogue | 25 Dialogue | 35 Dialogue | Dialogue |
| TOC | 4 Processes | 13 Address Spaces | 26 Concurrency and Threads <small>code</small> | 36 I/O Devices | Virtual Machines |
| 1 Dialogue | 5 Process API <small>code</small> | 14 Memory API | 27 Thread API | 37 Hard Disk Drives | Dialogue |
| 2 Introduction <small>code</small> | 6 Direct Execution | 15 Address Translation | 28 Locks | 38 Redundant Disk Arrays (RAID) | Monitors |
| | 7 CPU Scheduling | 16 Segmentation | 29 Locked Data Structures | 39 Files and Directories | Dialogue |
| | 8 Multi-level Feedback | 17 Free Space Management | 30 Condition Variables | 40 File System Implementation | Lab Tutorial |
| | 9 Lottery Scheduling <small>code</small> | 18 Introduction to Paging | 31 Semaphores | 41 Fast File System (FFS) | Systems Labs |
| 10 Multi-CPU Scheduling | 19 Translation Lookaside Buffers | 32 Concurrency Bugs | 42 FSCK and Journaling | 42 xv6 Labs | |
| | 20 Advanced Page Tables | 33 Event-based Concurrency | 43 Log-structured File System (LFS) | | |
| | 21 Swapping: Mechanisms | 34 Summary | 44 Data Integrity and Protection | | |
| | 22 Swapping: Policies | | 45 Summary | | |
| | 23 Case Study: VAX/VMS | | 46 Dialogue | | |
| | 24 Summary | | 47 Distributed Systems | | |
| | | | 48 Network File System (NFS) | | |
| | | | 49 Andrew File System (AFS) | | |
| | | | 50 Summary | | |

Another book

The little book of semaphores

Allen Downey



Book and materials available from:

[http://www.greenteapress.com/
semaphores/](http://www.greenteapress.com/sema...)

Contents

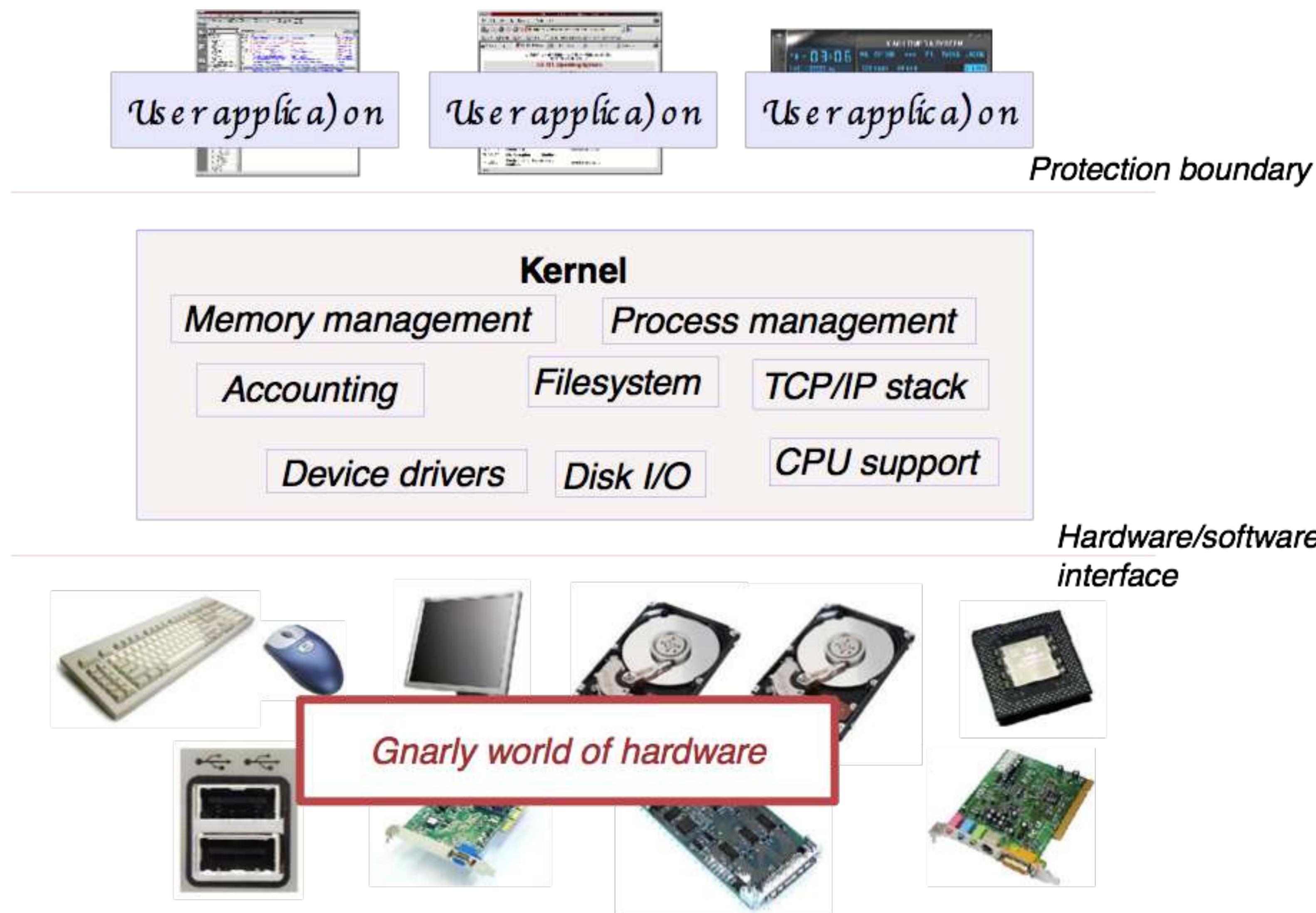
- What is an operating system?
- OS functions

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

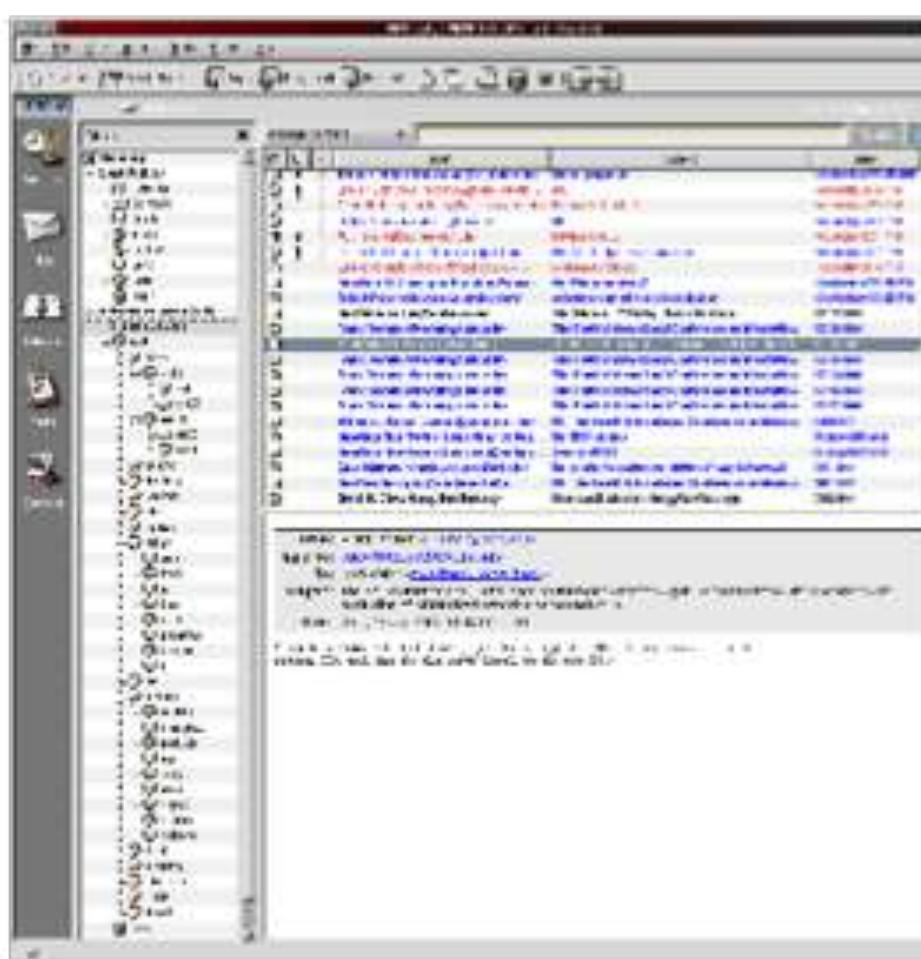
What is an operating system?

Software that provides an elaborate illusion to applications



One OS Function: Concurrency

Give every application the illusion of having its own CPU!



I think I have my own CPU



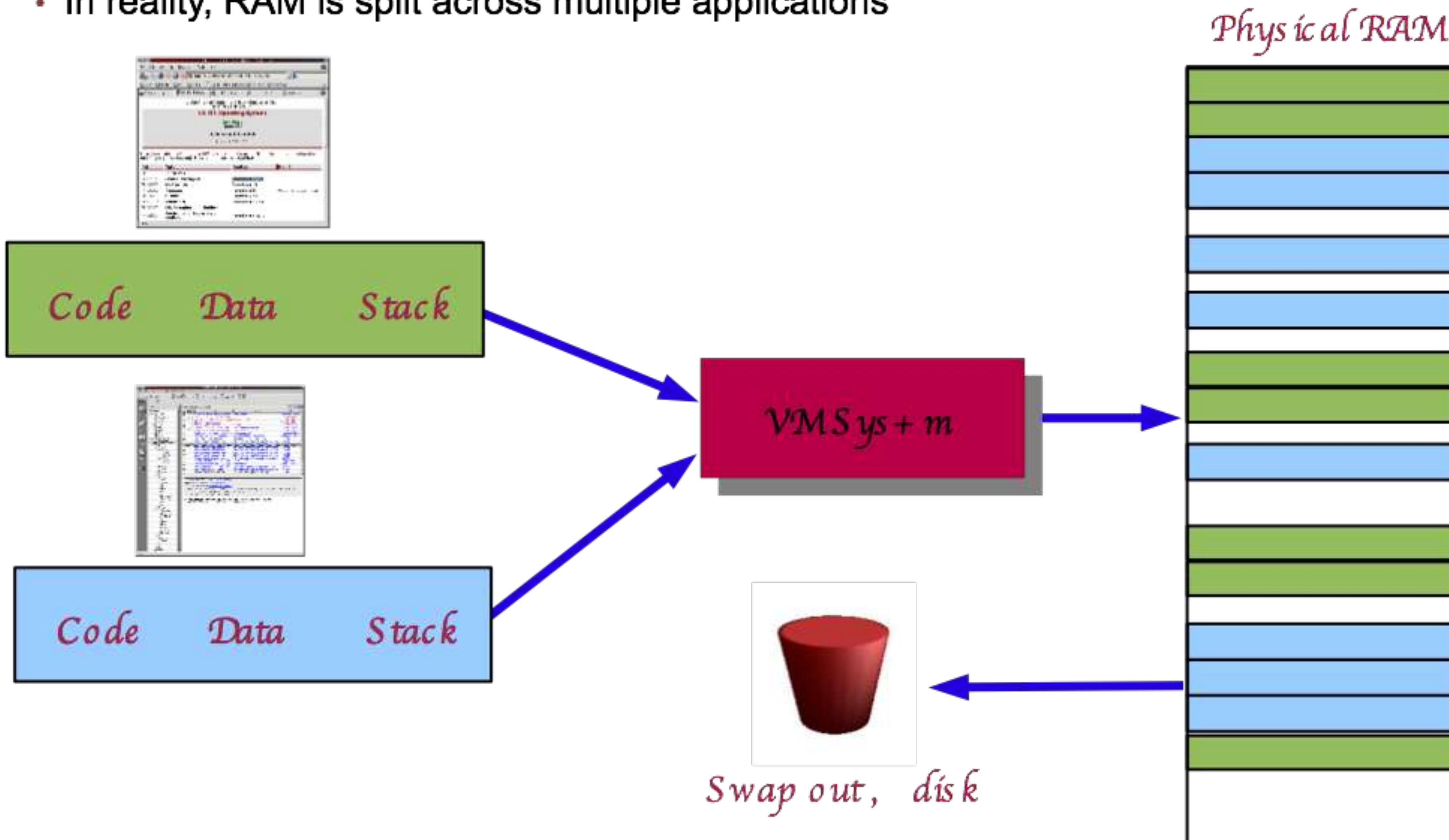
So do I



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

OS Overview

Contents

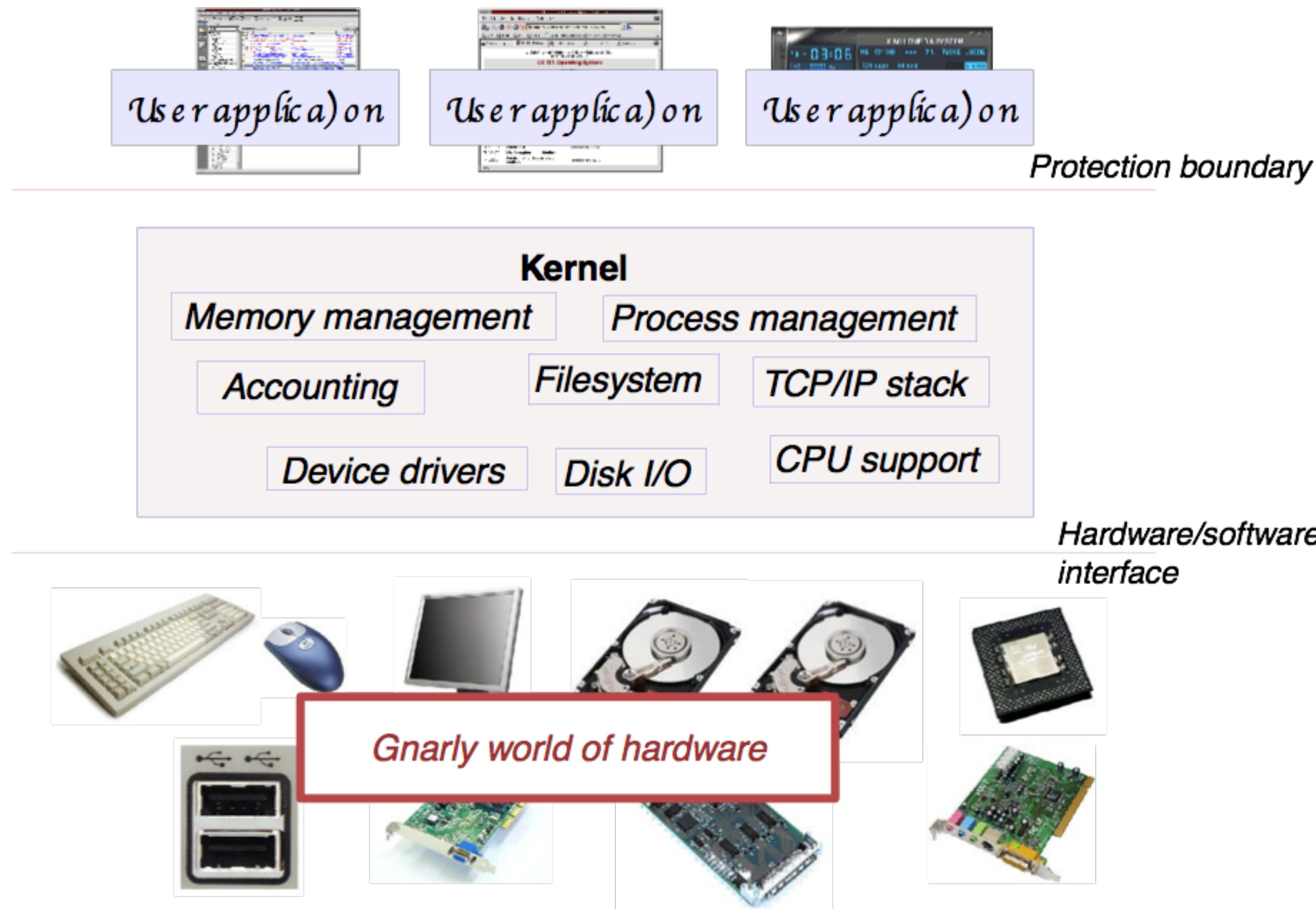
- What is an operating system?
- OS functions

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

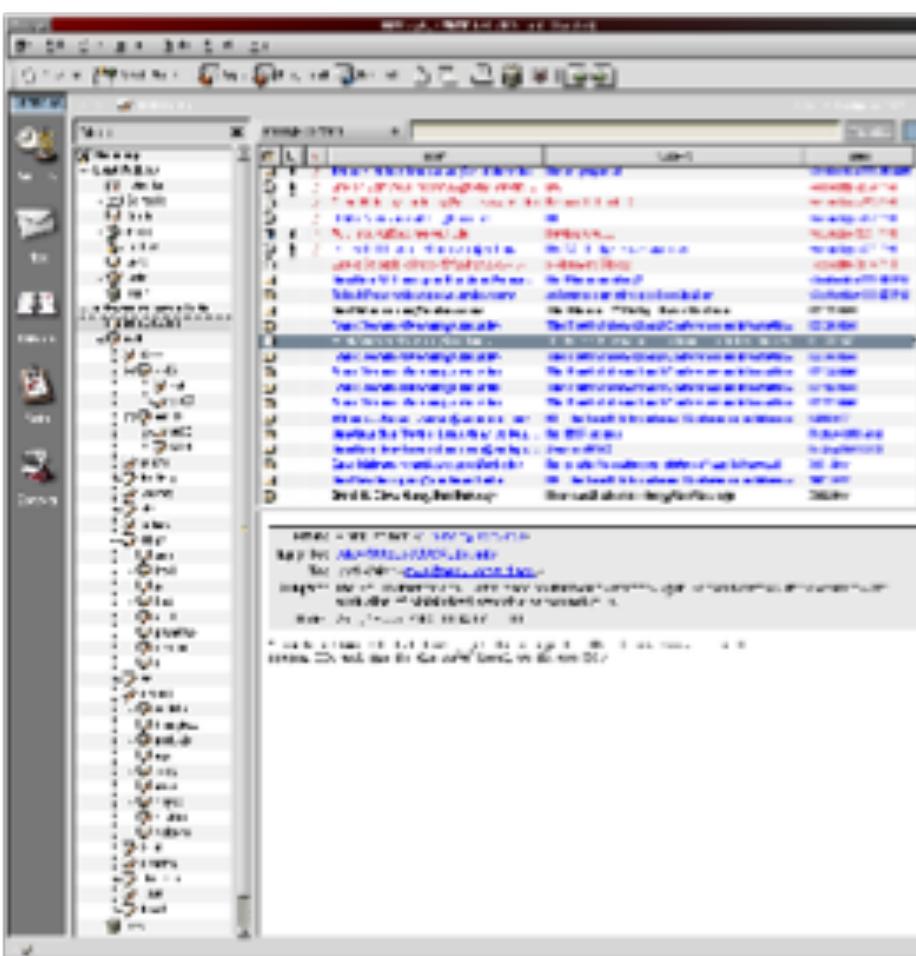
What is an operating system?

Software that provides an elaborate illusion to applications



One OS Function: Concurrency

Give every application the illusion of having its own CPU!



I think I have my own CPU

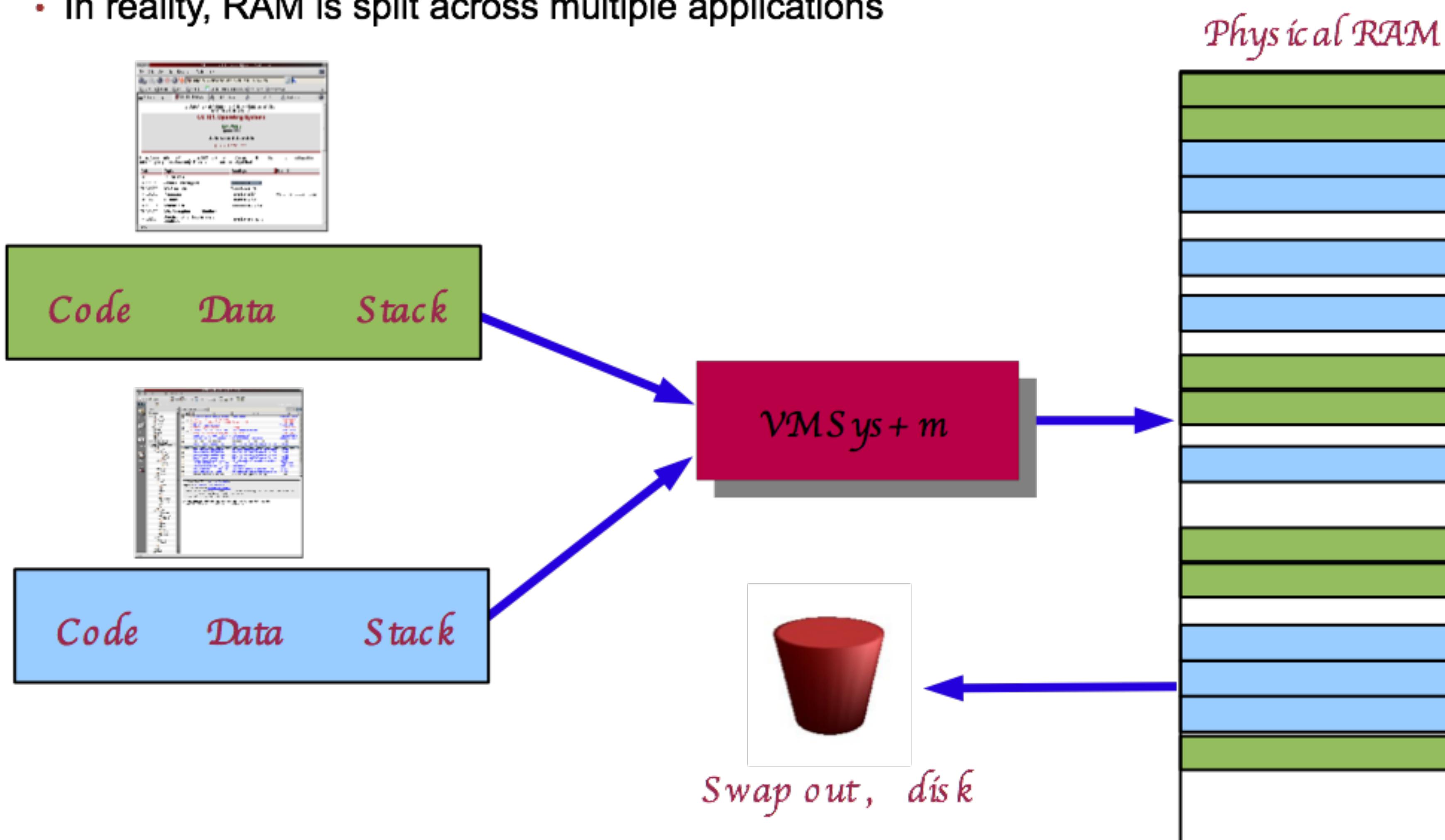
So do I



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

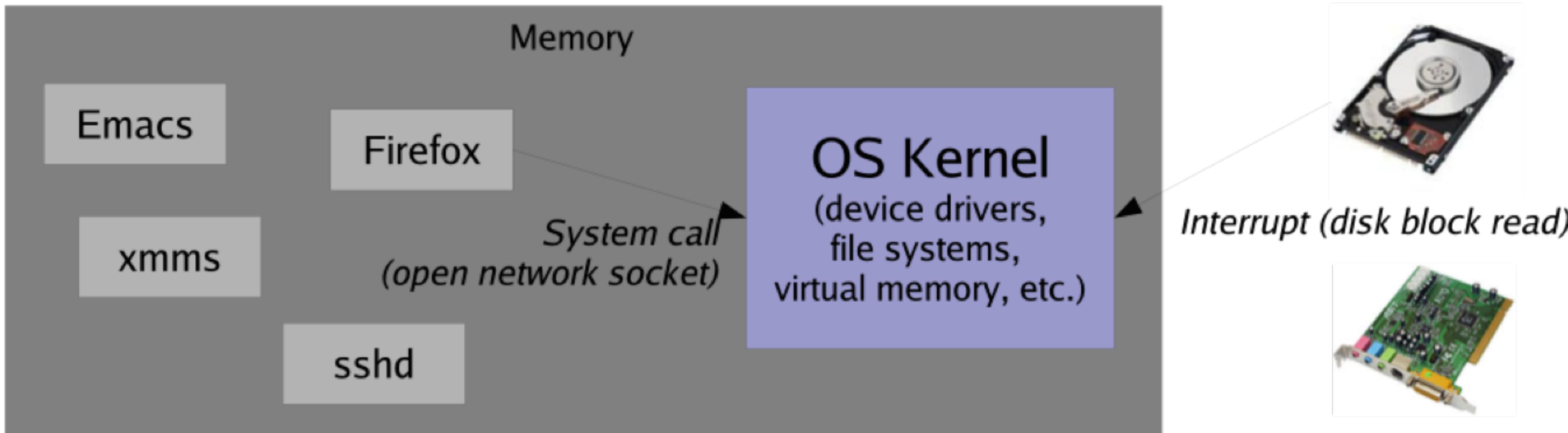
- Prevent multiple apps from interfering with each other and with normal system operation

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



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*

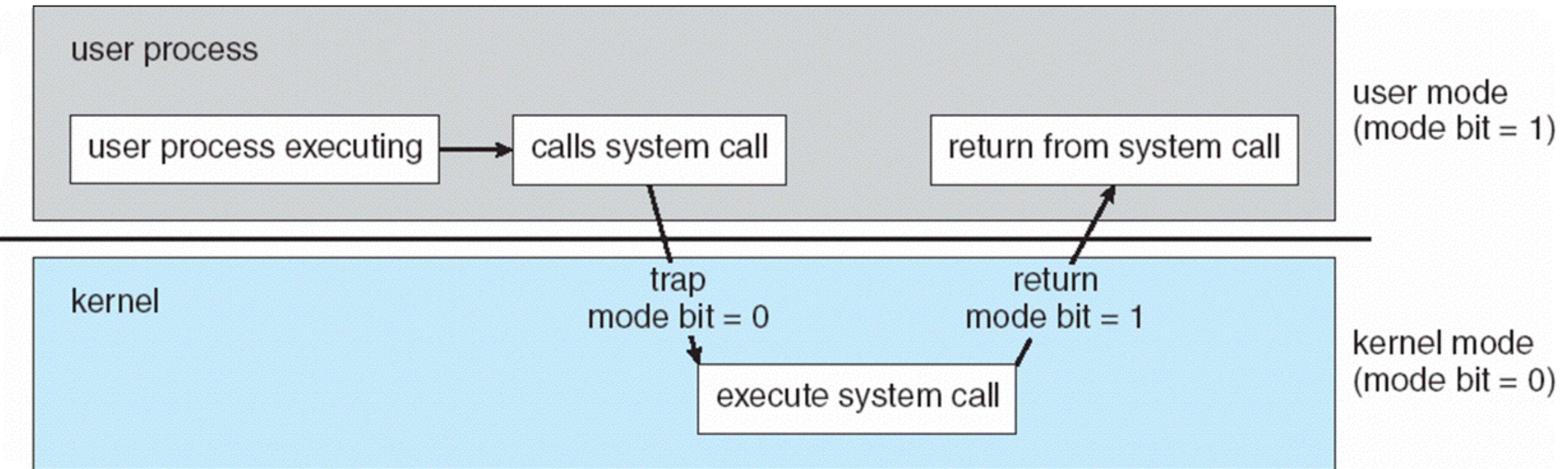
Examples of privileged instructions:

- 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*

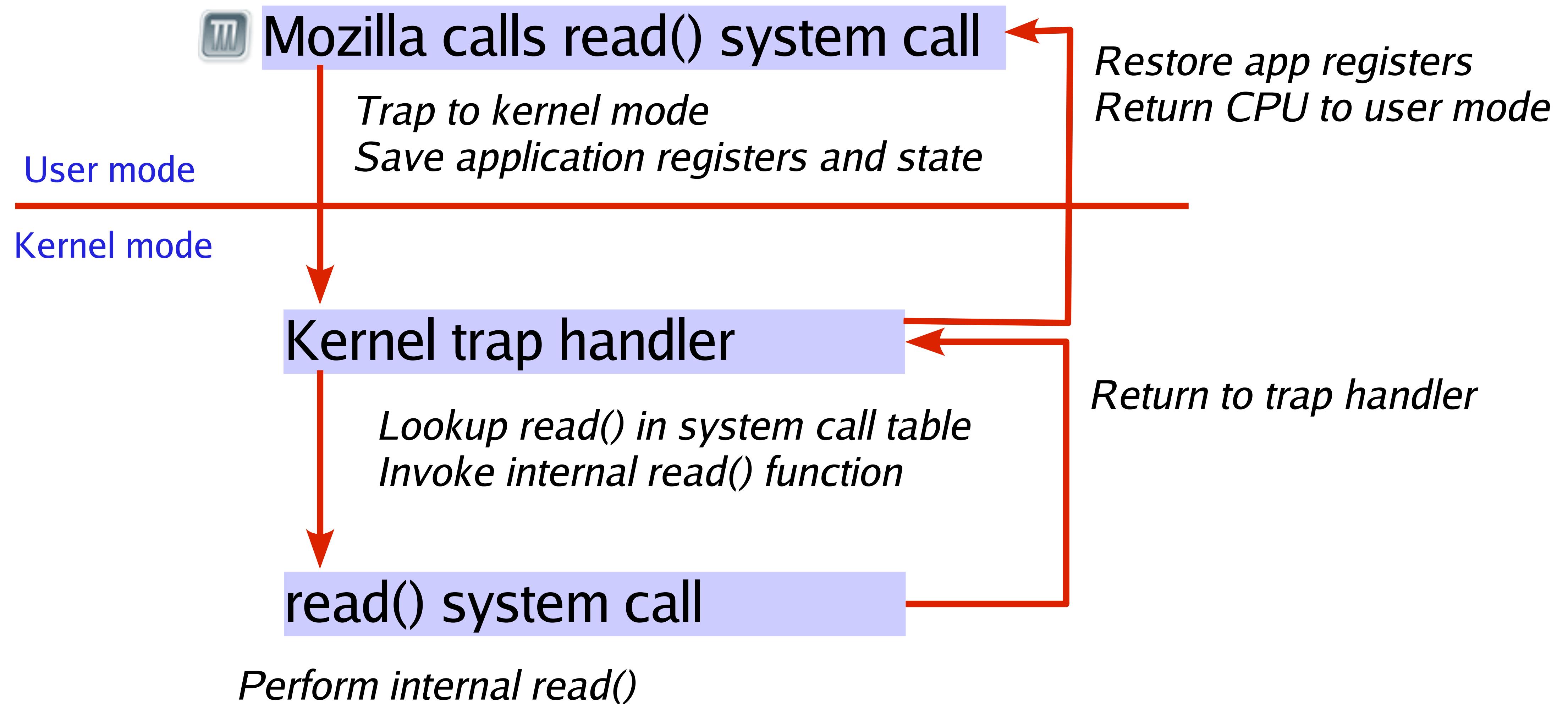
These are enforced by the **CPU hardware itself**.

- 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?

Transition from user to kernel mode



Transition from user to kernel mode: example



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

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.

Process Example

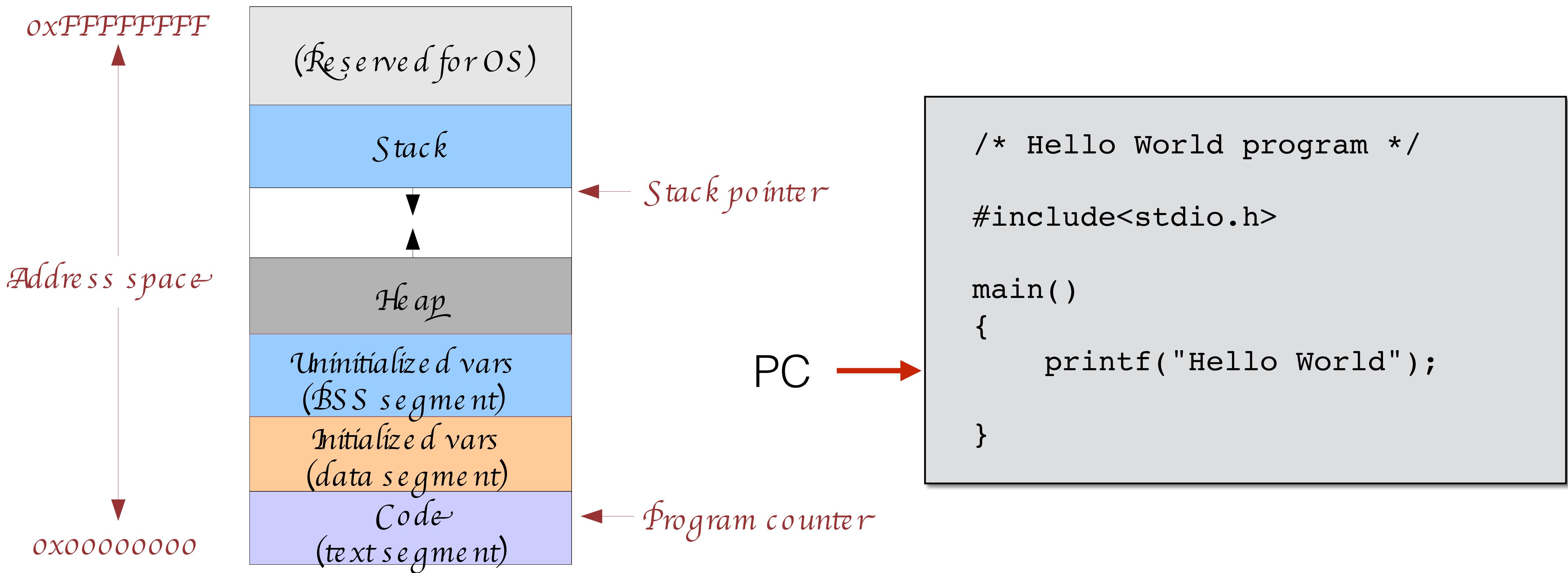
A **process** is an instance of a program being executed

- Use “ps” to list processes on UNIX systems

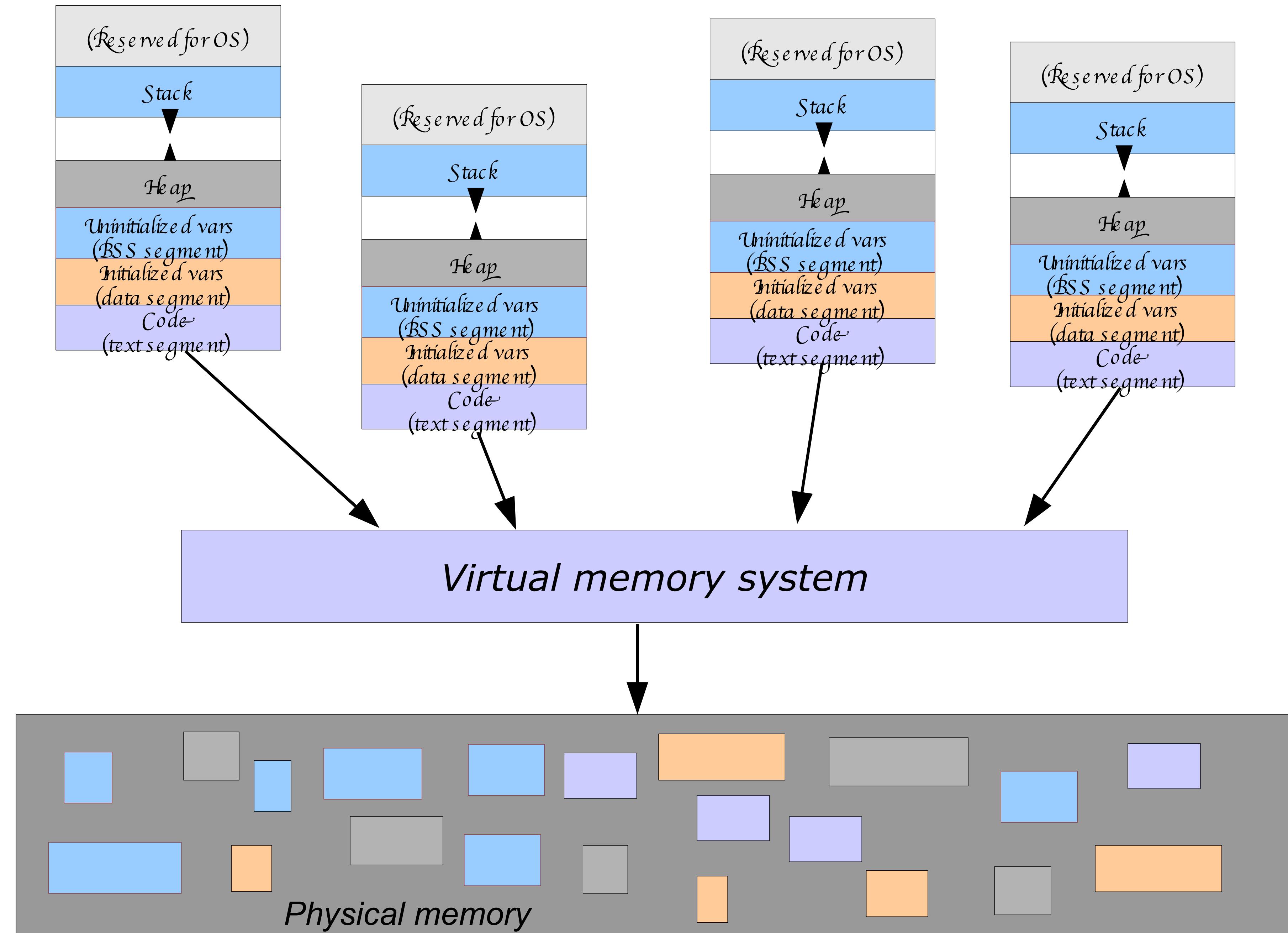
| PID | TTY | STAT | TIME | COMMAND |
|------|-------|------|------|---|
| 842 | tty1 | S | 0:00 | -bash |
| 867 | tty1 | S | 0:00 | xinit |
| 873 | tty1 | S | 0:00 | fvwm2 |
| 887 | tty1 | S | 0:00 | xload |
| 888 | tty1 | S | 0:02 | /usr/local/j2sdk1.4.0/bin/java ApmView 896 243 |
| 1881 | tty1 | S | 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 |

What is a process?

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



Multiple processes



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; Execution 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; Memory mgmt info  
    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; 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; Priority  
    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]; Accounting info  
    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;  
};
```

CPU Virtualization

```
//////////  
// 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

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

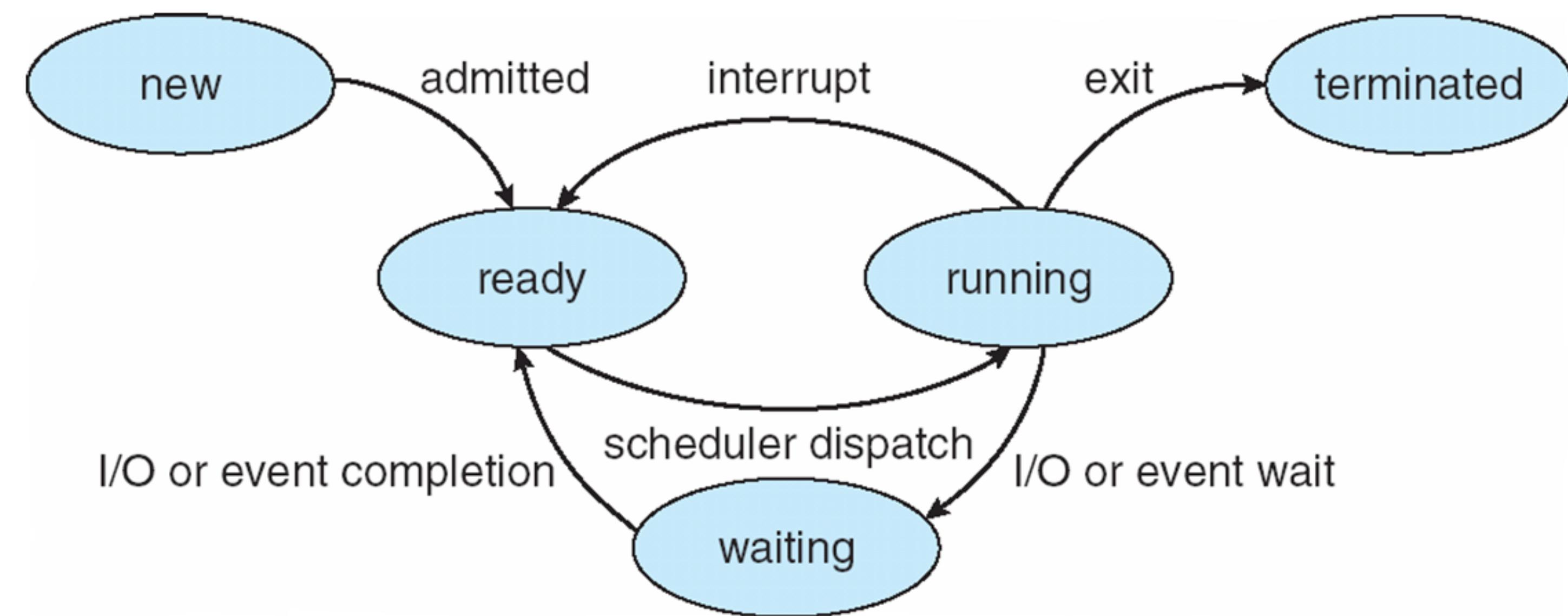
int
main(int argc, char *argv[])
{
    if (argc != 2) {
        fprintf(stderr, "usage: mem <value>\n");
        exit(1);
    }
    int *p;                      // memory for pointer is on "stack"
    p = malloc(sizeof(int));      // malloc'd memory is on "heap"
    assert(p != NULL);
    // printf("(pid:%d) 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;
}
```

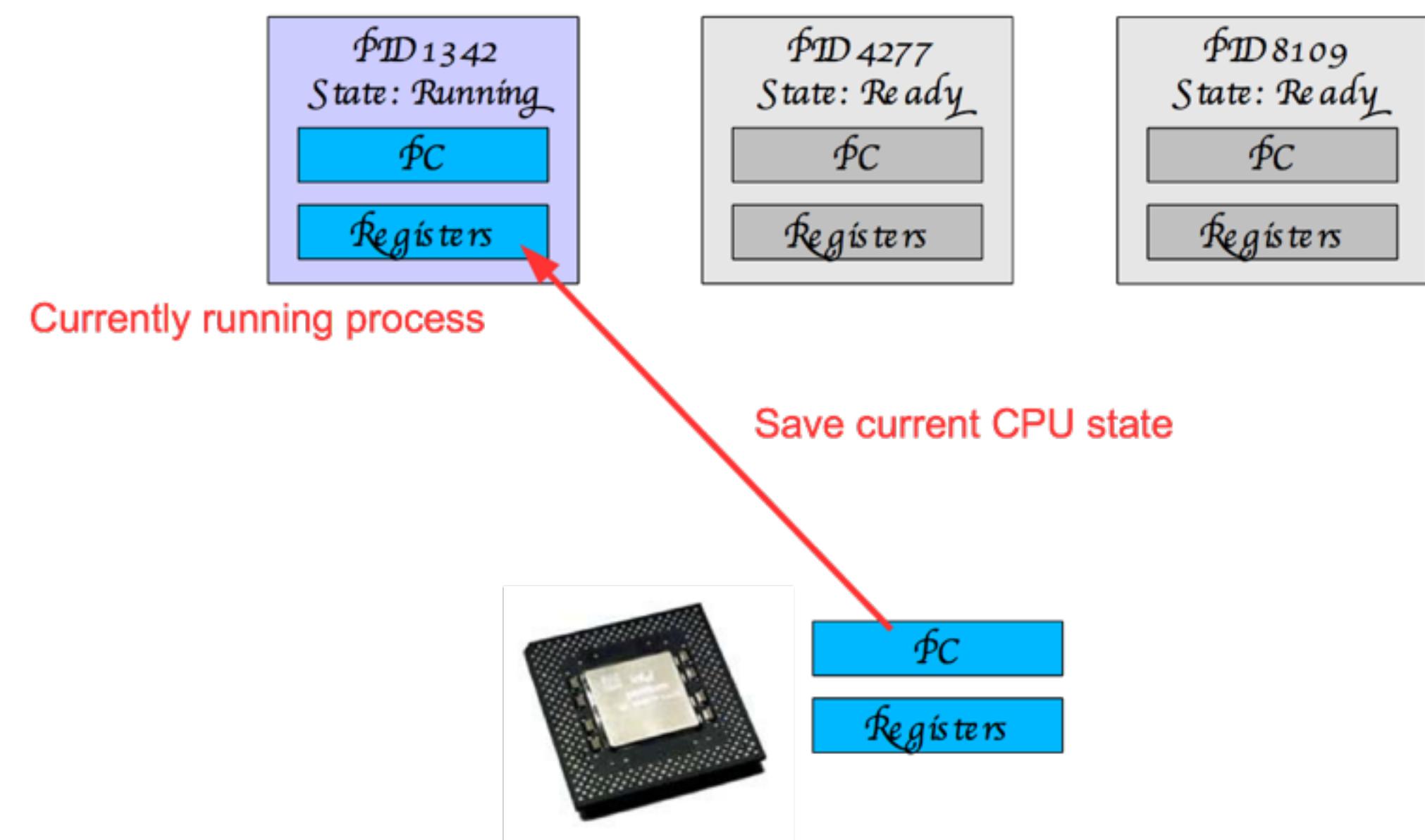
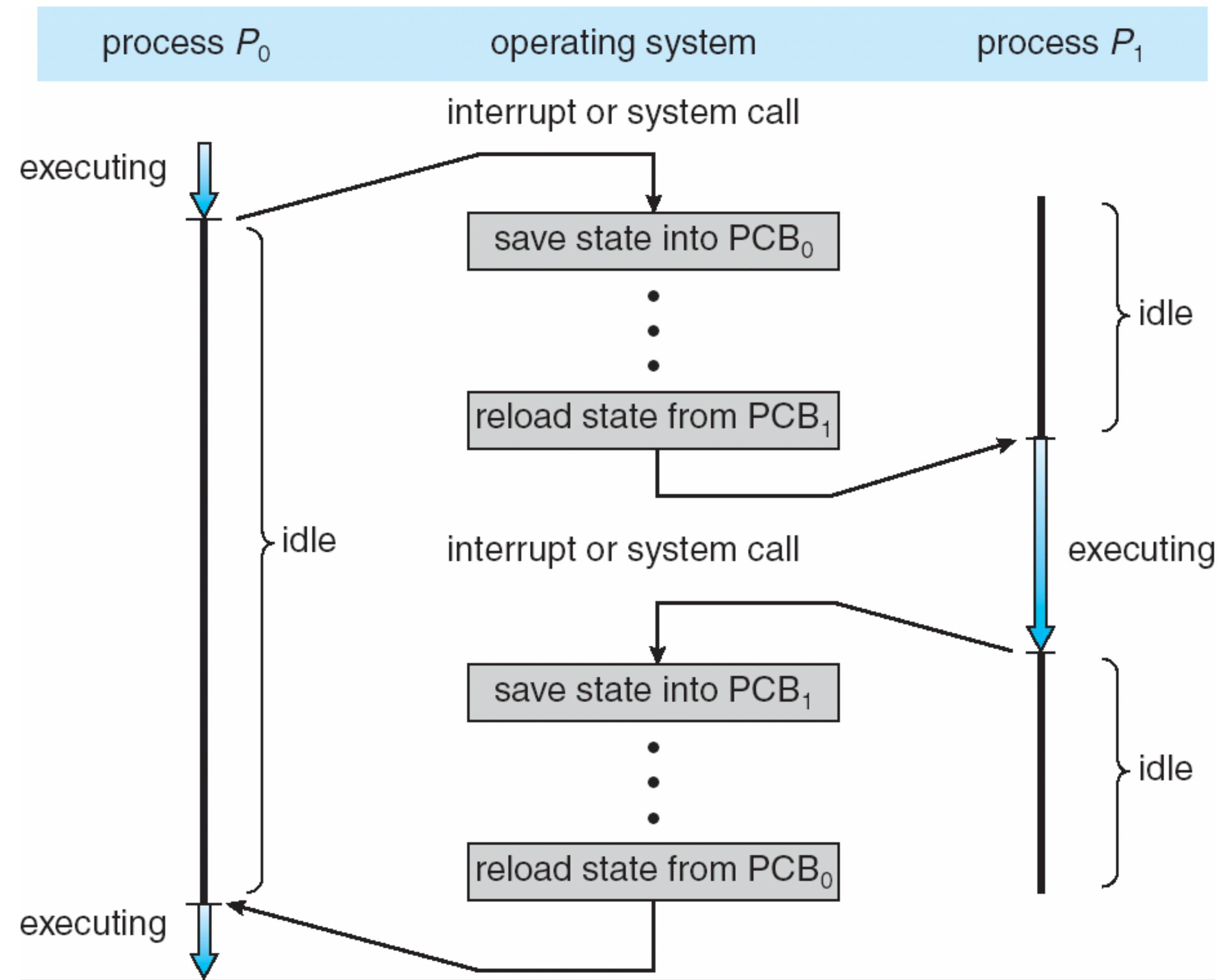
Life cycle of a process

States of a process:

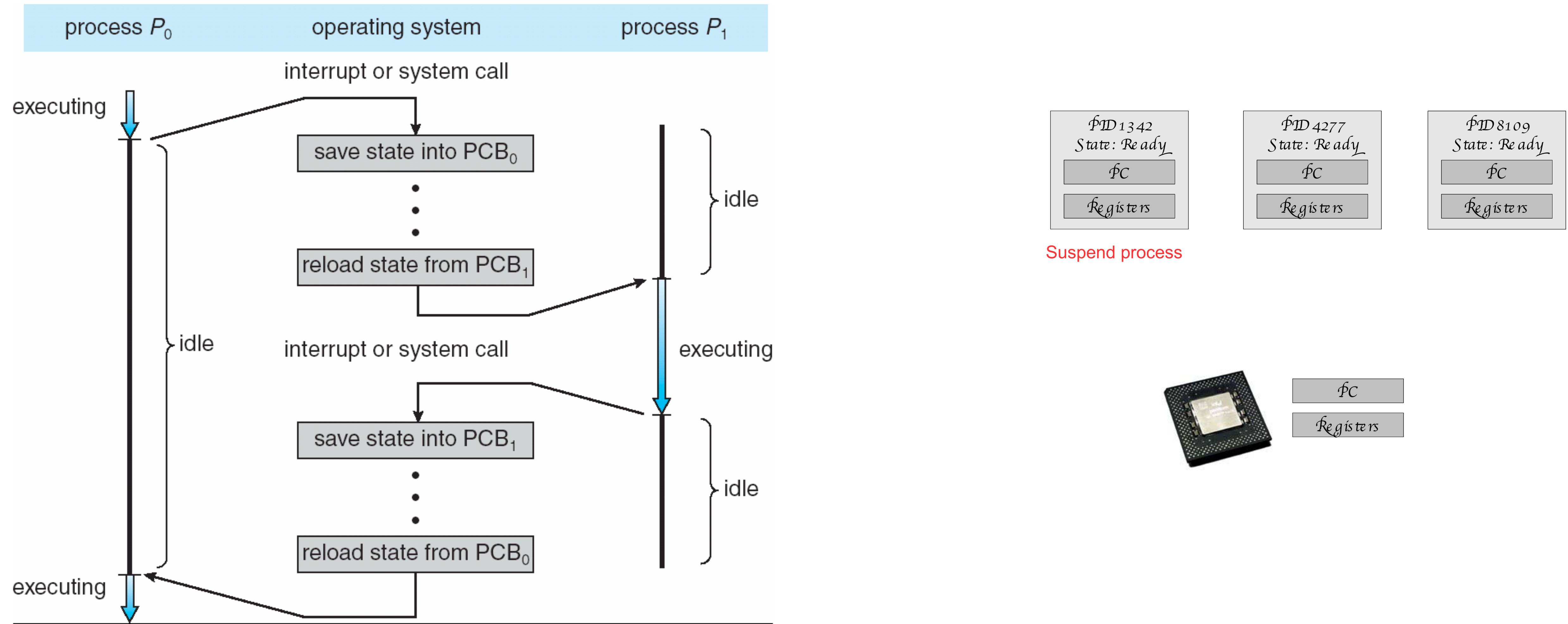
- **new**: The process is being created
- **running**: Instructions are being executed
- **waiting**: The process is waiting for some event to occur
- **ready**: The process is waiting to be assigned to a processor
- **terminated**: The process has finished execution



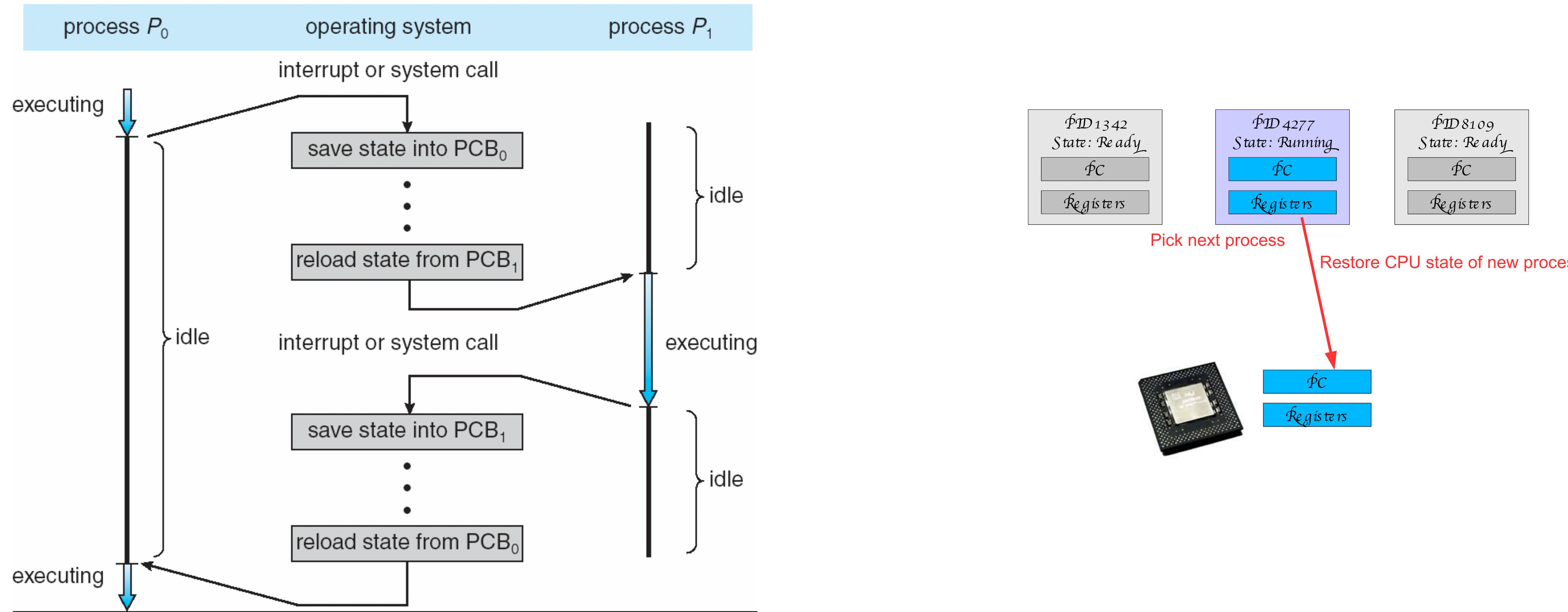
CPU switch from process to process



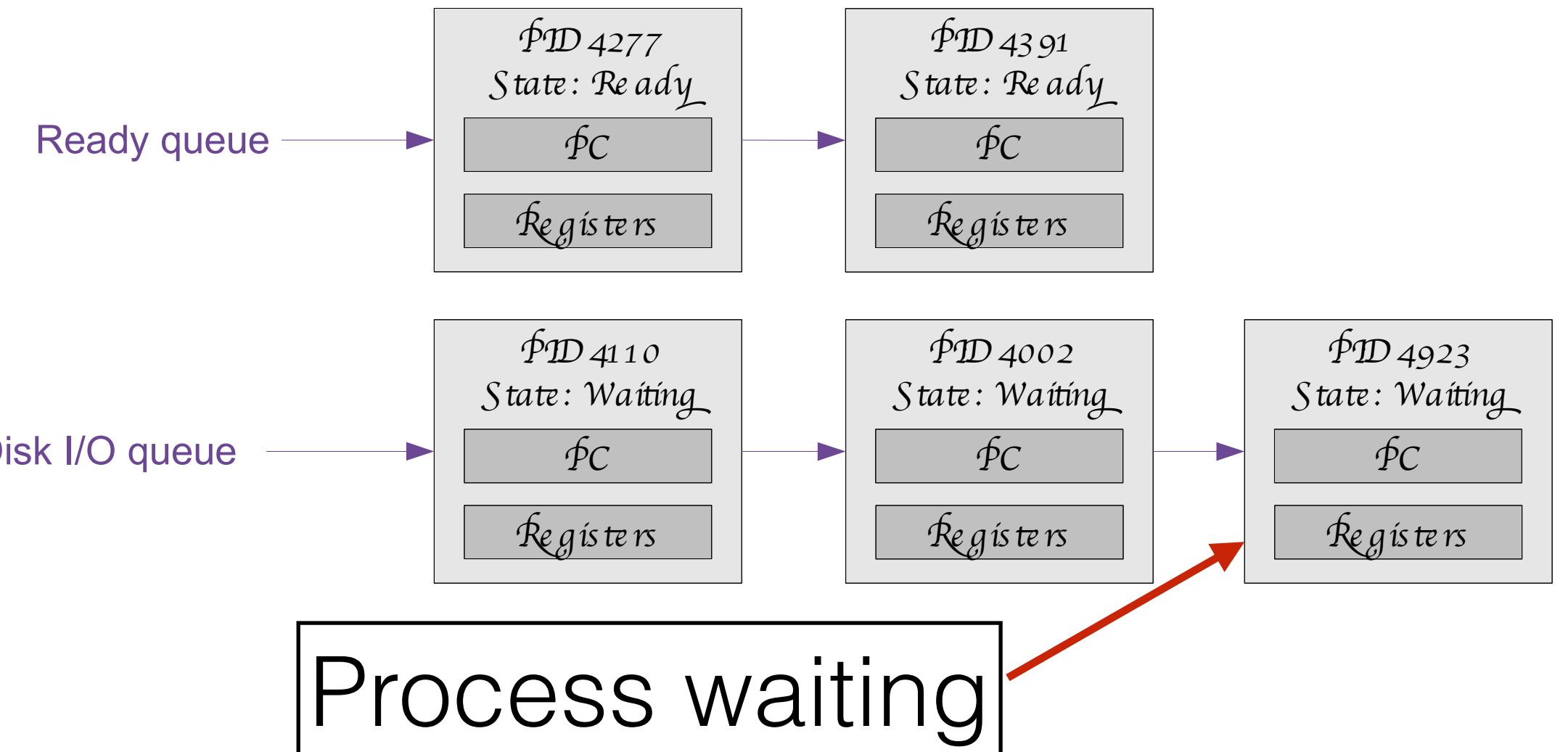
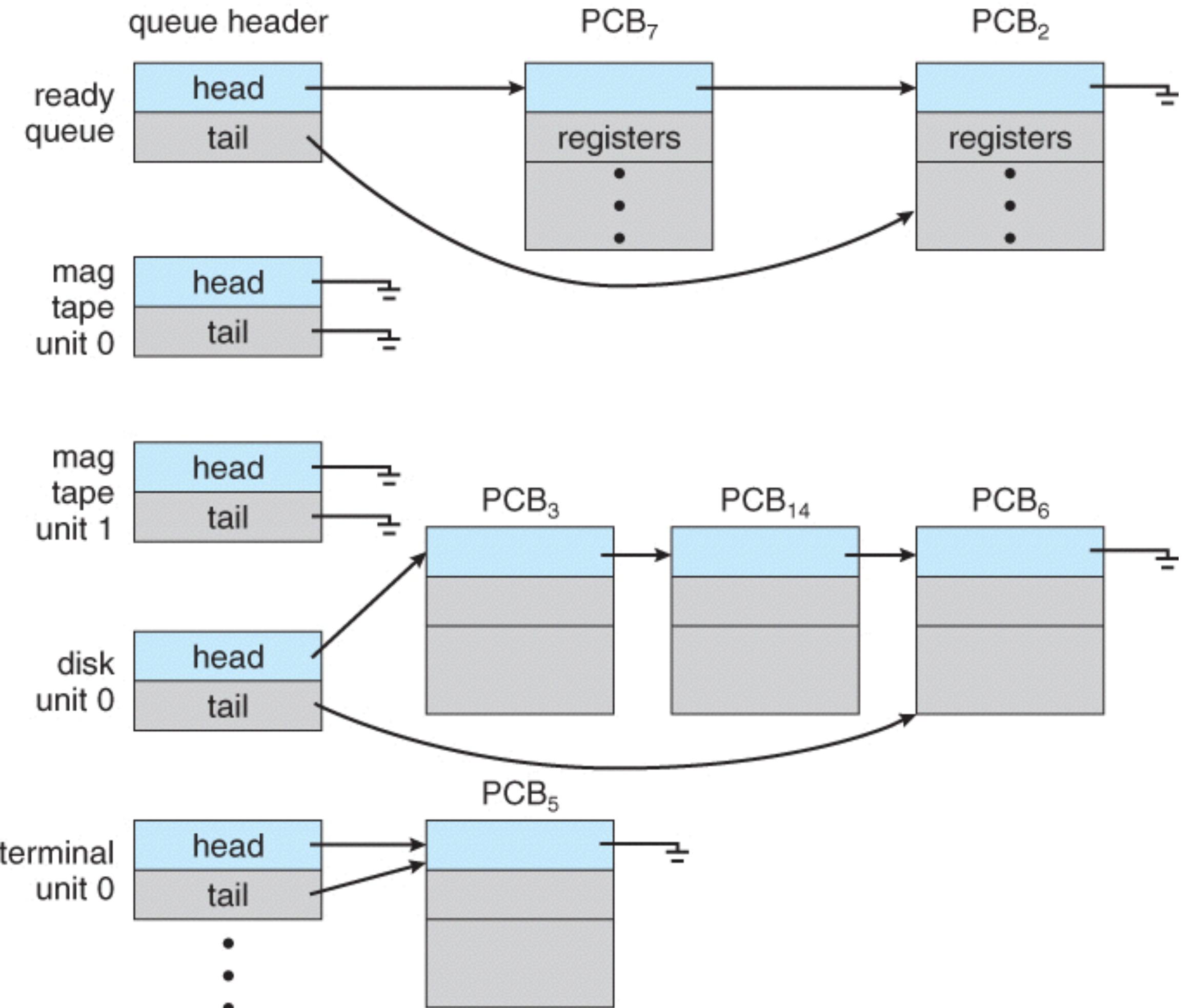
CPU switch from process to process



CPU switch from process to process

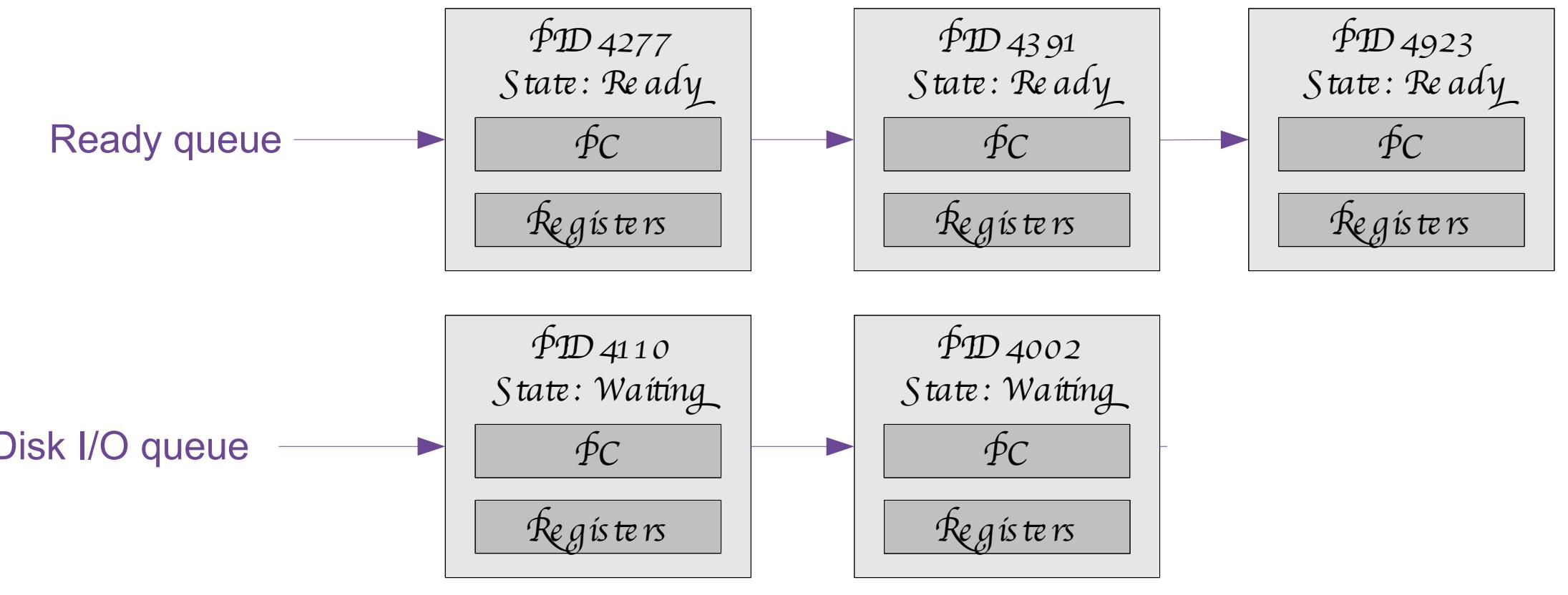
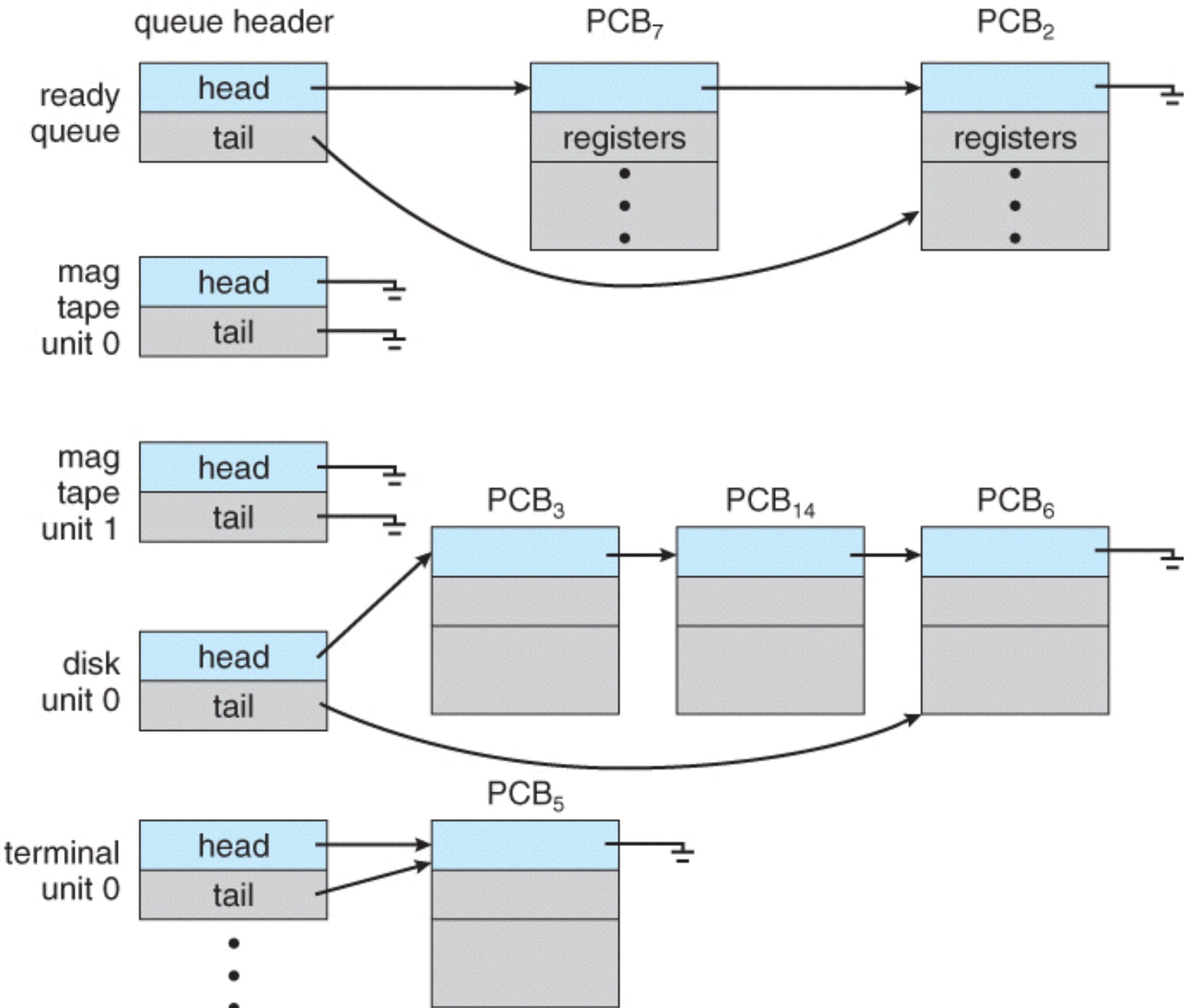


Ready queue and various I/O queues



- 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.

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.

System Calls

CSE 4001 Operating Systems Concepts

E. Ribeiro

January 24, 2022

Outline

- 1 What is a process?
- 2 Limited Direct Execution
- 3 OS/161 Examples

What is a process?

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

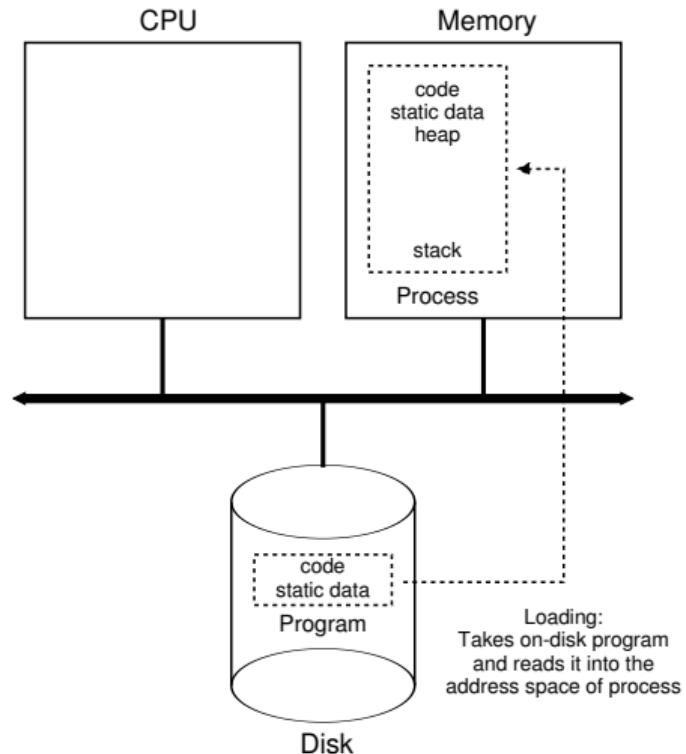


Figure from: OS in three easy pieces

Limited Direct Execution

Main question:

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

Life cycle of a process

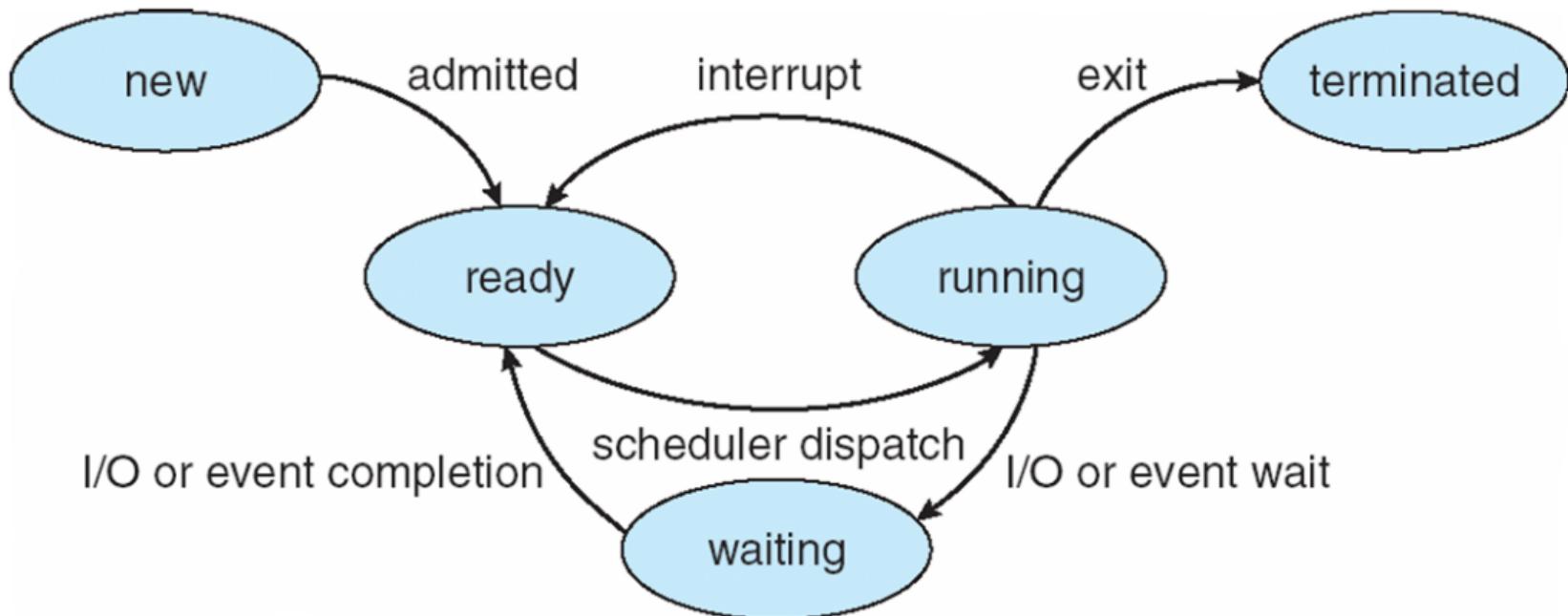


Figure adapted from Silberschatz, Galvin, and Gagne, 2009.

Limited Direct Execution

Main question:

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

Two Approaches:

- Cooperative processes
- Non-cooperative processes

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.



Slide text adapted from original slide by Larry Zhang.

Approach 1: Cooperative processes

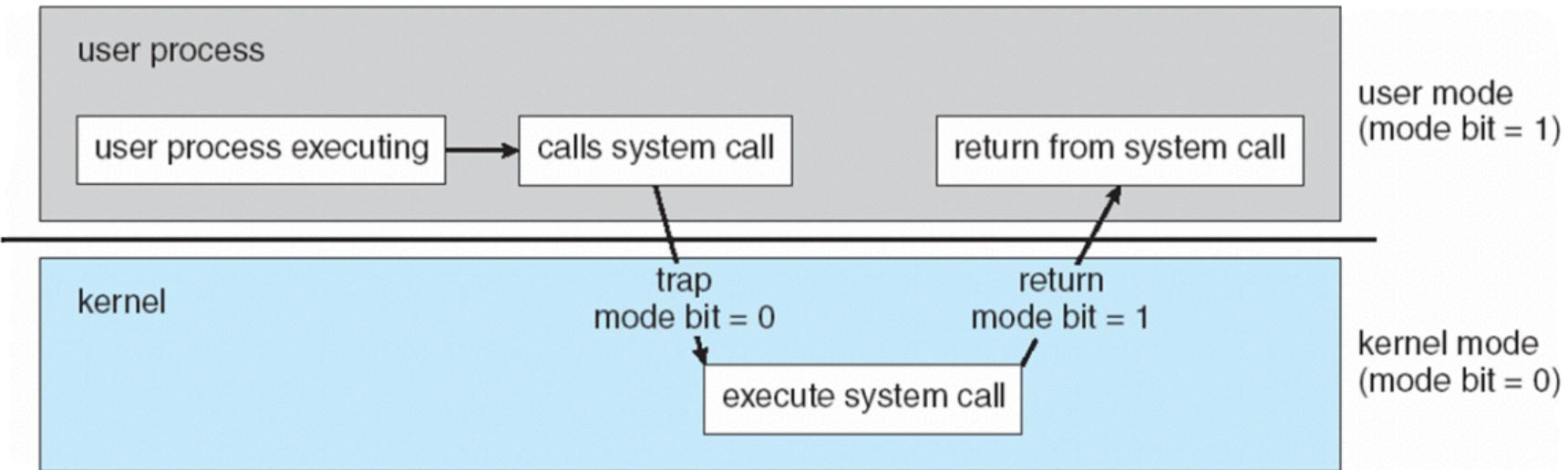


Figure adapted from Silberschatz, Galvin, and Gagne, 2009.

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.



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

Context-switch steps:

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

In the next slides, let's see two examples of context switch in OS/161, one caused by the timer and the other caused by a trap (or exception).

OS/161 Examples: Context switch triggered by timer.

function hardclock in /kern/thread/clock.c

```
/*
 * This is called HZ times a second (on each processor) by the timer
 * code.
 */
void
hardclock(void)
{
    /*
     * Collect statistics here as desired.
     */

    curcpu->c_hardclocks++;
    if ((curcpu->c_hardclocks % MIGRATE_HARDCLOCKS) == 0) {
        thread_consider_migration();
    }
    if ((curcpu->c_hardclocks % SCHEDULE_HARDCLOCKS) == 0) {
        schedule();
    }
    thread_yield();
}
```

OS/161 Examples: Context switch triggered by timer.

function thread_yield in /kern/thread/thread.c

```
/*
 * Yield the cpu to another process, but stay runnable.
 */
void
thread_yield(void)
{
    thread_switch(S_READY, NULL, NULL);
}
```

OS/161 Examples: Context switch triggered by timer.

function thread_switch in /kern/thread/thread.c calls low-level context switcher in assembler in /kern/arch/mips/thread/switch.S

```
659         */
660         curcpu->c_curthread = next;
661         curthread = next;
662
663         /* do the switch (in assembler in switch.S) */
664         switchframe_switch(&cur->t_context, &next->t_context);
665
666         /*
667         * If we're switching to a thread that has a
668         * pending interrupt, then we need to
669         * clear it before we switch contexts.
670         *
671         * This is done in the assembly code in
672         * switch.S.
673         */
674     }
675 }
```

OS/161 Examples: Context switch triggered by timer.

[https://github.com/eribeiroClassroom/
os161-Kernel-Src-Add-System-Call-Assignment/blob/master/kern/arch/mips/
thread/switch.S](https://github.com/eribeiroClassroom/os161-Kernel-Src-Add-System-Call-Assignment/blob/master/kern/arch/mips/thread/switch.S)

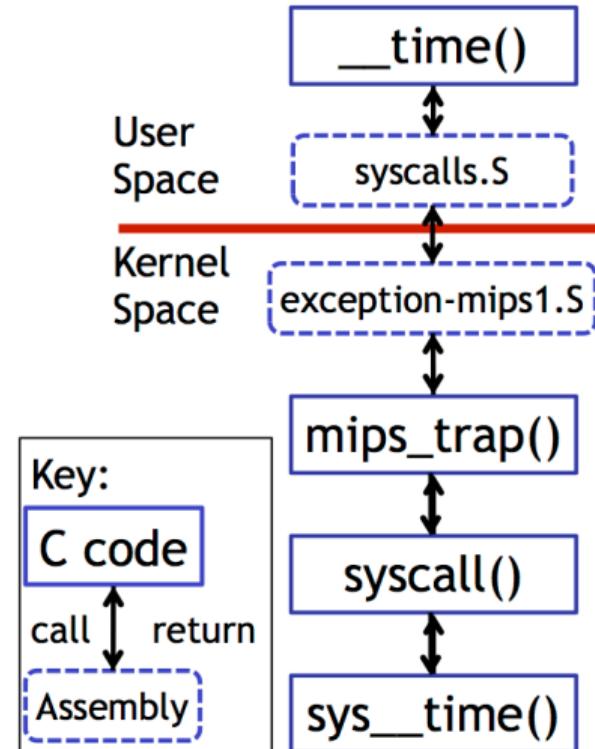
OS/161 Examples: Context switch triggered by an exception or trap.

General exception occurs which causes the hardware to call:

[https://github.com/eribeiroClassroom/
os161-Kernel-Src-Add-System-Call-Assignment/blob/master/kern/arch/mips/
locore/exception-mips1.S](https://github.com/eribeiroClassroom/os161-Kernel-Src-Add-System-Call-Assignment/blob/master/kern/arch/mips/locore/exception-mips1.S)

exception-mips1.S creates and fills in the trapframe and then calls `mips_trap()`. This C-language function is a general trap (exception) handling function.

OS/161 Examples: Context switch triggered by an exception or trap.



Adding System Calls to OS/161

CSE 4001 Operating Systems Concepts

E. Ribeiro

January 26, 2022

Outline

- 1 Review: traps and system calls
- 2 Overview of steps to add system calls to OS/161
 - Kernel-level steps
 - User-level steps
 - Testing the system call
- 3 Kernel-level steps in detail
- 4 User-level steps in detail
- 5 Testing steps in detail

Review: System-call trapping mechanism

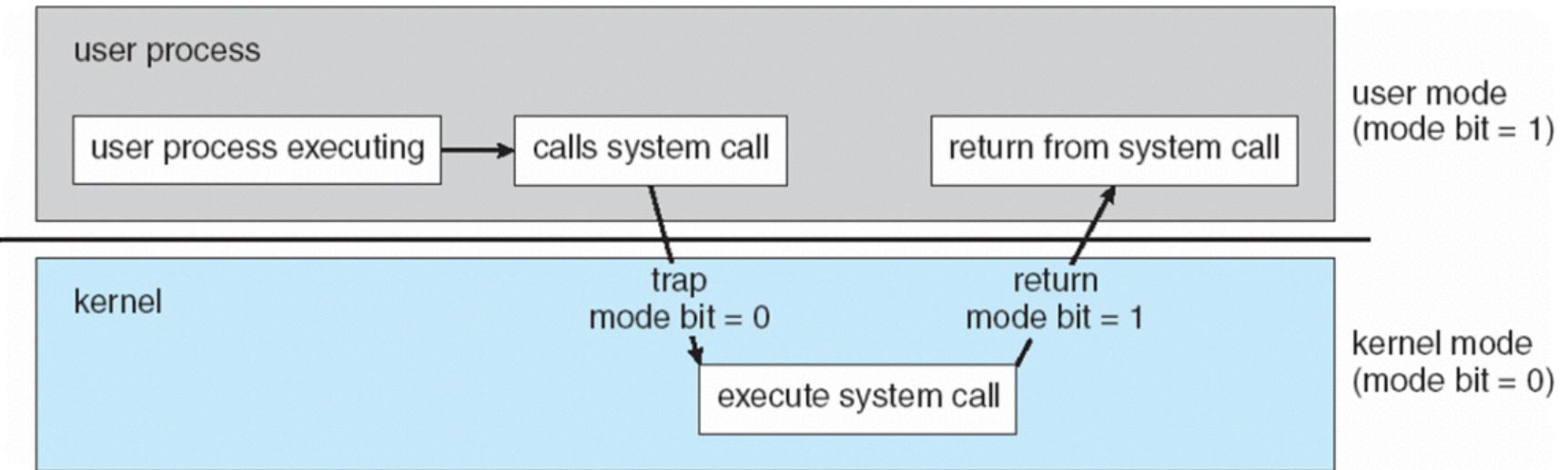
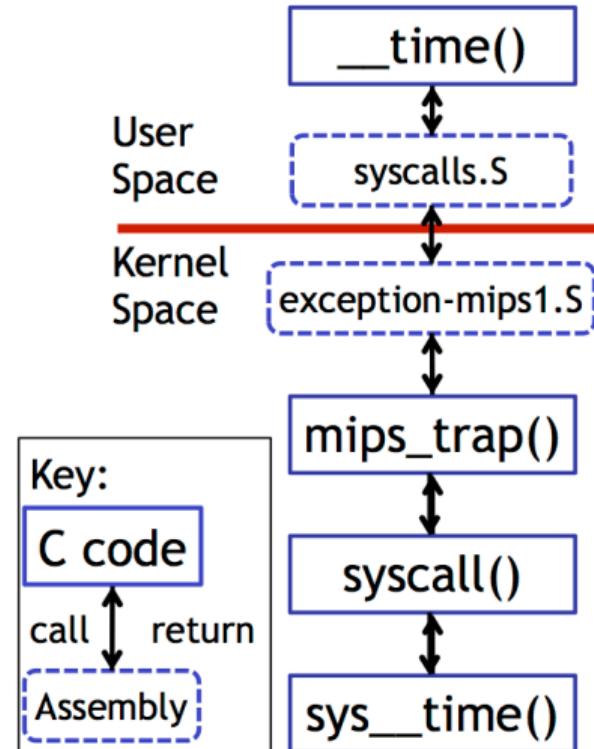


Figure adapted from Silberschatz, Galvin, and Gagne, 2009.

Review: System-call trapping mechanism in OS/161



Kernel-level steps

- ① Add the prototype of the system-call function to the header file:
`kern/include/syscall.h`
- ② The kernel-level implementation (e.g., `newsySCALL.c`) goes into `kern/syscall/`
- ③ Add a new ID number for the system call. The new entry goes in the file
`kern/include/kern/syscall.h`
- ④ Add a new branch in the switch-case statement in:
`kern/arch/mips/syscall/syscall.c`
- ⑤ Add file entry definition for `syscall/newsySCALL.c` in `kern/conf/conf.kern`

User-level steps

- ① Add the user-level prototype of the system call to: `user/include/unistd.h`
- ② Add the user-level test function. For this, create a new subdirectory directory `user/testbin/testnewsyscall/` and inside it add the test function (e.g., `testnewsyscall.c`).
- ③ Create a Makefile inside this subdirectory for building the test function. You can use one of the subdirectories as a template.
- ④ Add an entry to the new function to the top-level Makefile in `user/testbin`

Testing the new system call

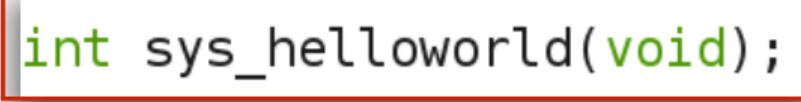
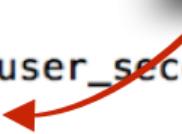
- ① Re-build the kernel
- ② Start the new kernel (i.e., run sys161 kernel in the root directory)
- ③ At the OS161 prompt, use the p option (from OS161 menu) to run the test program, i.e.,
`p testbin/testnewsyscall`

Kernel-level steps

1 Prototype of the system call

- ① Add the prototype of the system call to the header file: kern/include/syscall.h
- ② At the end of the file, you will find prototypes for sys_reboot() and sys_time().

```
53
54 /*
55  * Prototypes for IN-KERNEL entry points for system call
56  * implementations.
57 */
58 int sys_reboot(int code);
59 int sys_time(userptr_t user_seconds, userptr_t user_nanoseconds);
60
61 #endif /* _SYSCALL_H_ */
```



2 Kernel-level implementation

- ➊ The kernel-level implementation goes into kern/syscall. This directory contains an example of a system call, i.e., time_syscalls.c.
- ➋ Here, create a program called simple_syscall.c, and implement your system call in it.

```
int sys_helloworld(void){  
    return kprintf("Hello World!\n");  
}
```

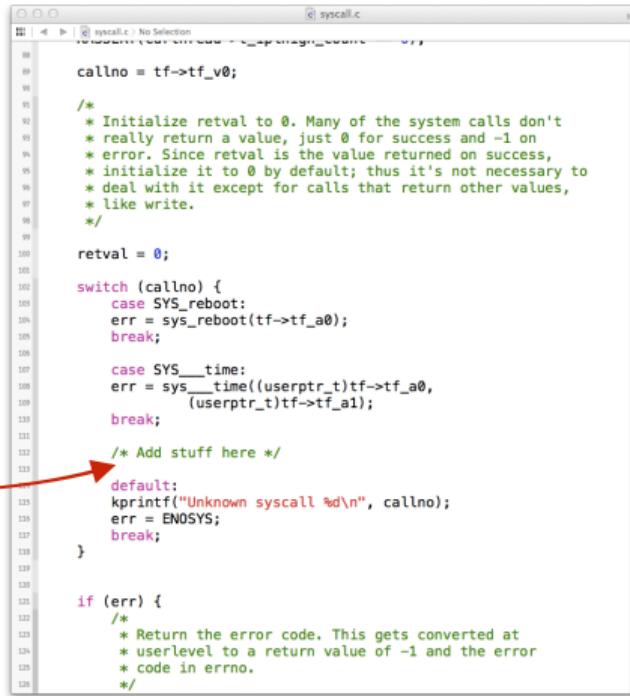
3 Create the ID number for the new system call

- ① The OS needs to know the ID number of the system call
- ② Add a new entry to the file kern/include/kern/syscall.h

```
98  ##define SYS_setpgid    41
99  ##define SYS_getsid    42
100 ##define SYS_setsid    43
101 //                                     (userlevel debugging)
102 ##define SYS_ptrace     44
103
104 //                                     -- File-handle-related --
105
106
107 #define SYS_open         45
108 #define SYS_pipe        46
109 #define SYS_dup         47
110 #define SYS_dup2        48
```

4 Add a new branch in the switch-case statement in: kern/arch/mips/syscall/syscall.c

```
case SYS_helloworld:  
    err = sys_helloworld();  
    break;
```



```
callno = tf->tf_v0;  
  
/*  
 * Initialize retval to 0. Many of the system calls don't  
 * really return a value, just 0 for success and -1 on  
 * error. Since retval is the value returned on success,  
 * initialize it to 0 by default; thus it's not necessary to  
 * deal with it except for calls that return other values,  
 * like write.  
 */  
  
retval = 0;  
  
switch (callno) {  
    case SYS_reboot:  
        err = sys_reboot(tf->tf_a0);  
        break;  
  
    case SYS_time:  
        err = sys_time((userptr_t)tf->tf_a0,  
                      (userptr_t)tf->tf_a1);  
        break;  
  
    /* Add stuff here */  
    default:  
        kprintf("Unknown syscall %d\n", callno);  
        err = ENOSYS;  
        break;  
}  
  
if (err) {  
    /*  
     * Return the error code. This gets converted at  
     * userlevel to a return value of -1 and the error  
     * code in errno.  
     */  
}
```

Note how user-level input parameters are passed to kernel-level functions via the trapframe.

5 Add file-entry definition to config.kern

```
358
359     file      vfs/devnull.c
360
361     #
362     # System call layer
363     # (You will probably want to add stuff here while doing the basic system
364     # calls assignment.)
365     #
366
367     file      syscall/loadelf.c
368     file      syscall/runprogram.c
369     file      syscall/time_syscalls.c
370
371     #
372     # Startup and initialization
373     #
374
375     file      startup/main.c
376     file      startup/menu.c
377
378 ######
379     #                         #
380     #             Filesystems          #
381     #                         #
382 ######
```

User-level steps

1. Add the user-level prototype of the system call to: userland/include/unistd.h

The screenshot shows a code editor window titled "unistd.h" with the file content displayed. A red arrow points from the text "/* Required. */" at line 137 to the line "#define __DEAD". A red box highlights the function prototypes "int helloworld();" and "int printchar(char c);".

```
* This file is *not* shared with the kernel, even though in a sense
* the kernel needs to know about these prototypes. This is because,
* due to error handling concerns, the in-kernel versions of these
* functions will usually have slightly different signatures.
*/
#ifndef __GNUC__
/* GCC gets into a snit if _exit isn't declared to not return */
#define __DEAD __attribute__((__noreturn__))
#else
#define __DEAD
#endif

/* Required. */
__DEAD void _exit(int code);
int execv(const char *prog, char *const *args);
pid_t fork(void);
int waitpid(pid_t pid, int *returncode, int flags);
/*
 * Open actually takes either two or three args: the optional third
 * arg is the file security and p
 */
int open(const ch
int read(int file
int write(int filehandle, const void *buf, size_t size);
int close(int filehandle);
int reboot(int code);
int sync(void);
/* mkdir - see sys/stat.h */
int rmdir(const char *dirname);

/* Recommended. */
int getpid(void);
int ioctl(int filehandle, int code, void *buf);
```

2. Add the user-level test function.

For this, create a new subdirectory directory `user/testbin/testnewsyscall/` and inside it add the test function (e.g., `testnewsyscall.c`).

```
.c helloworldtest.c ✘
1 #include <unistd.h>
2
3 int
4 main()
5{
6    helloworld();
7    return 0;
8}
```

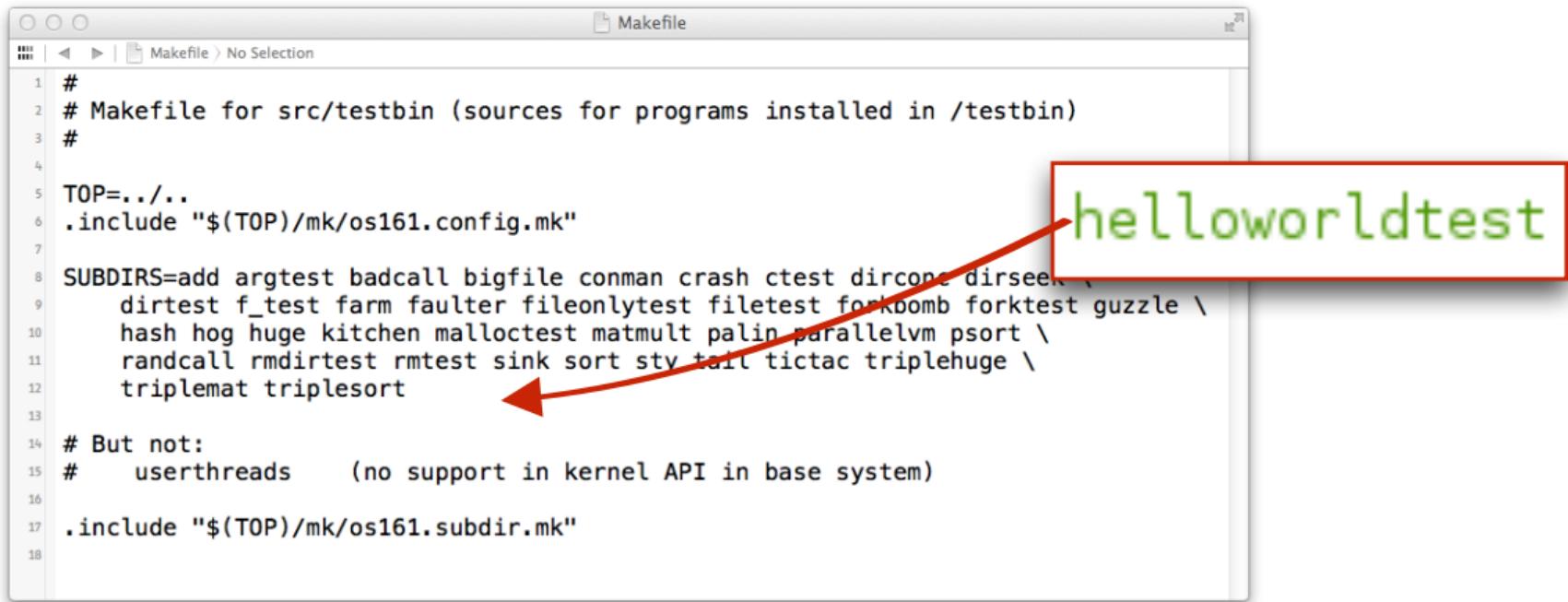
```
# Makefile for helloworldtest
TOP=../../..
.include "$TOP/mk/os161.config.mk"

PROG=helloworldtest
SRCS=helloworldtest.c
BINDIR=/testbin

.include "$TOP/mk/os161.prog.mk"
```

3. Modify the top-level makefile.

Add an entry to the new function to the top-level Makefile in user/testbin/

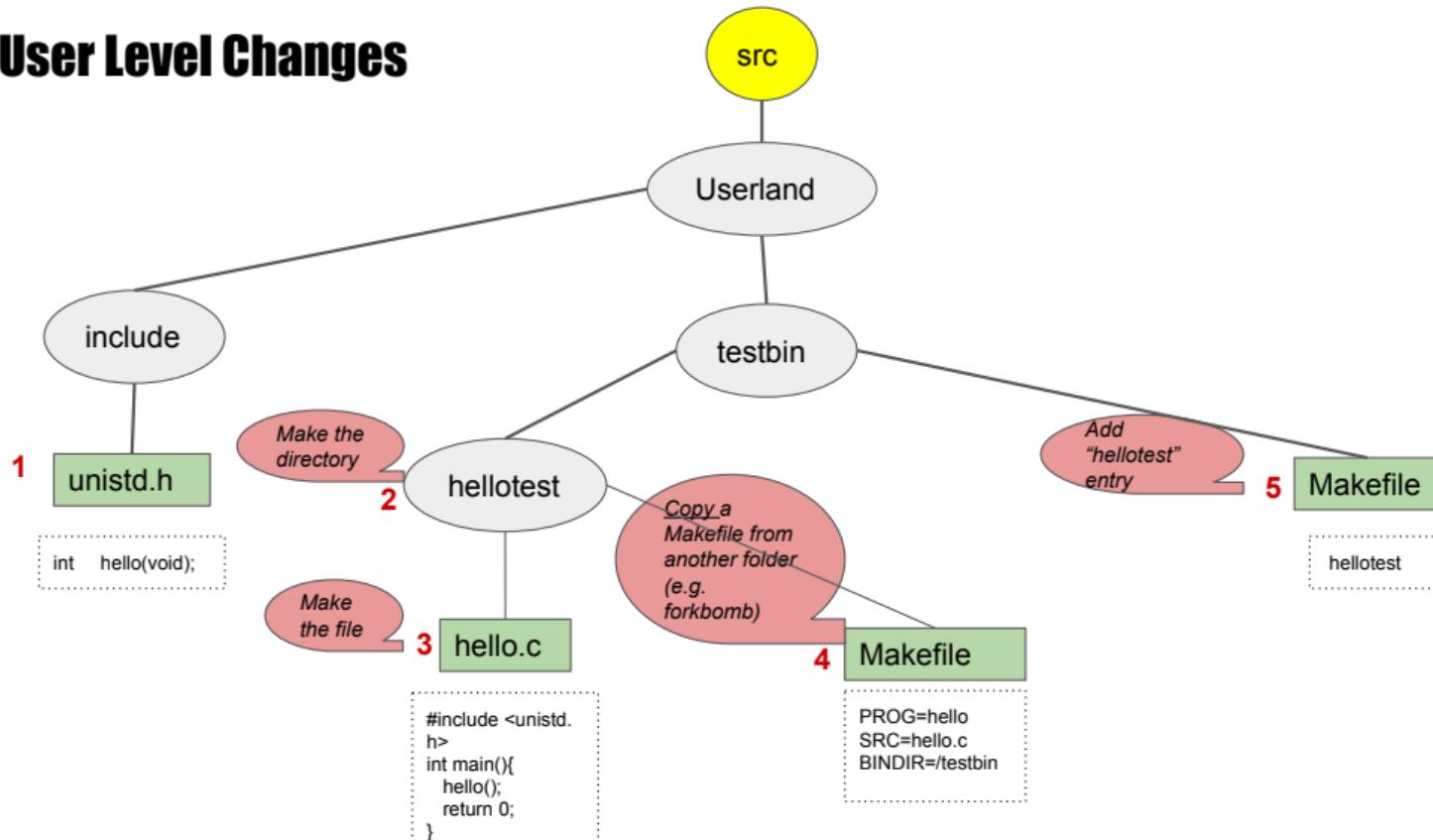


```
#  
# Makefile for src/testbin (sources for programs installed in /testbin)  
#  
TOP=../../  
.include "$(TOP)/mk/os161.config.mk"  
  
SUBDIRS=add argtest badcall bigfile conman crash ctest dircone dirseek  
dirtest f_test farm faulter fileonlytest filetest forkbomb forktest Guzzle \  
hash hog huge kitchen malloctest matmult palin parallelvml psort \  
randcall rmdirtest rmtest sink sort stv tail tictac triplehuge \  
triplemat triplesort  
  
# But not:  
#     userthreads      (no support in kernel API in base system)  
  
.include "$(TOP)/mk/os161.subdir.mk"
```

helloworldtest

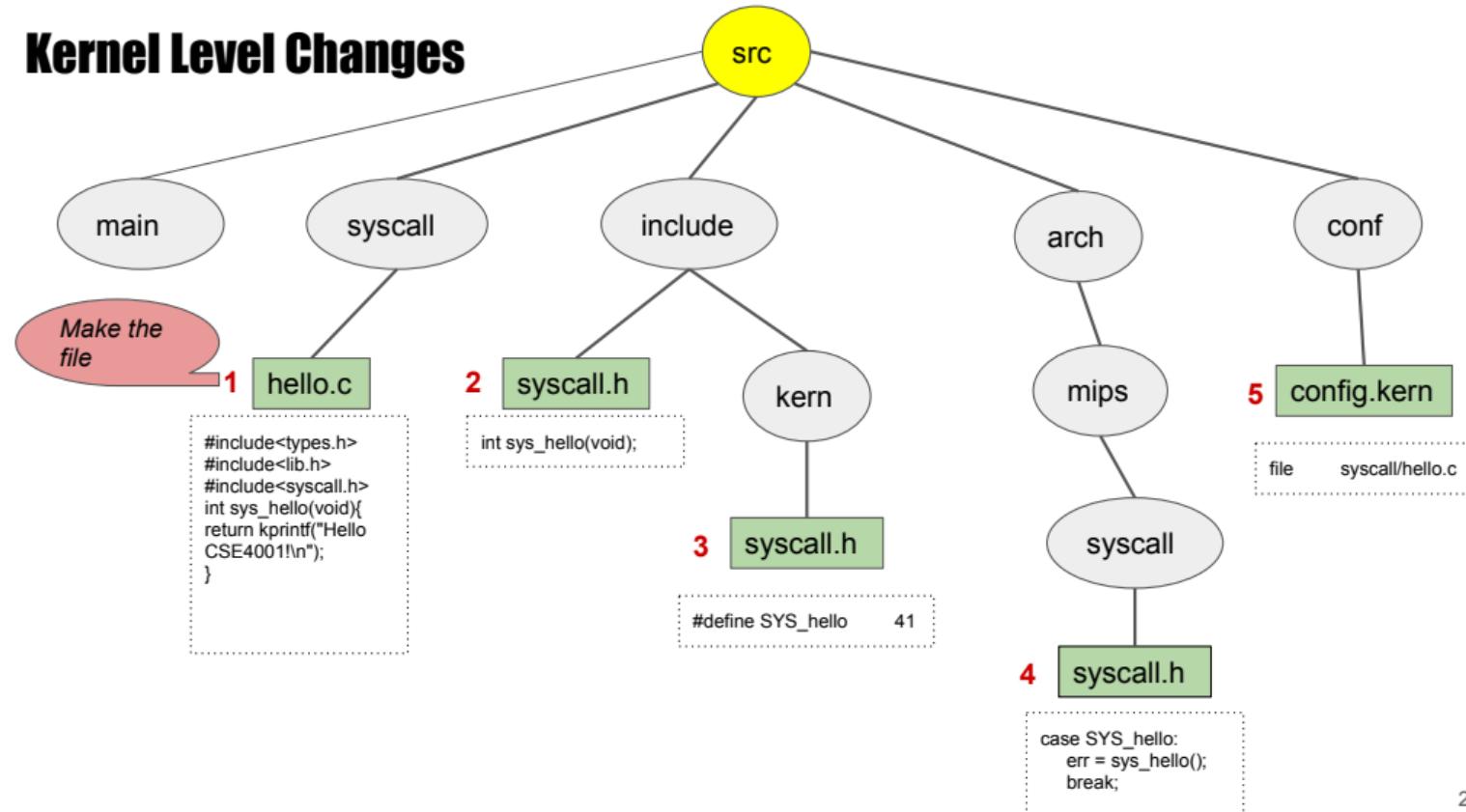
Directory tree showing main changes that need to be made

User Level Changes



Directory tree showing main changes that need to be made

Kernel Level Changes



Testing the system call

Testing the system call

- ① Inside the root folder, run the command sys161 kernel.
- ② In the os161 terminal, run the command p testbin/[name] where your [name] is the name of your program.

Helloworld Program:

```
OS/161 kernel [? for menu]: p testbin/helloworld
Operation took 0.000145920 seconds
OS/161 kernel [? for menu]: syscall: #40, args 0 0 0 0
Hello World!
syscall: #3, args 0 0 0 0
Thread testbin/helloworld exiting due to 0 with value 0
```

Processes

CSE 4001 Operating Systems Concepts

E. Ribeiro

January 26, 2022

Outline

1 Processes

A process is a program in execution

What is a process?

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

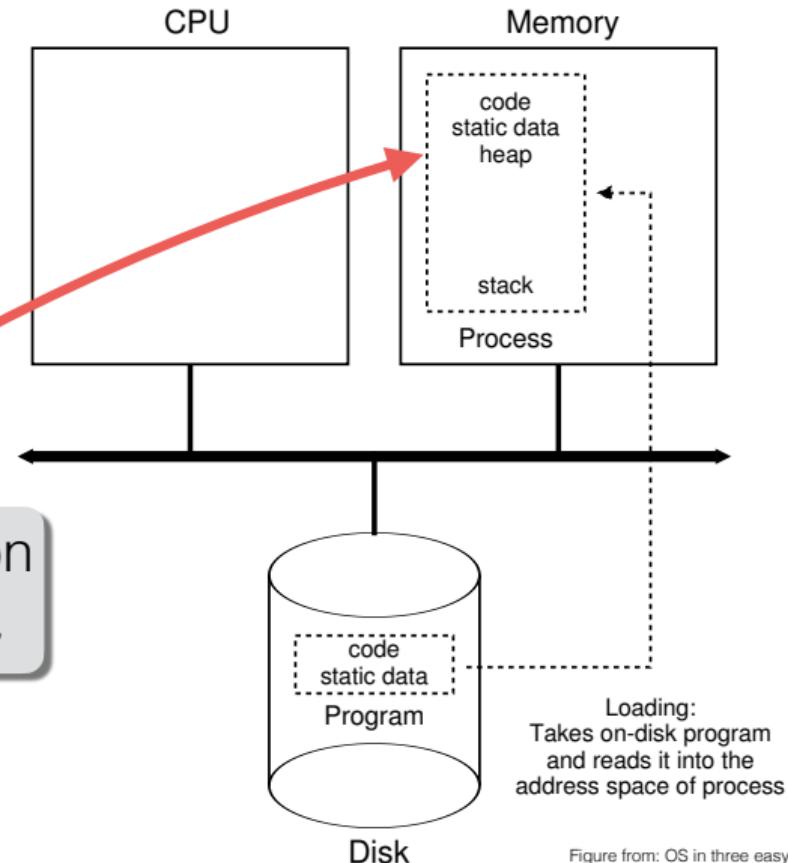
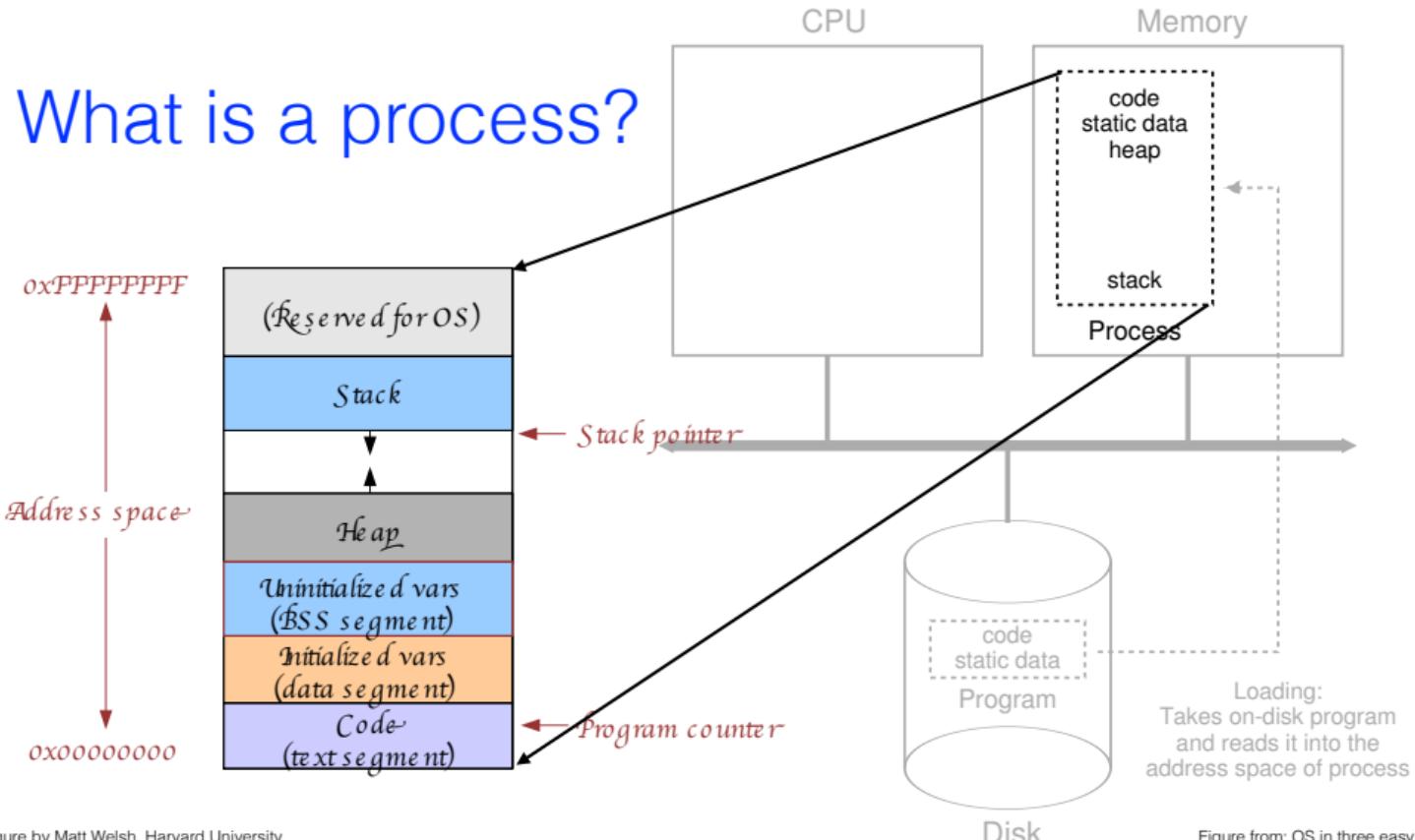


Figure from: OS in three easy pieces

Main components of the address space

What is a process?



The code part of the address space

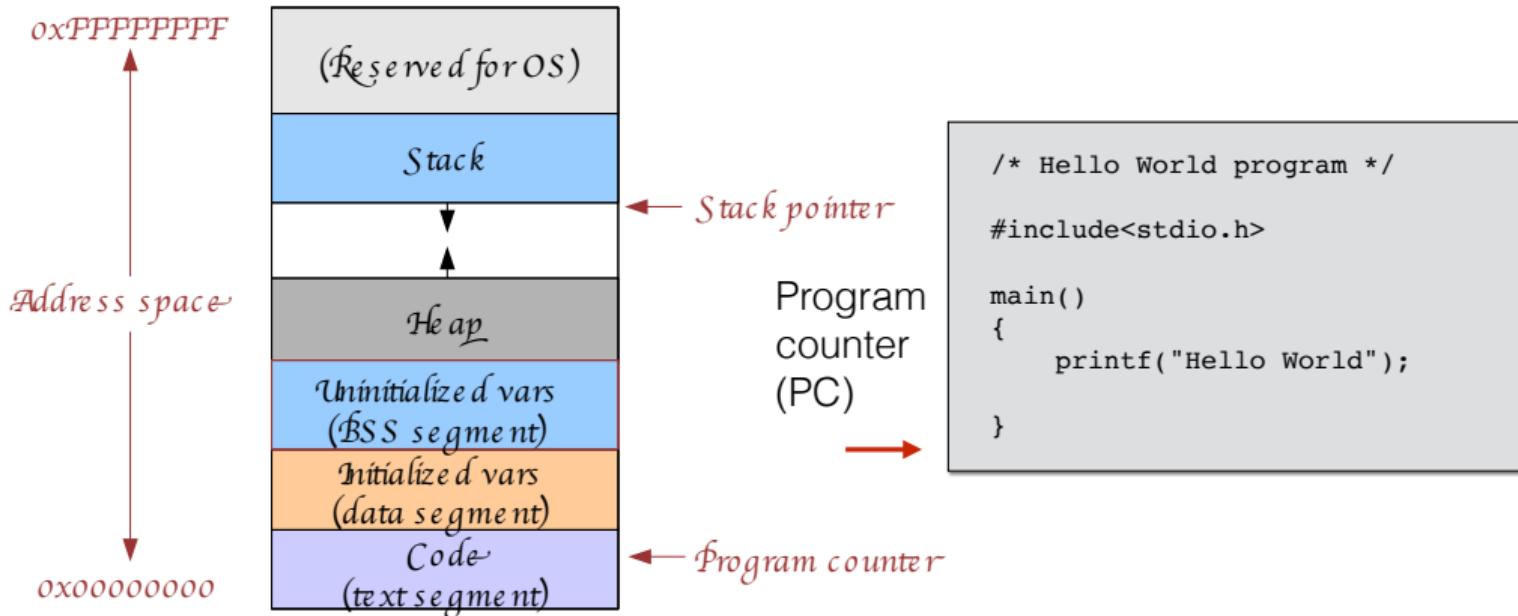


Figure by Matt Welsh, Harvard University.

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)

The Process Control Block (PCB)

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

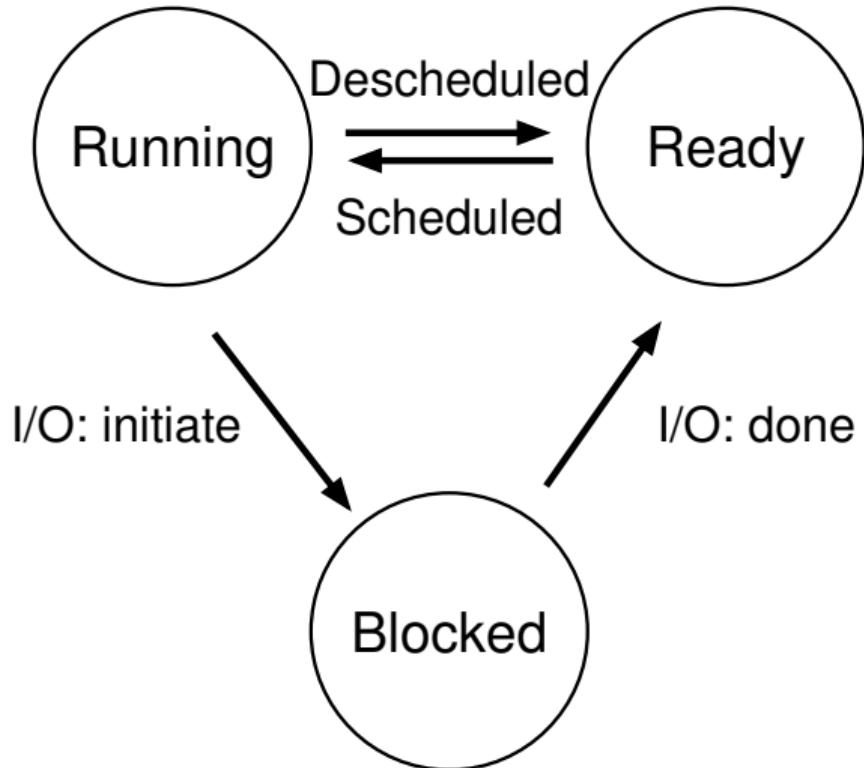
Defined in:

/include/linux/sched.h

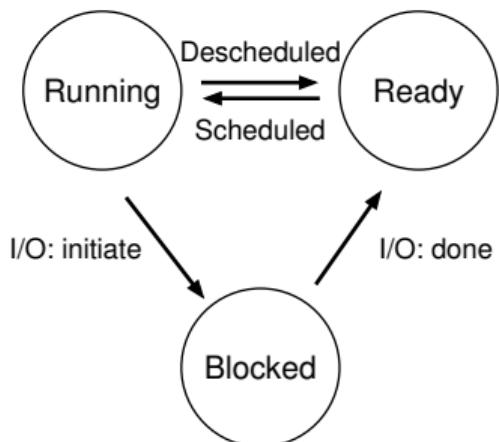
```
struct task_struct {  
    volatile long state; Execution state  
    unsigned long flags;  
    mm_segment_t addr_limit;  
    struct exec_domain *exec_domain;  
    volatile long need_resched;  
    unsigned long ptrace;  
    int lock_depth;  
    unsigned int cpu;  
    int prio, static_prio;  
    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_exec1;  
    unsigned task_dumpable:1;  
    pid_t pid; Process ID  
    pid_t pgid;  
    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_childexit;  
    struct completion *fork_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;  
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 swapped: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 pending 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.

Life cycle of a process

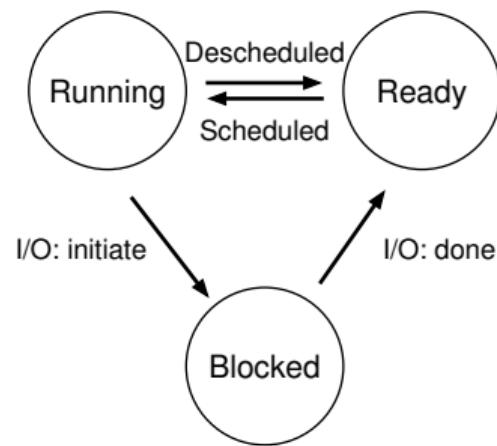


Two processes running, no I/O



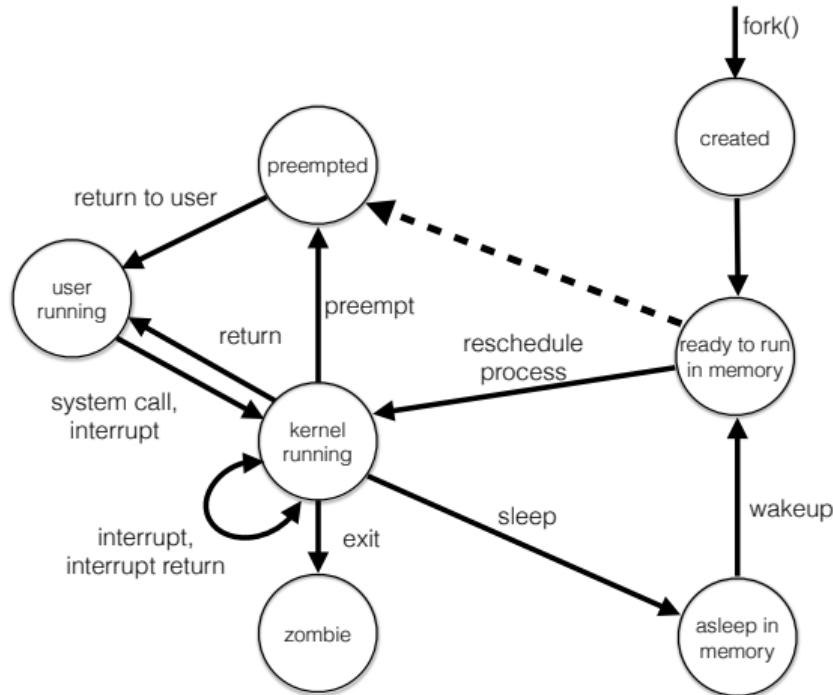
| Time | Process ₀ | Process ₁ | Notes |
|------|----------------------|----------------------|-------------------------------|
| 1 | Running | Ready | |
| 2 | Running | Ready | |
| 3 | Running | Ready | |
| 4 | Running | Ready | Process ₀ now done |
| 5 | – | Running | |
| 6 | – | Running | |
| 7 | – | Running | |
| 8 | – | Running | Process ₁ now done |

Two processes running, with I/O



| Time | Process ₀ | Process ₁ | Notes |
|------|----------------------|----------------------|--|
| 1 | Running | Ready | |
| 2 | Running | Ready | |
| 3 | Running | Ready | |
| 4 | Blocked | Running | Process ₀ initiates I/O |
| 5 | Blocked | Running | Process ₀ is blocked, 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 |

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.

Process states (Unix) without hard drive

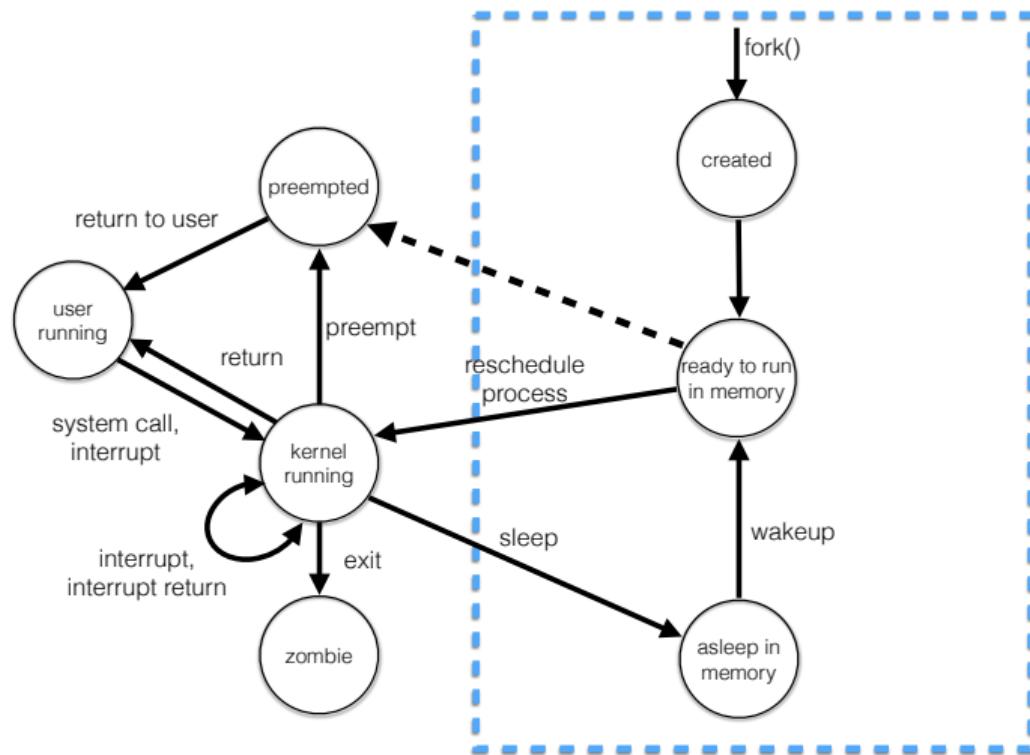


Figure adapted from Stallings' book

Process states (Unix) with hard drive

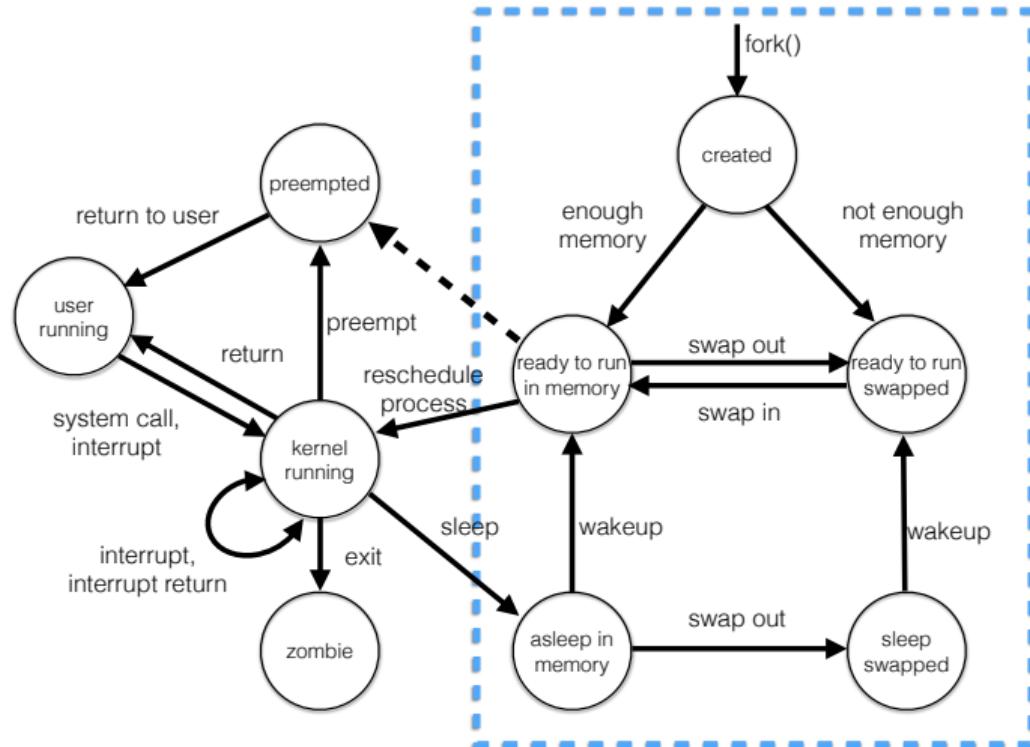
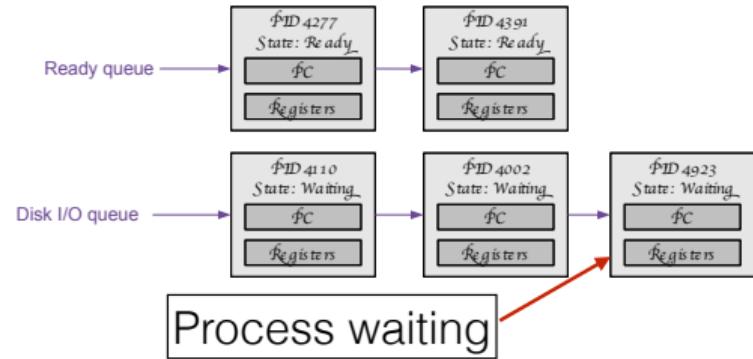
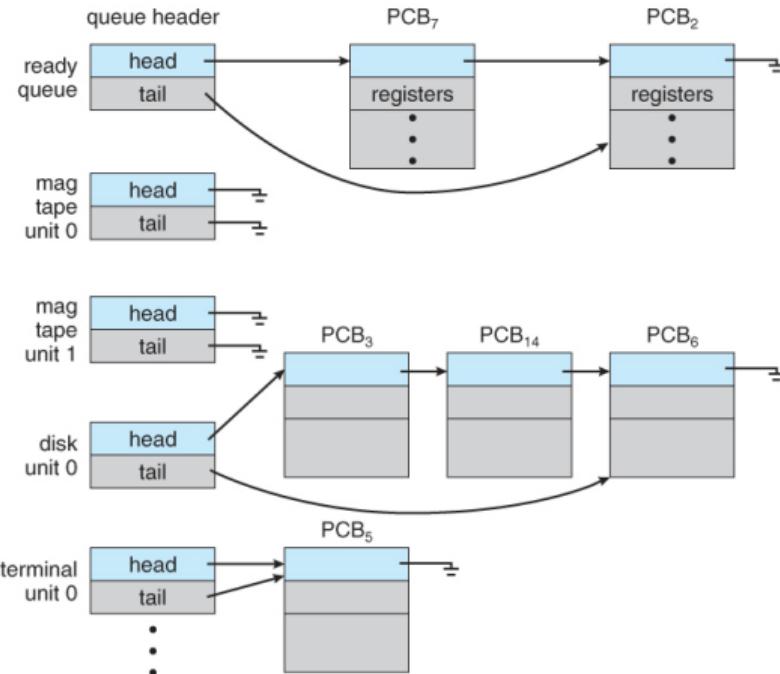


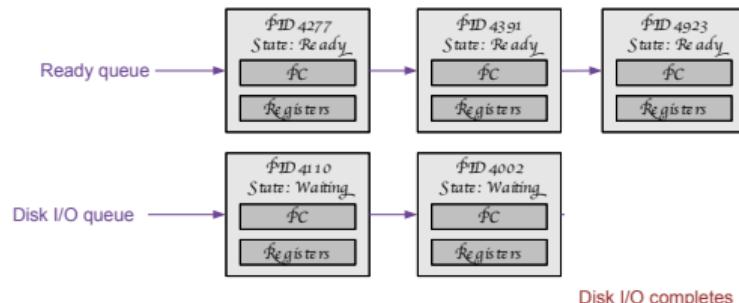
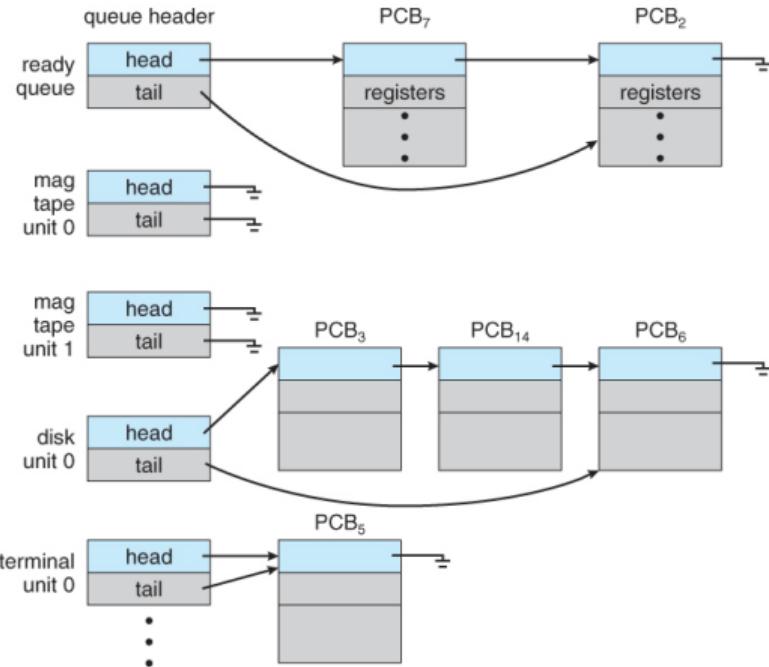
Figure adapted from Stallings' book

Ready queue and various I/O queue: 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.

Ready queue and various I/O queue: process moved to ready



- 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.

Processes

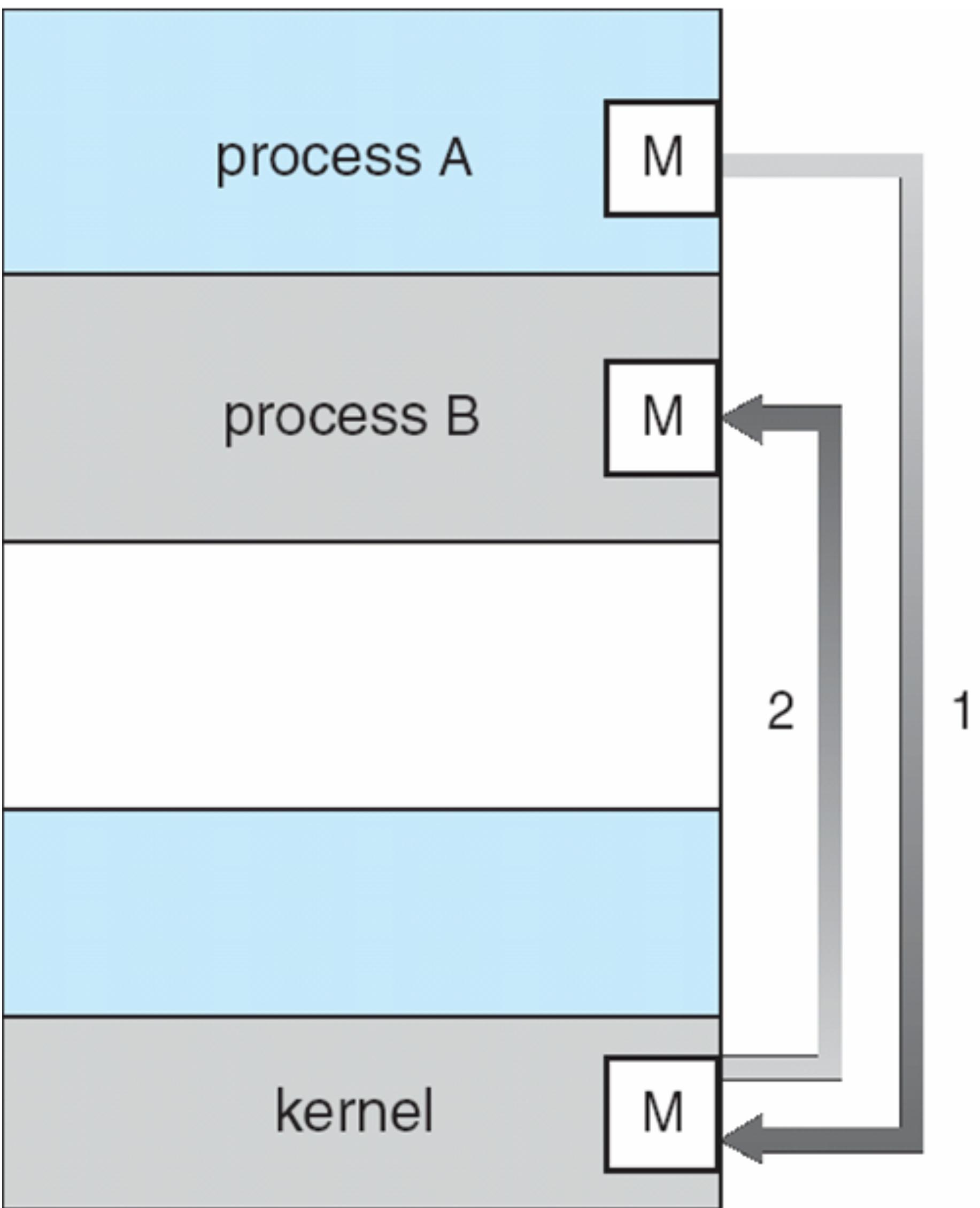
Inter-process communication

Cooperating Processes

- **Independent processes:** execute independently of other processes.
- **Cooperating processes:** depend on the execution of other processes.
 - Cooperation requires inter-process communication.

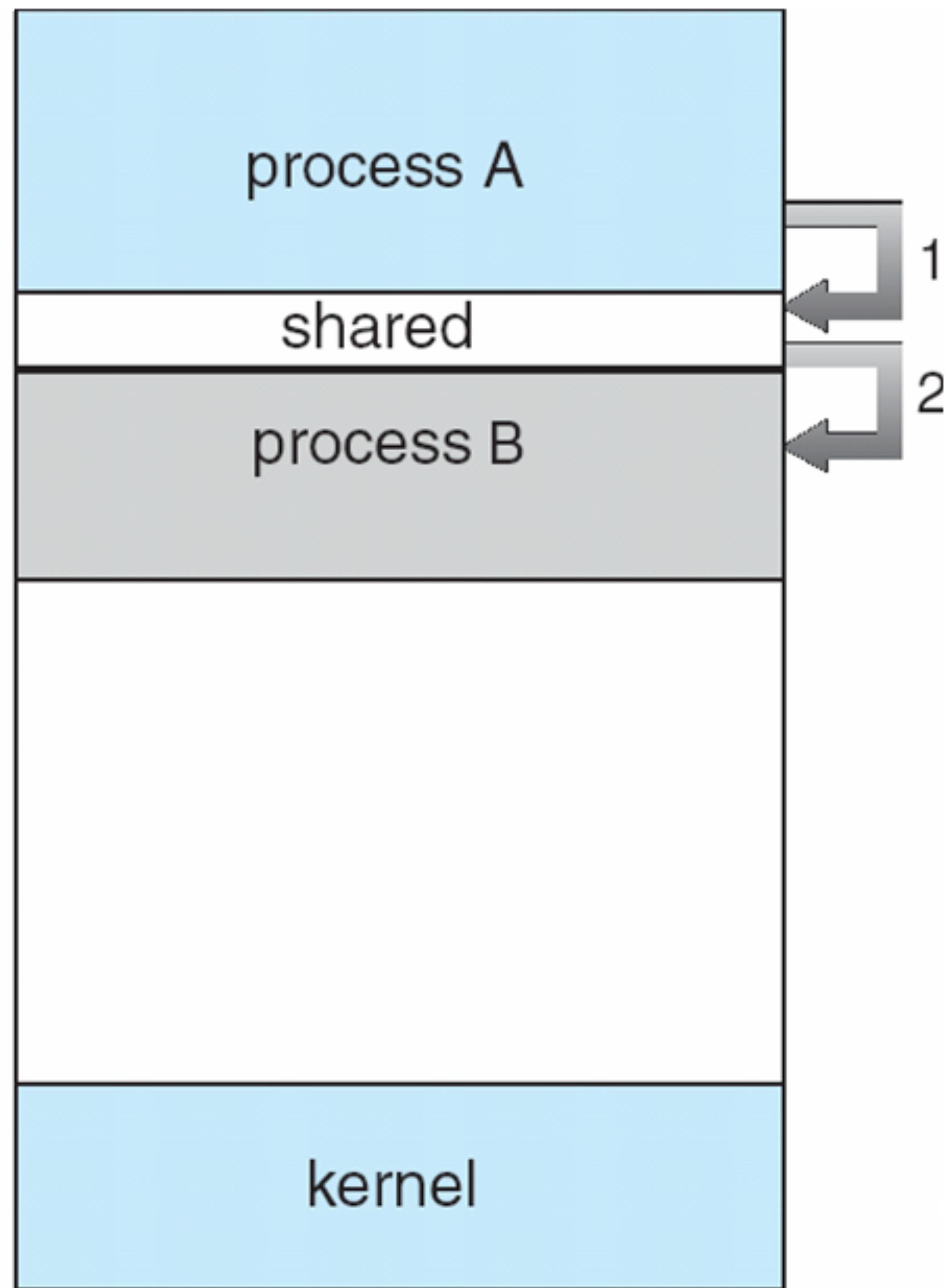
Models of IPC

Message-passing



(a)

Shared memory



(b)

Posix Shared Memory

1. Process first creates shared memory segment

```
segment id = shmget(IPC_PRIVATE, size, S_IRUSR |  
S_IWUSR);
```

2. Process wanting access to the shared memory must attach to it

```
shared_memory = (char *) shmat(id, NULL, 0);
```

3. Now the process could write to the shared memory

```
sprintf(shared_memory, "Writing to shared memory");
```

4. When done, a process can detach the shared memory from its address space

```
shmdt(shared_memory);
```

shm_server.c

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>

#define SHMSZ 27

main()
{
    char c;
    int shmid;
    key_t key;
    char *shm, *s;

    /*
     * We'll name our shared memory segment
     * "5678".
     */
    key = 5678;

    /*
     * Create the segment.
     */
    if ((shmid = shmget(key, SHMSZ,
                         IPC_CREAT | 0666)) < 0)
    {
        perror("shmget");
        exit(1);
    }
```

```
    /*
     * attach the segment to our data space.
     */
    if ((shm = shmat(shmid, NULL, 0)) ==
        (char *) -1)
    {
        perror("shmat");
        exit(1);
    }

    /* put things into the memory for the
     * other process to read.*/
    s = shm;

    for (c = 'a'; c <= 'z'; c++)
        *s++ = c;
    *s = NULL;

    /* Finally, wait until the other process
     * changes the first character of our
     * memory to '*', indicating that it
     * has read what we put there. */
    while (*shm != '*')
        sleep(1);

    exit(0);
}
```

shm_client.c

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>

#define SHMSZ 27

main()
{
    int shmid;
    key_t key;
    char *shm, *s;

    /*
     * We need to get the segment named
     * "5678", created by the server.
     */
    key = 5678;

    /*
     * Locate the segment.
     */
    if((shmid=shmget(key,SHMSZ,0666)) < 0)
    {
        perror("shmget");
        exit(1);
    }
```

```
/*
 * attach the segment to our data space.
 */
if ((shm = shmat(shmid, NULL, 0)) ==
    (char *) -1)
{
    perror("shmat");
    exit(1);
}

/*
 * read what the server put in
 * the memory.
 */
for (s = shm; *s != NULL; s++)
    putchar(*s);
putchar('\n');

/*
 * Finally, change the first character
 * of the segment to '*', indicating
 * we have read the segment.
 */
*shm = '*';

exit(0);
}
```

Message Passing

Show programs:

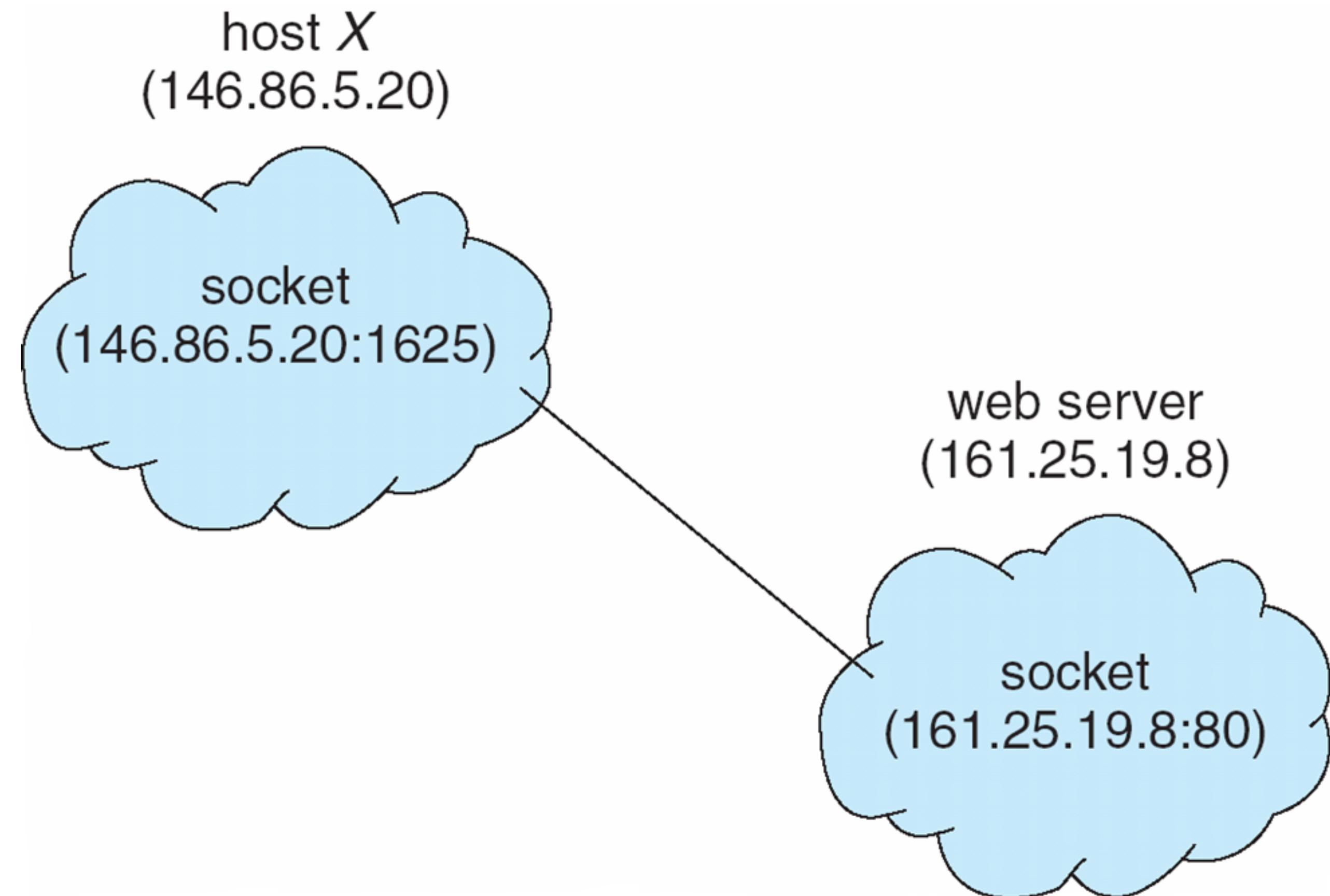
- `message_send.c`
- `message_rec.c`

Processes

Sockets (communicating using the network)

Sockets

- A **socket** is defined as an *endpoint for communication*
- Concatenation of **IP address** and **port**
- The socket **161.25.19.8:1625** refers to port **1625** on host **161.25.19.8**
- Communication consists between a pair of sockets



Sockets

- Most interprocess communication uses the **client-server model**.
- Client needs to know of the existence of and the address of the server, but the server does not need to know the address of (or even the existence of) the client prior to the connection being established.
- Once a connection is established, both sides can send and receive information.
- A socket is one end of an interprocess communication channel. The two processes each establish their own socket.

Socket address domains

- Two processes can communicate with each other only if their sockets are of the same type and in the same domain.
- There are two widely used address domains, each has its own address format:
 - the unix domain: two processes share a common file system.
 - the Internet domain: two processes running on any two hosts on the Internet.

Sockets

- Connection steps on the client side
- Connection steps on the server side (single connection)

Sockets

Connection steps on the client side:

1. Create a socket with the `socket()` system call.
2. Connect the socket to the address of the server using the `connect()` system call.
3. Send and receive data. There are many ways to do this. The simplest is to use the `read()` and `write()` system calls.

Sockets

Connection steps on the server side (single connection):

1. Create a socket with the `socket()` system call
2. Bind the socket to an address using the `bind()` system call. For a server socket on the Internet, an address consists of a port number on the host machine.
3. Listen for connections with the `listen()` system call
4. Accept a connection with the `accept()` system call. This call typically blocks until a client connects with the server.
5. Send and receive data

Socket types

Two widely used socket types:

- **stream sockets**: communicate via a continuous stream of characters. Stream sockets use TCP (Transmission Control Protocol), which is a reliable, stream-oriented protocol.
- **datagram sockets**: read entire messages at once. Use UDP (Unix Datagram Protocol), which can be unreliable and message oriented.

in CSE4001, we will work with sockets in the Internet domain using the TCP protocol.

Examples of connecting using sockets

- client-server with single connection.
- server forking multiple "handler" processes for each client connection.

```
while (1)
{
    newsockfd = accept(sockfd,
                       (struct sockaddr *) &cli_addr, &clilen);
    if (newsockfd < 0)
        error("ERROR on accept");
    pid = fork();
    if (pid < 0)
        error("ERROR on fork");
    if (pid == 0)
    {
        close(sockfd);
        dostuff(newsockfd);
        exit(0);
    }
    else
        close(newsockfd);
} /* end of while */
```

Server forking multiple "handler" processes for each client connection.

Accepting connection goes inside an infinite loop.

```
while (1)
{
    newsockfd = accept(sockfd,
                       (struct sockaddr *) &cli_addr, &clilen);
    if (newsockfd < 0)
        error("ERROR on accept");
    pid = fork();
    if (pid < 0)
        error("ERROR on fork");
    if (pid == 0)
    {
        close(sockfd);
        dostuff(newsockfd);
        exit(0);
    }
    else
        close(newsockfd);
} /* end of while */
```

Server forking multiple "handler" processes for each client connection.

Connection is established.
Create new process to handle the service.

```
while (1)
{
    newsockfd = accept(sockfd,
                       (struct sockaddr *) &cli_addr, &clilen);
    if (newsockfd < 0)
        error("ERROR on accept");
    pid = fork();
    if (pid < 0)
        error("ERROR on fork");
    if (pid == 0)
    {
        close(sockfd);
        dostuff(newsockfd);
        exit(0);
    }
    else
        close(newsockfd);
} /* end of while */
```

Server forking multiple "handler" processes for each client connection.

Child process will close **sockfd** and call the handling function passing **newsockfd** as argument.

Once the communication between client and handler is completed, child exits.

```
while (1)
{
    newsockfd = accept(sockfd,
                       (struct sockaddr *) &cli_addr, &clilen);
    if (newsockfd < 0)
        error("ERROR on accept");
    pid = fork();
    if (pid < 0)
        error("ERROR on fork");
    if (pid == 0)
    {
        close(sockfd);
        dostuff(newsockfd);
        exit(0);
    }
    else
        close(newsockfd);
} /* end of while */
```

Server forking multiple "handler" processes for each client connection.

Parent closes newsockfd and returns to accept () to wait for a new connection.

The return of the zombies

A zombie is a process that has terminated but cannot be permitted to fully die because at some point in the future, the parent of the process might execute a `wait()` and would want information about the death of the child.



The invasion of the zombies

Problem with the previous code:

- Each of these connections will create a zombie when the connection is terminated.
- When a child dies, it sends a **SIGCHLD** signal to its parent. But, the handling of this signal is system dependent.

```

void proc_exit() {
    int wstat;
    union wait wstat;
    pid_t pid;

    while (TRUE) {
        pid = wait3 (&wstat, WNOHANG, (struct rusage *)NULL );
        if (pid == 0)
            return;
        else if (pid == -1)
            return;
        else
            printf ("Return code: %d\n", wstat.w_retcode);
    }
}

int main () {
    signal (SIGCHLD, proc_exit);
    switch (fork()) {
        case -1:
            perror ("main: fork");
            exit (0);
        case 0:
            printf ("I'm alive (temporarily)\n");
            exit (rand());
        default:
            pause();
    }
}

```

SunOS: Example of catching SIGCHLD and avoiding zombies



Credits:

Slides on socket communication based on the sockets tutorial from:

http://www.linuxhowtos.org/C_C++/socket.htm

CSE4001: Operating Systems Concepts

Processes

How to create and control processes: Process API

- Create
- Destroy
- Wait
- Miscellaneous control
- Status

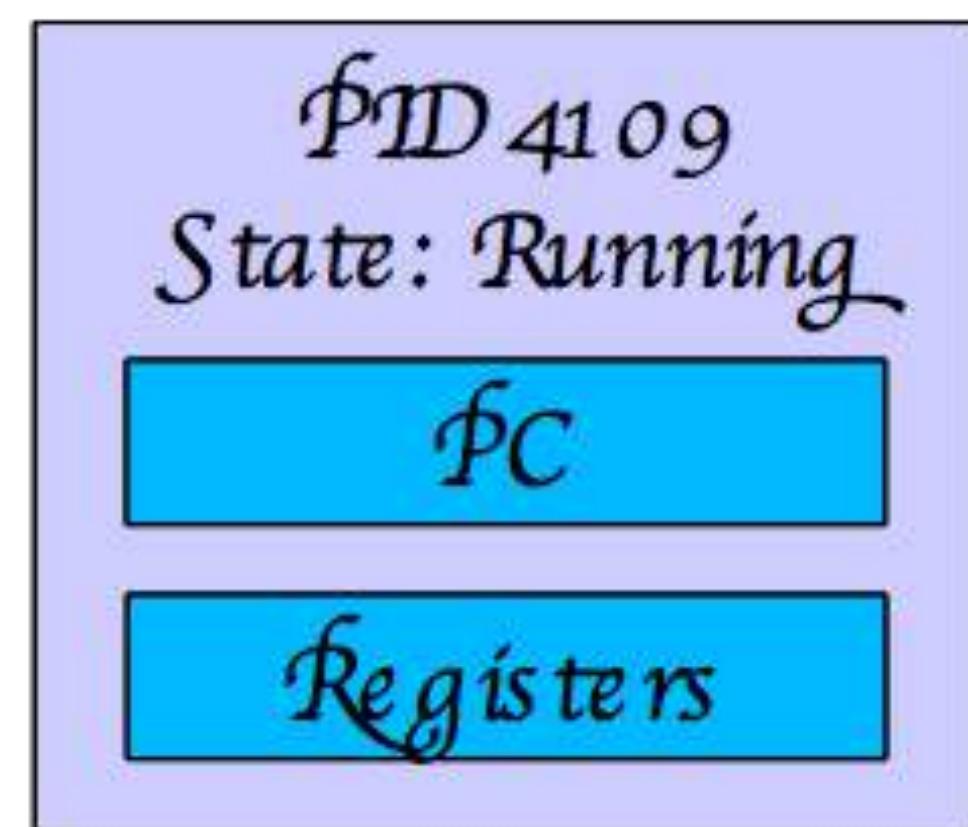
Content

- Creating processes with `fork()`

In UNIX, use the fork() system call to create a new process

- This creates an **exact duplicate** of the parent process!!
- Creates and initializes a new PCB
- Creates a new address space
- Copies entire contents of parent's address space into the child
- Initializes CPU and OS resources to a copy of the parent's
- Places new PCB on ready queue

Process calls fork()



```
int main()
{
    int x = 0;
    ➔ fork();
    x = 1;

    return 1;
}
```

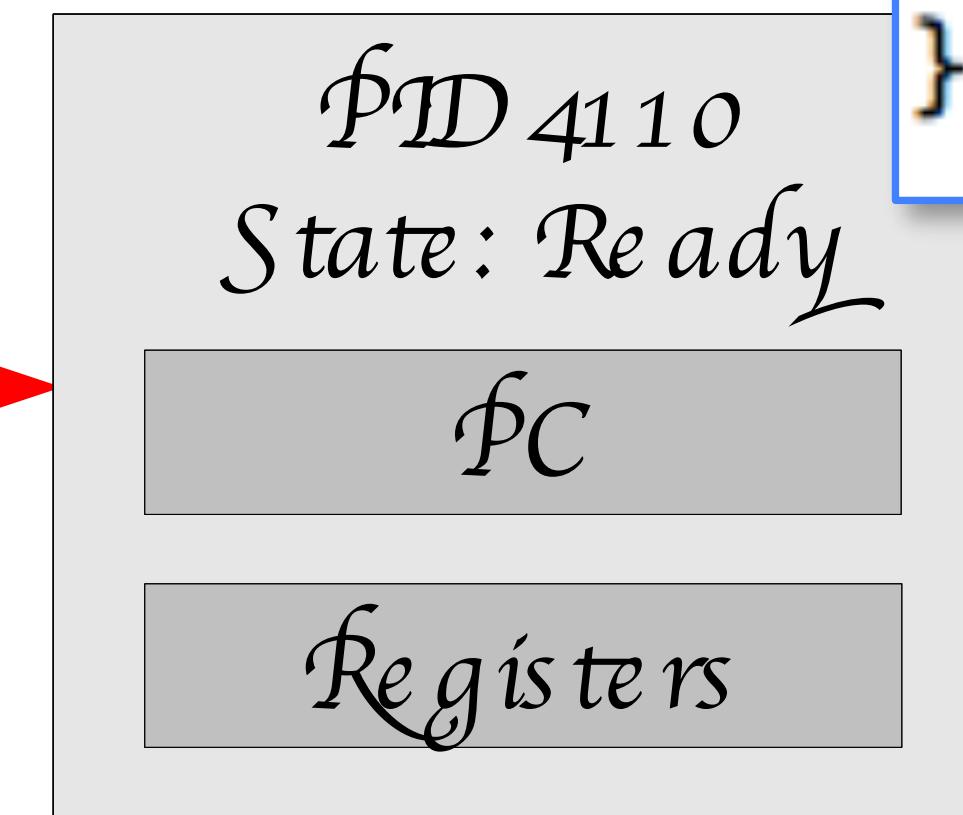
In UNIX, use the fork() system call to create a new process

- This creates an **exact duplicate** of the parent process!!
- Creates and initializes a new PCB
- Creates a new address space
- Copies entire contents of parent's address space into the child
- Initializes CPU and OS resources to a copy of the parent's
- Places new PCB on ready queue

Process calls fork()



Copy state



```
int main()
{
    int x = 0;
    → fork();
    x = 1;

    return 1;
}
```

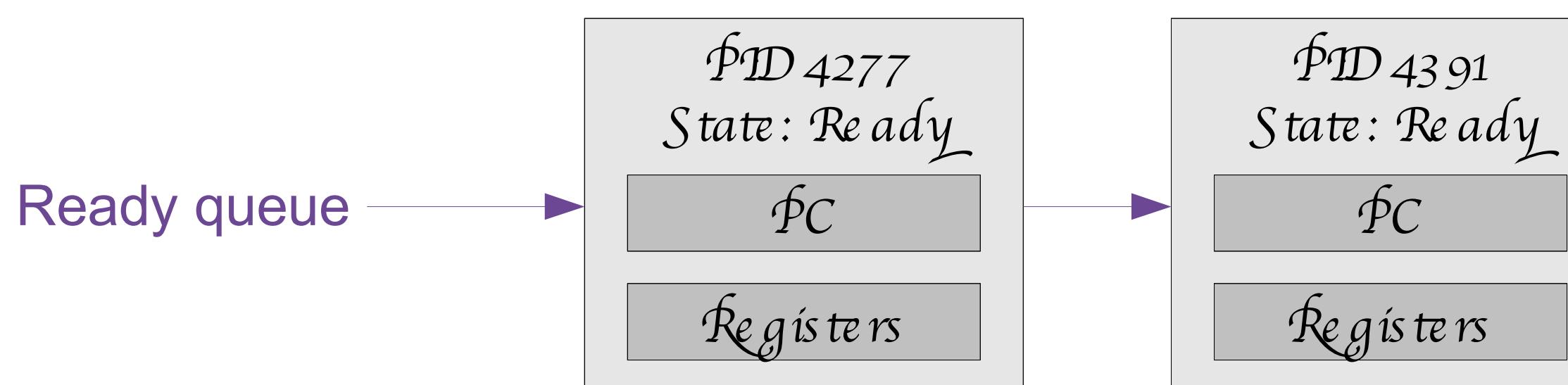
UNIX fork mechanism

In UNIX, use the fork() system call to create a new process

- This creates an **exact duplicate** of the parent process!!
- Creates and initializes a new PCB
- Creates a new address space
- Copies entire contents of parent's address space into the child
- Initializes CPU and OS resources to a copy of the parent's
- Places new PCB on ready queue

```
int main()
{
    int x = 0;
    → fork();
    x = 1;

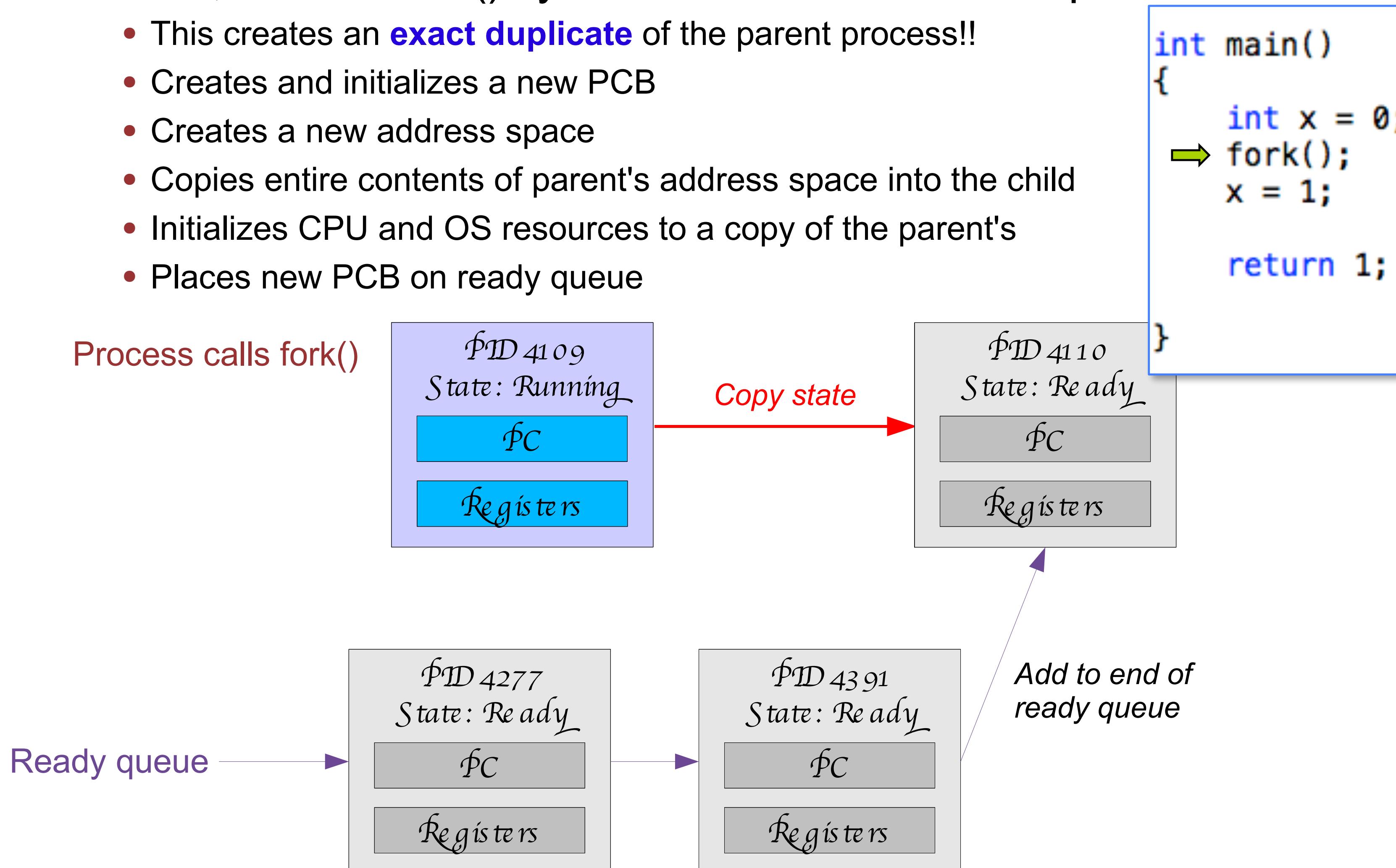
    return 1;
}
```



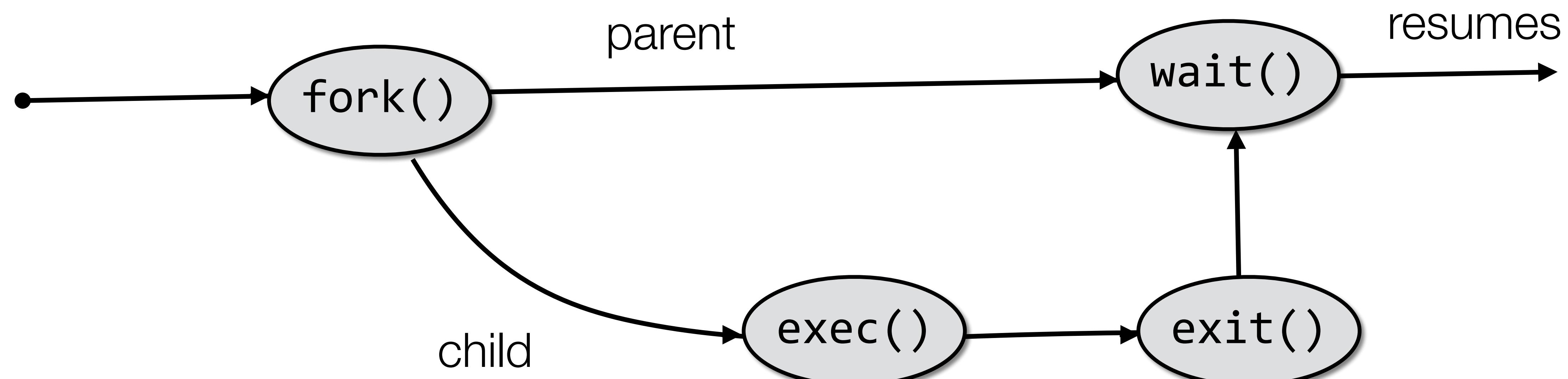
UNIX fork mechanism

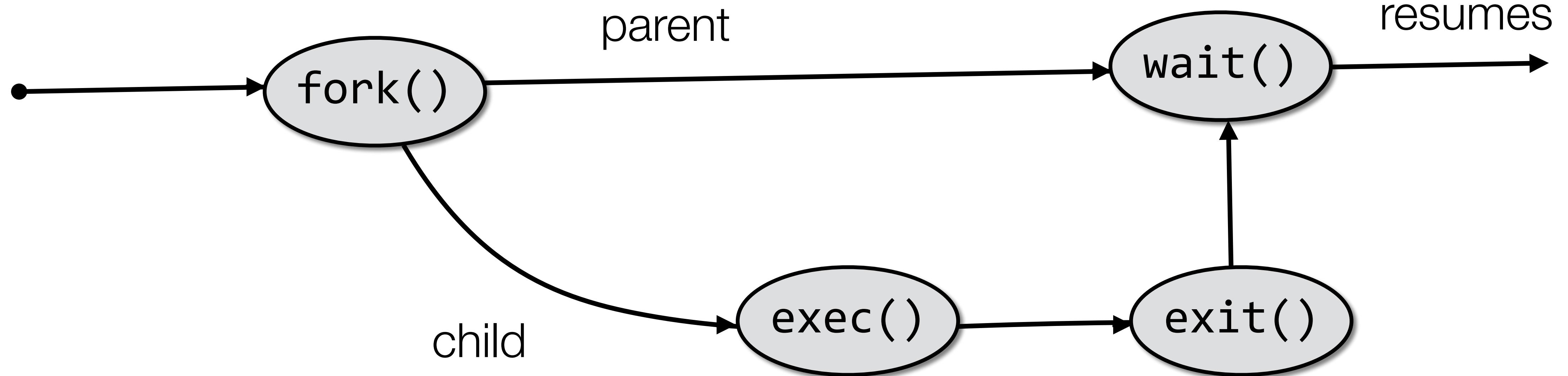
In UNIX, use the fork() system call to create a new process

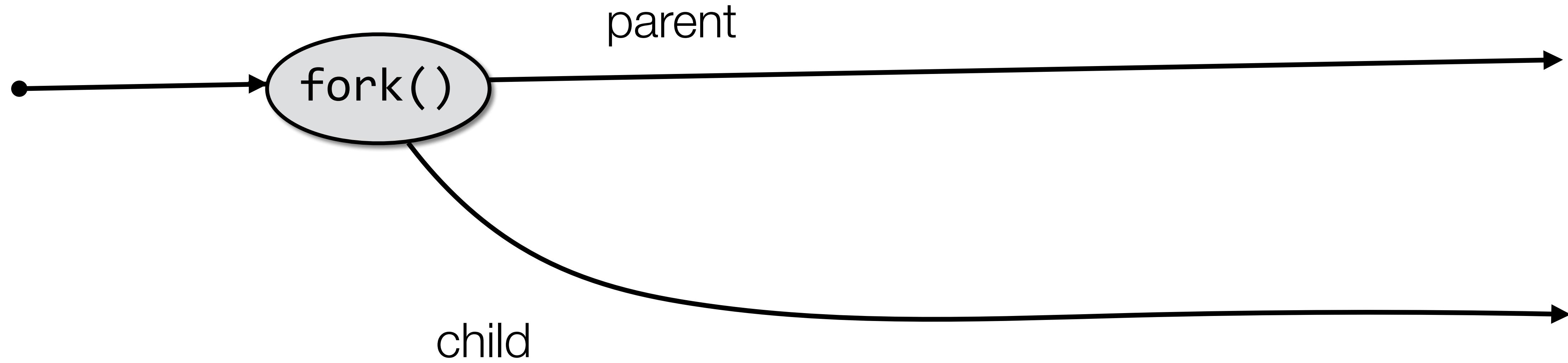
- This creates an **exact duplicate** of the parent process!!
- Creates and initializes a new PCB
- Creates a new address space
- Copies entire contents of parent's address space into the child
- Initializes CPU and OS resources to a copy of the parent's
- Places new PCB on ready queue

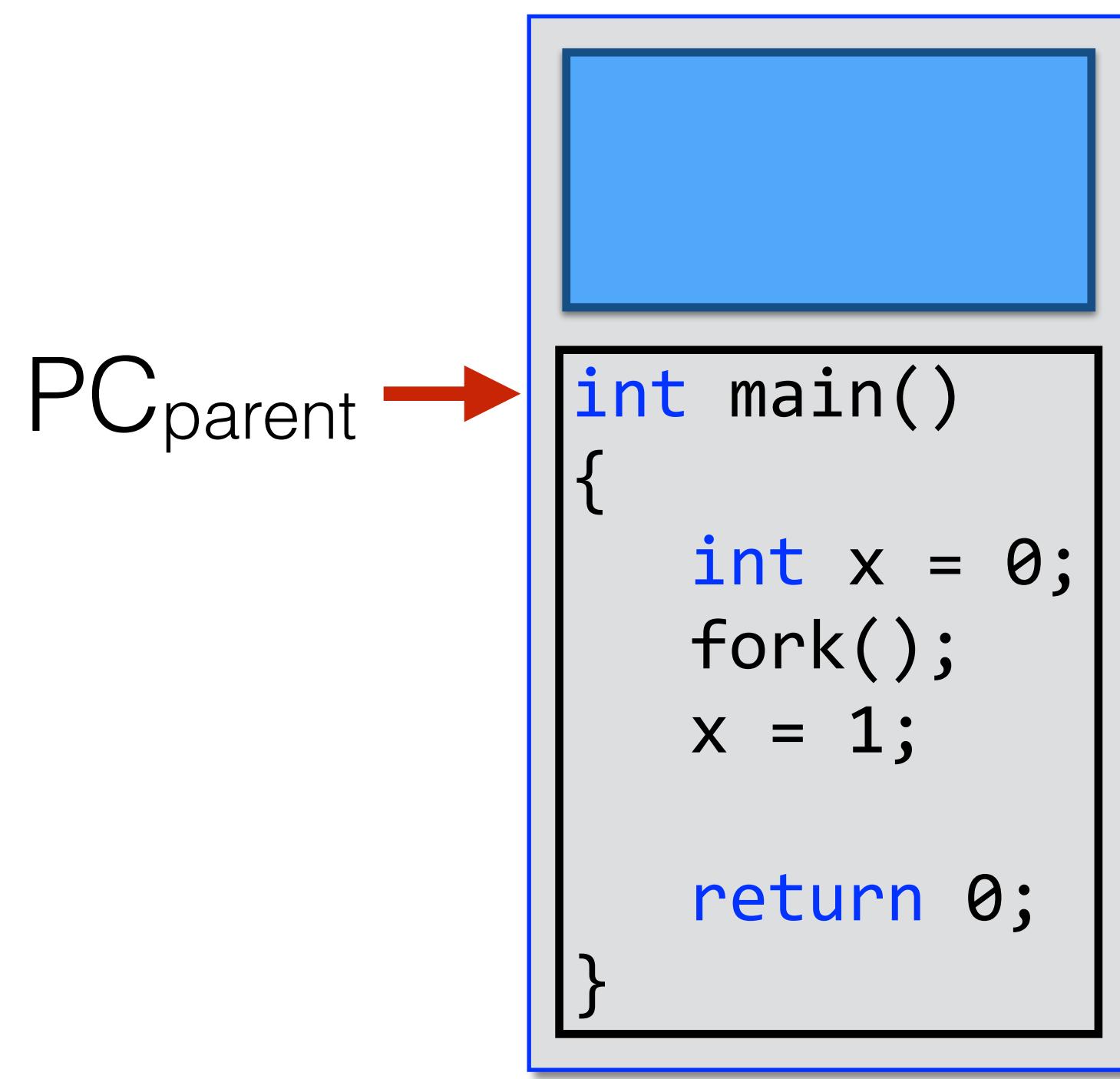
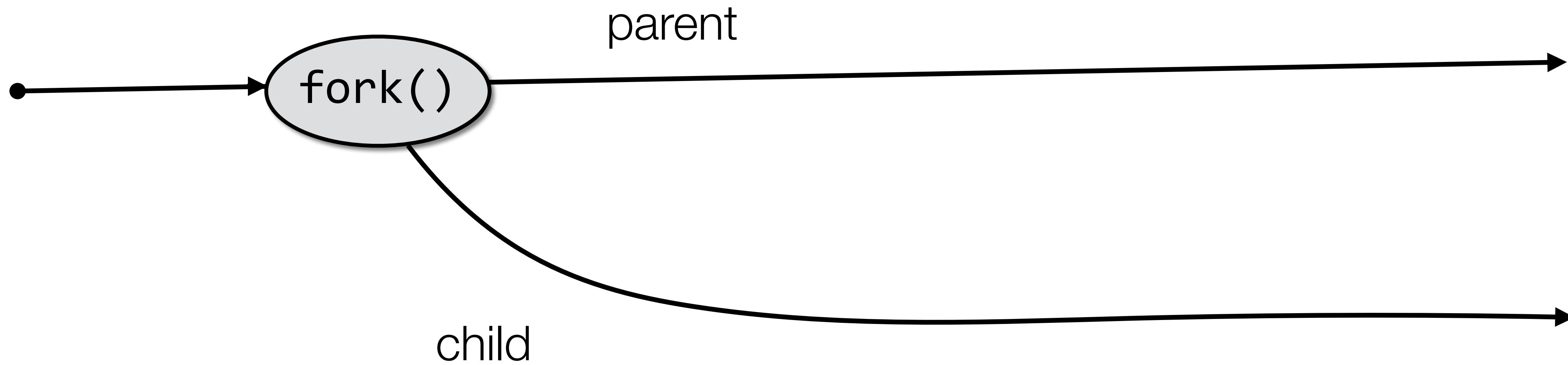


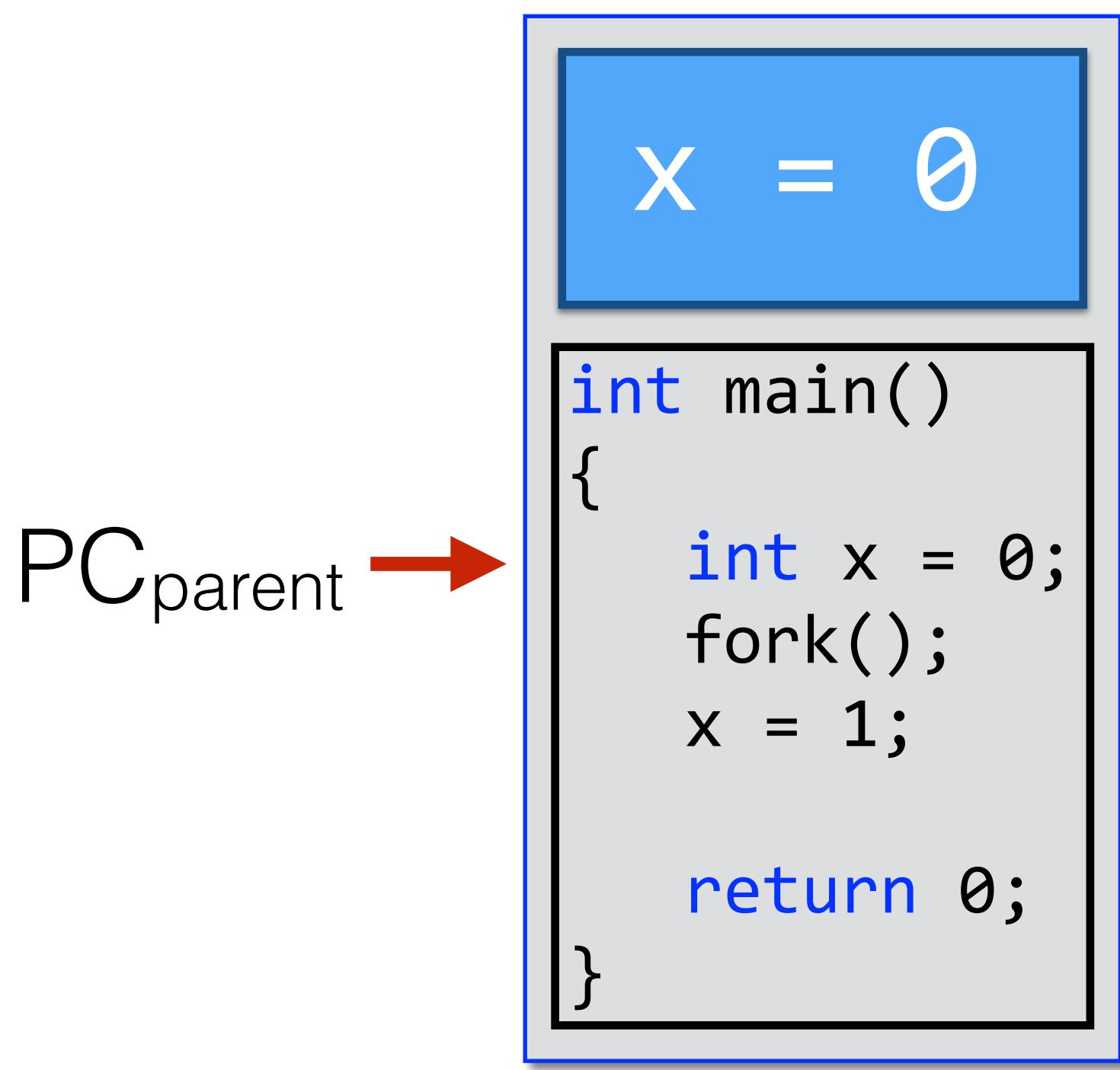
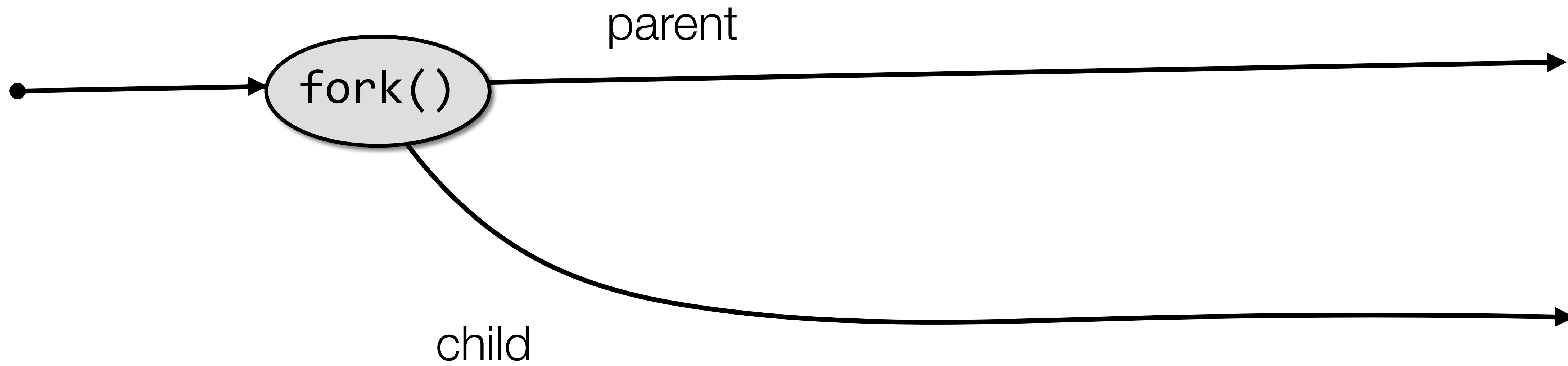
Process API: creation and program execution

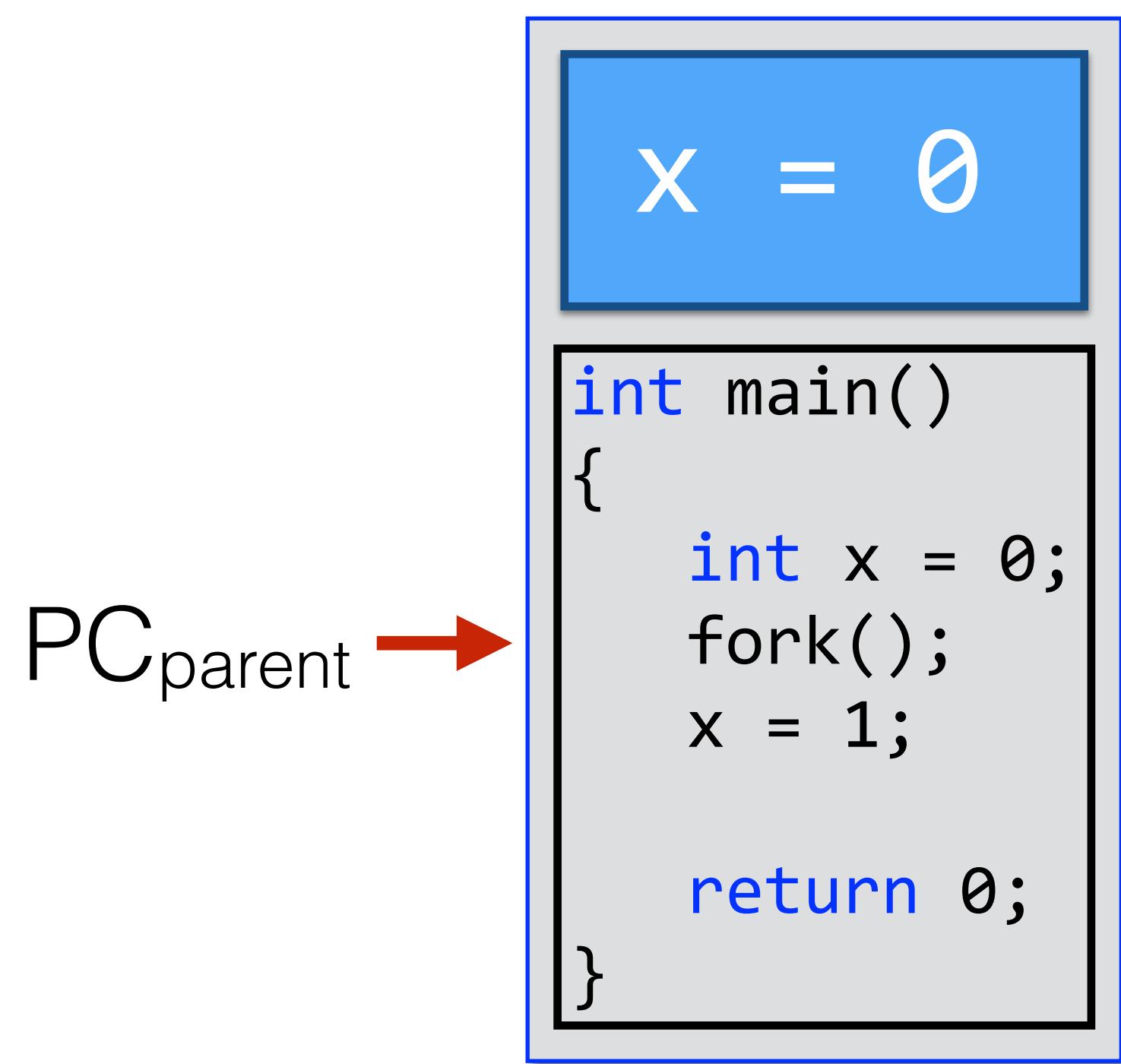
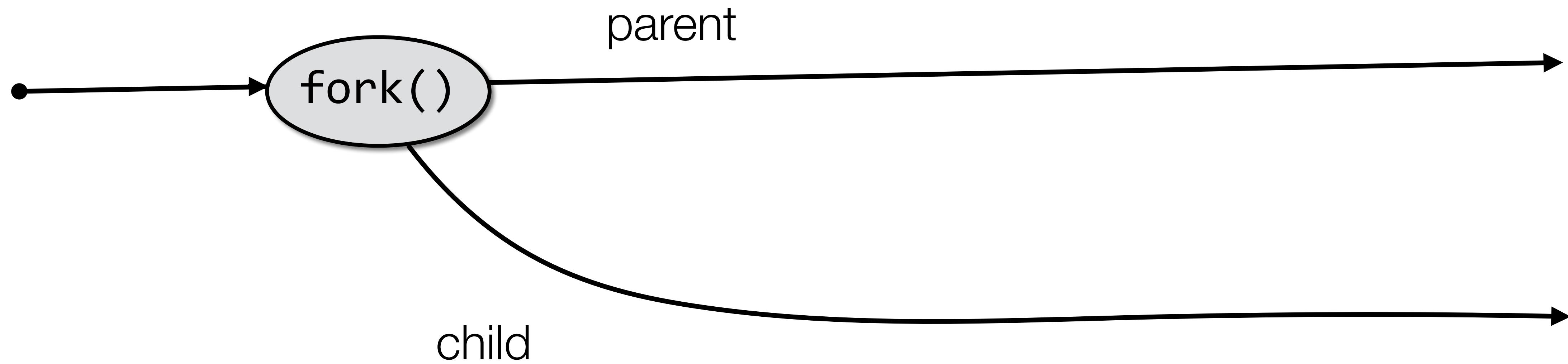


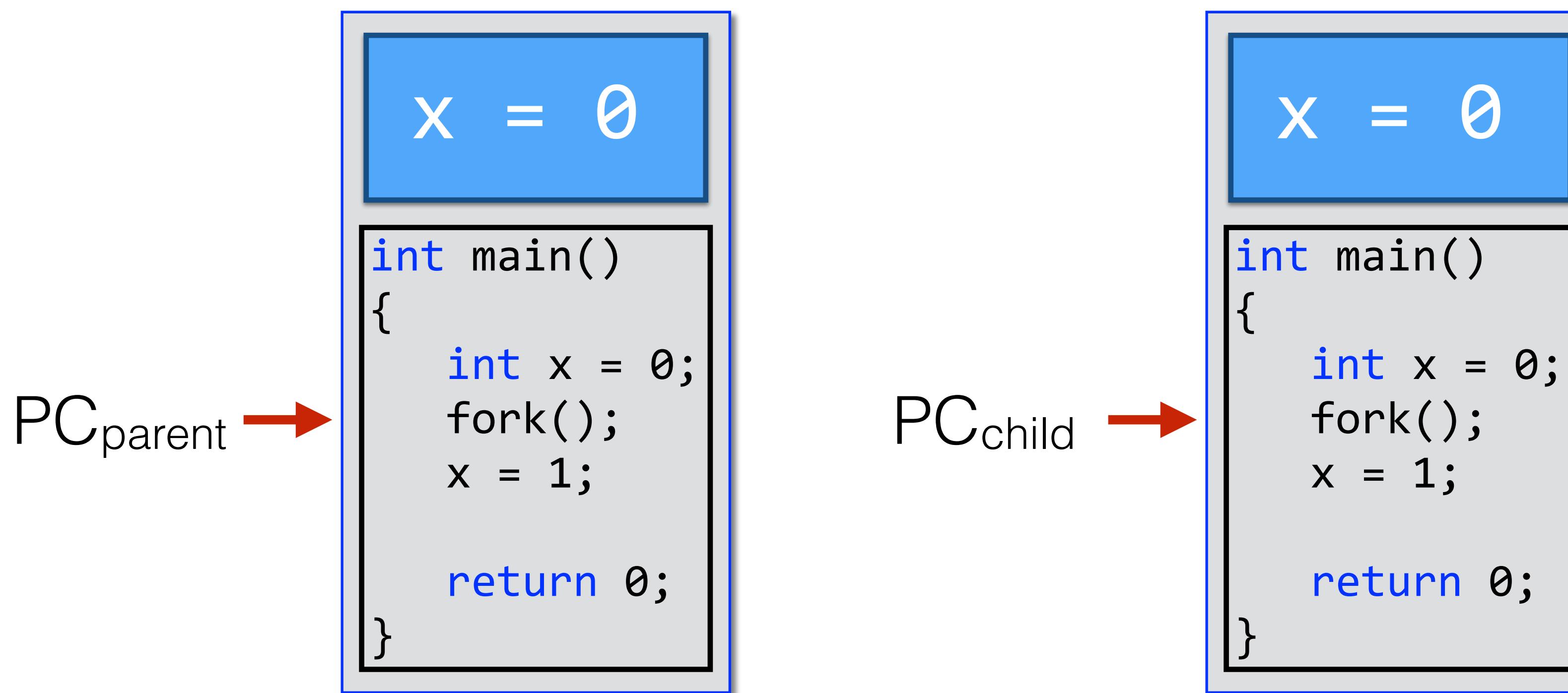
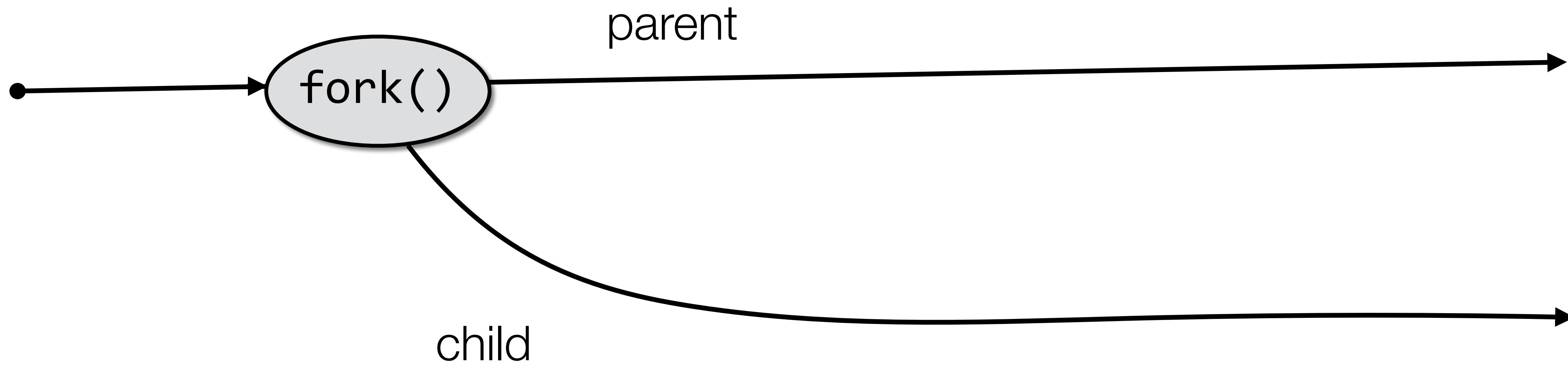


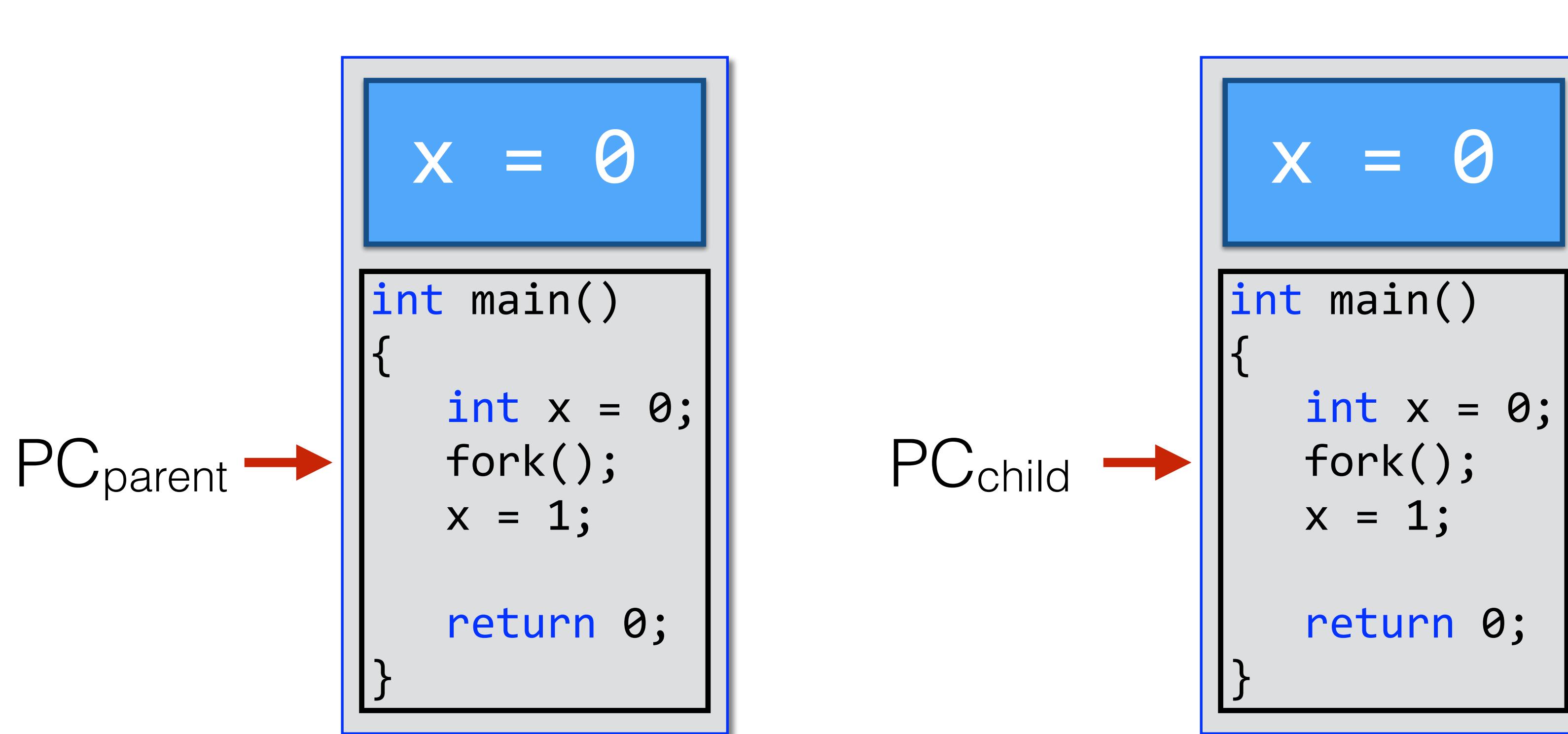
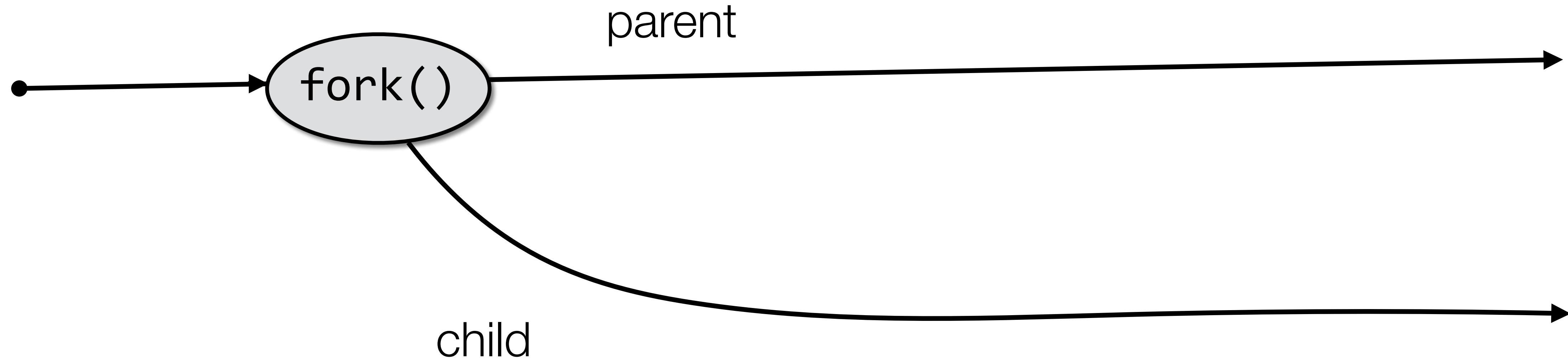






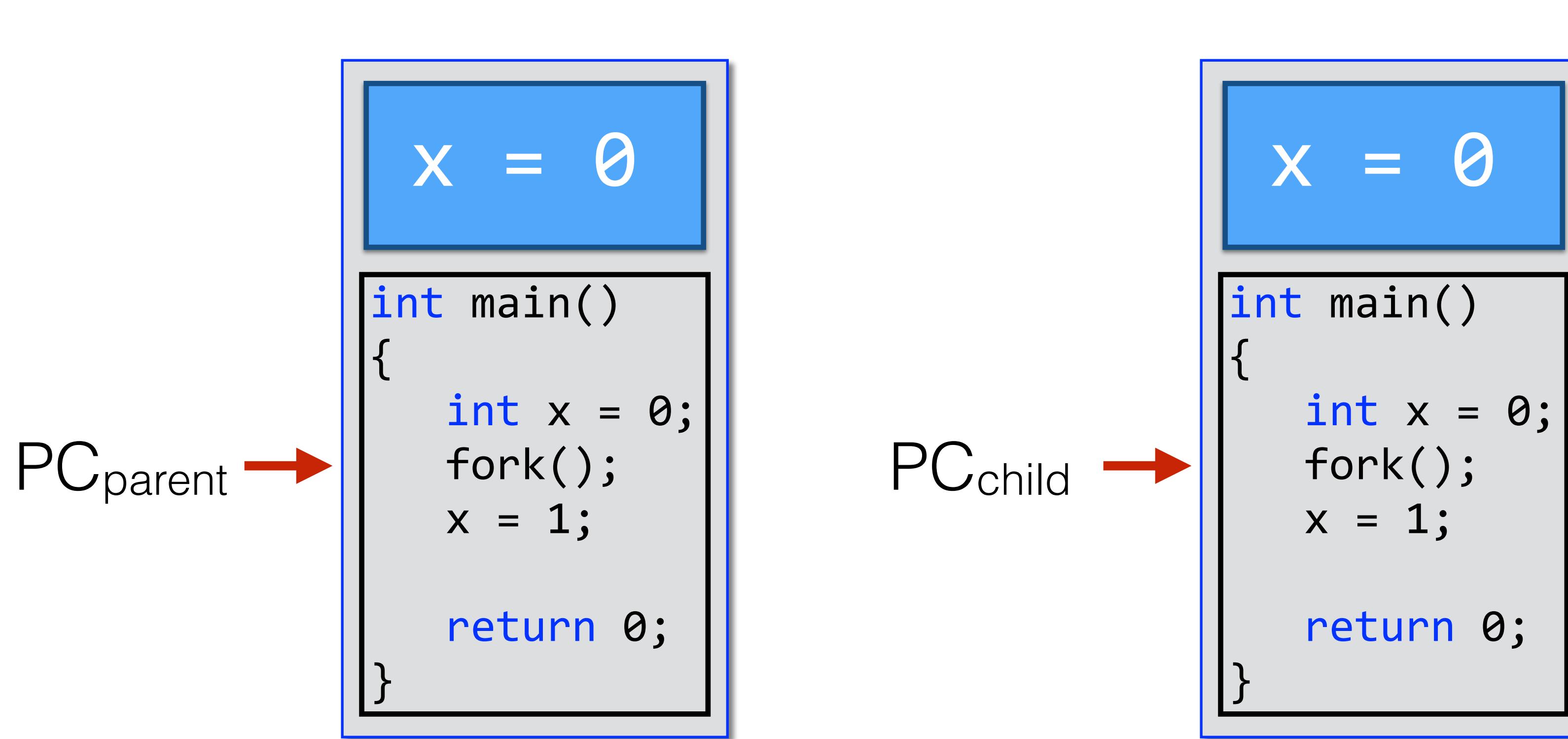
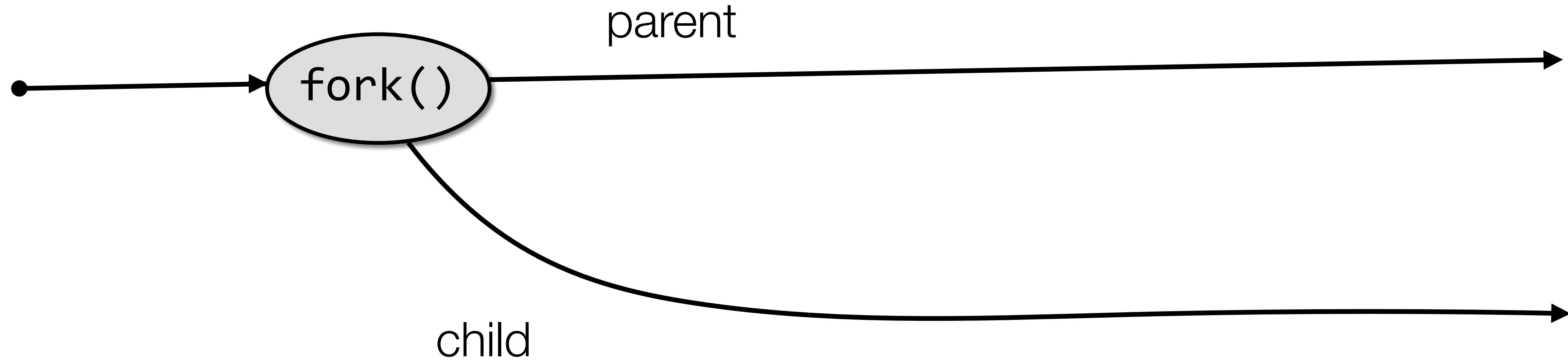






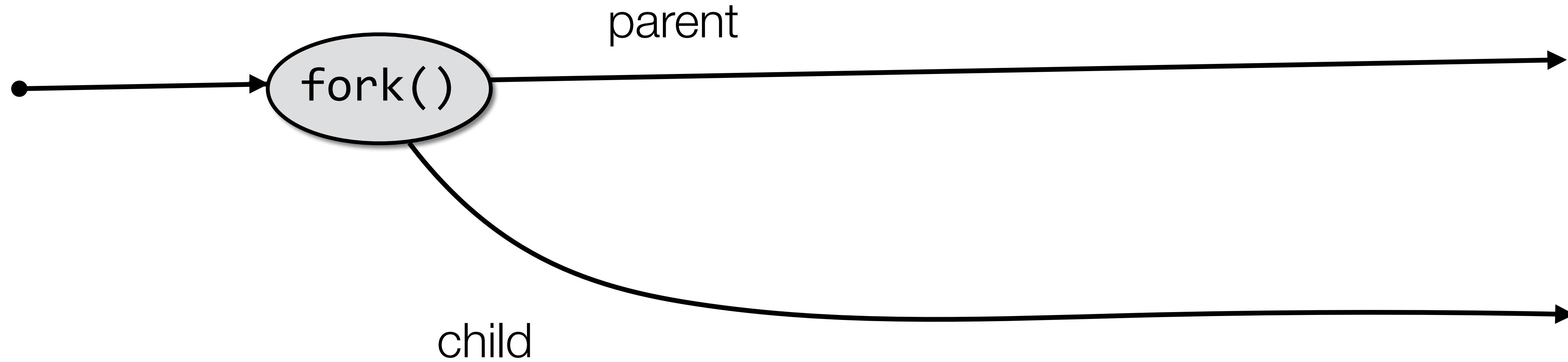
The `fork()` call is issued. A new process is created (child).

What are the states of the parent and child processes?



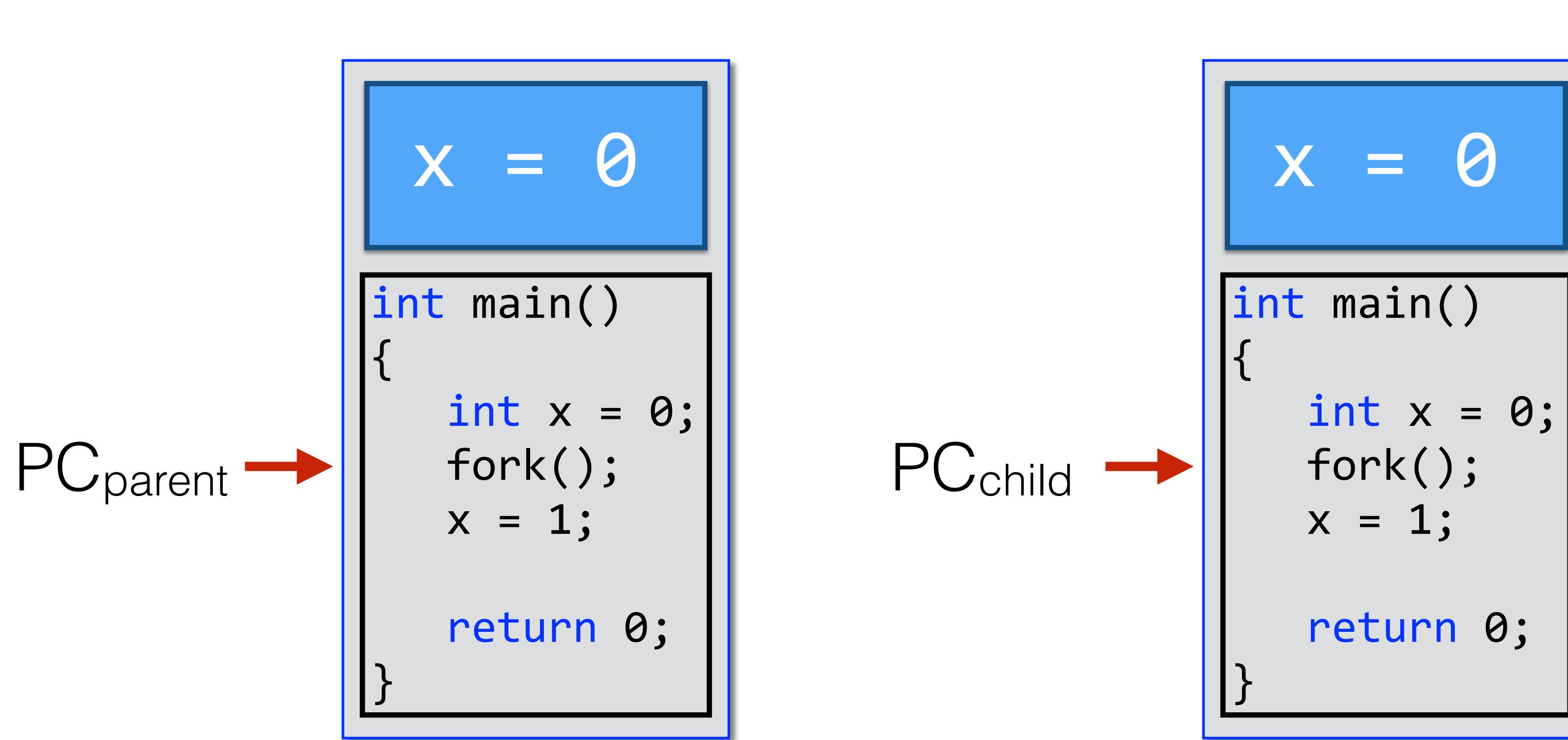
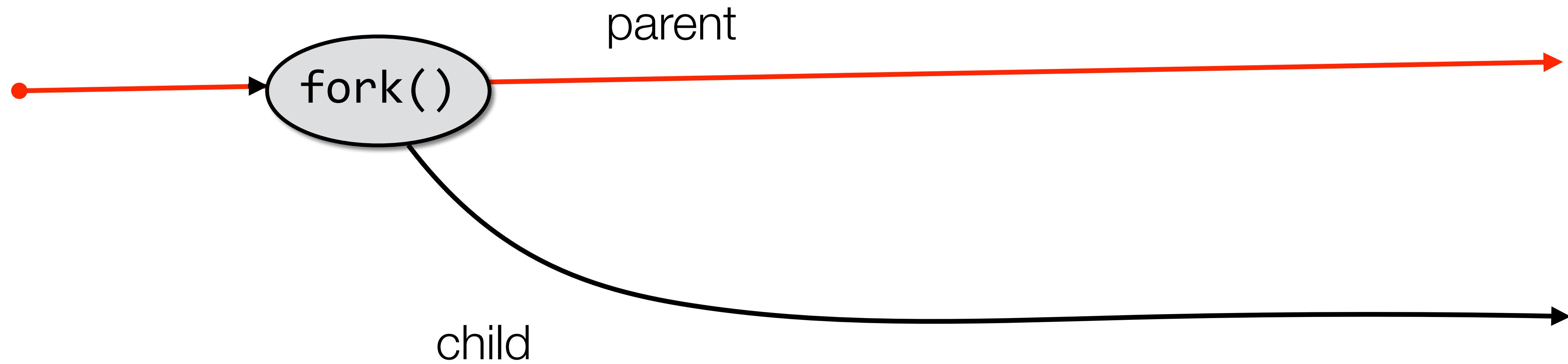
The `fork()` call is issued. A new process is created (child).

Which one of the two process will be scheduled to run first?



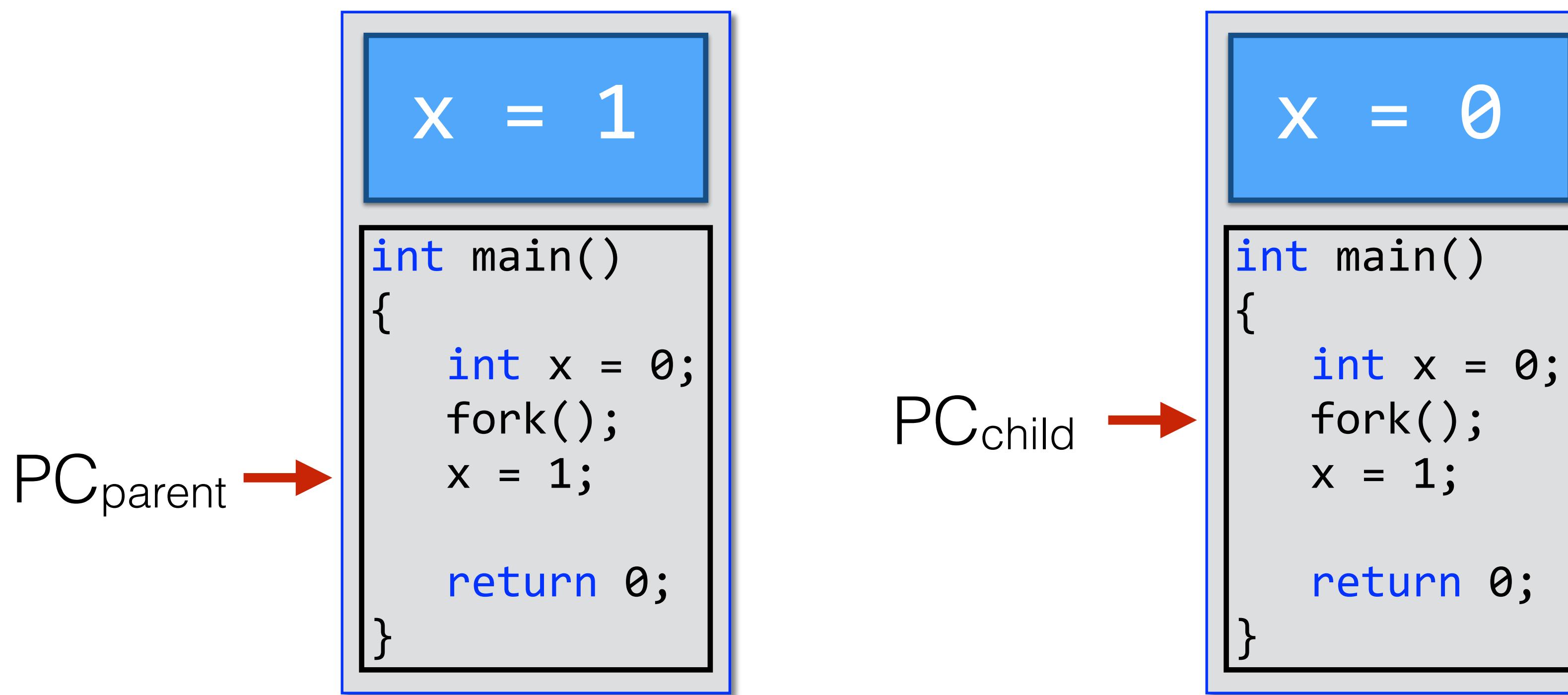
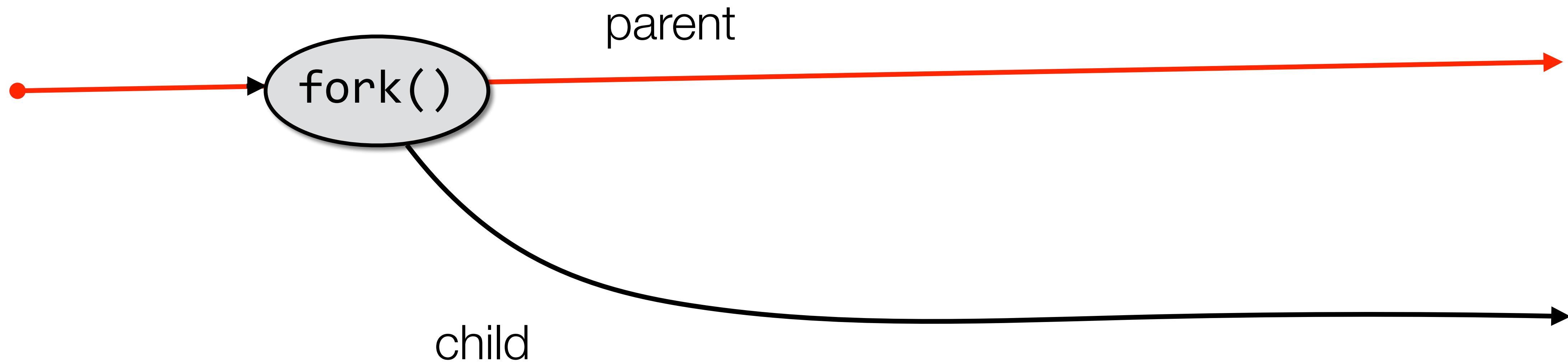
After Creation

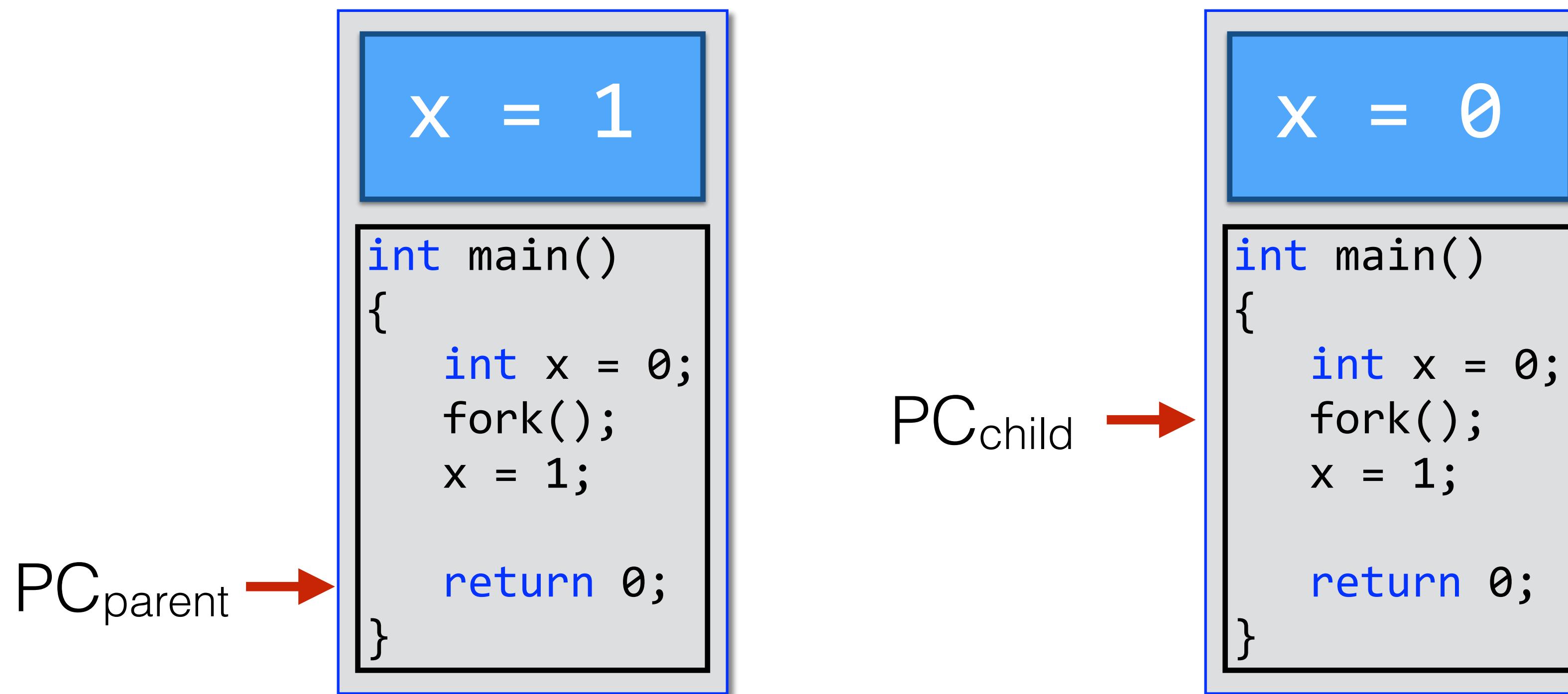
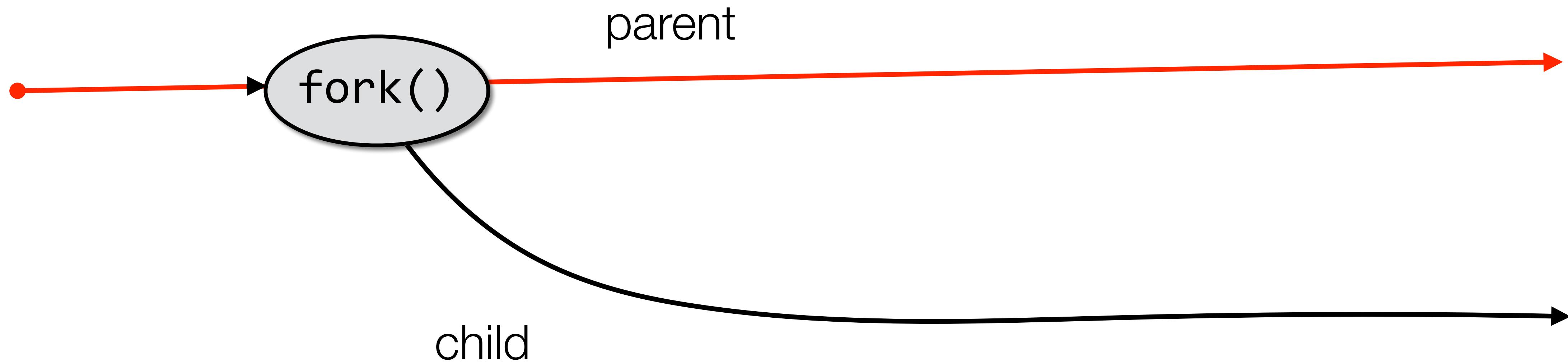
- After creating the process the Kernel can do one of the following, as part of the dispatcher routine:
 - Stay in the parent process.
 - Transfer control to the child process
 - Transfer control to another process.



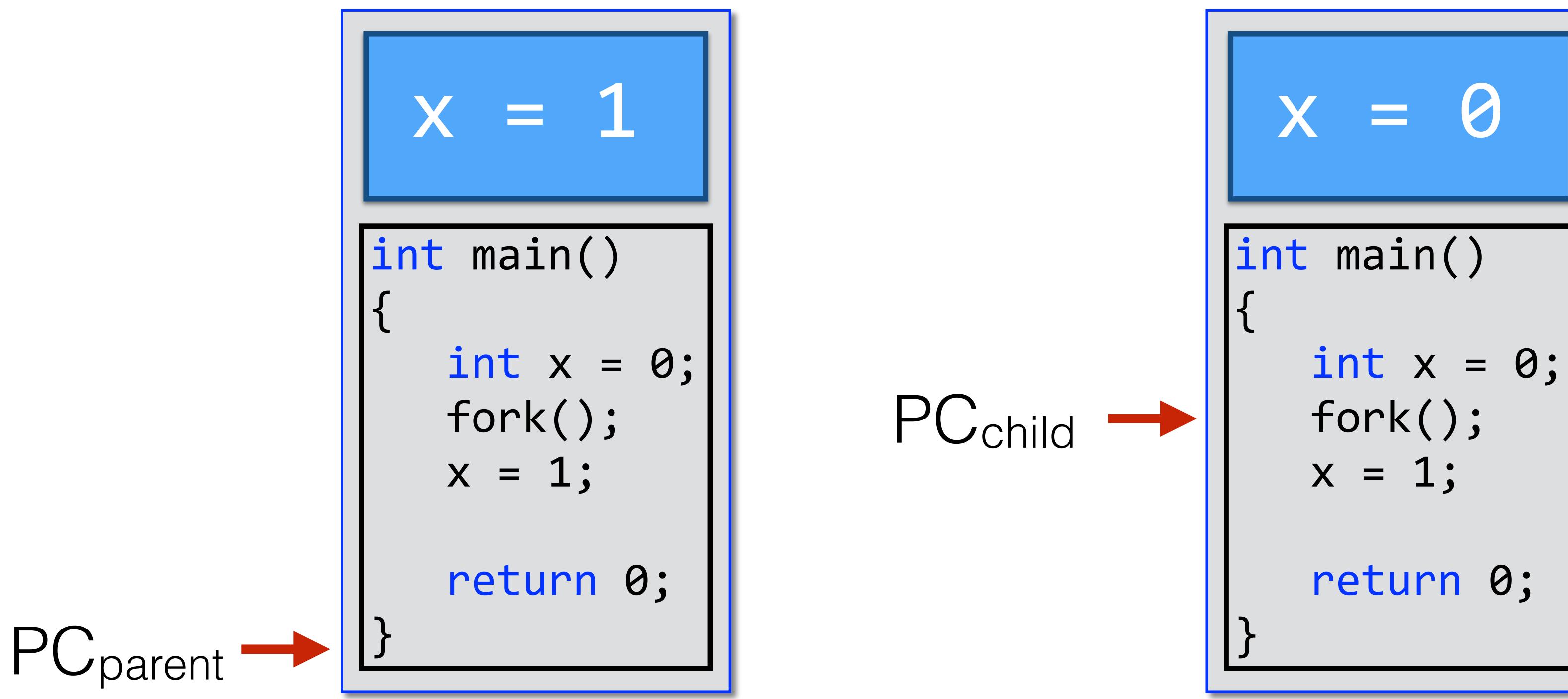
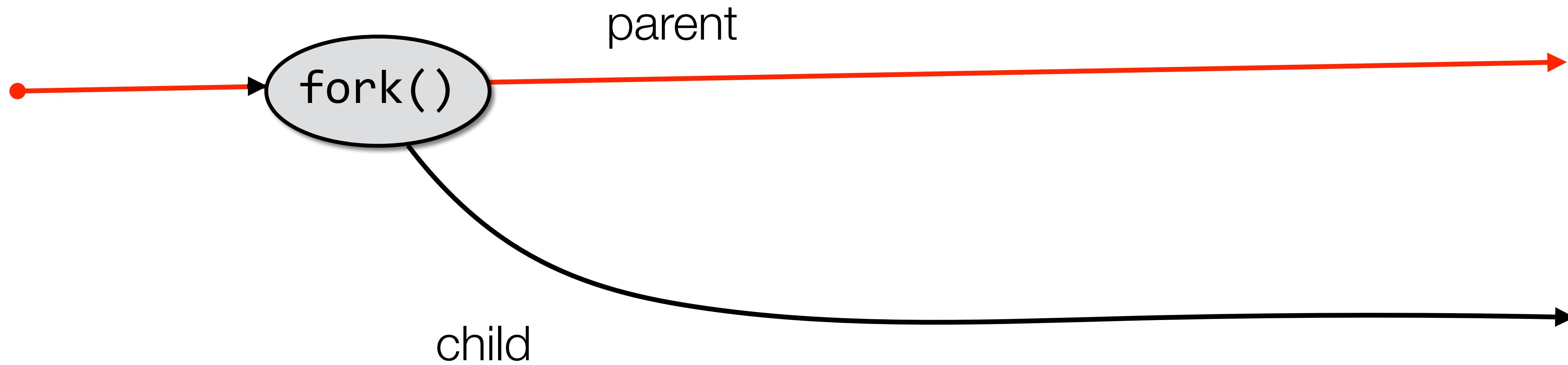
The `fork()` call is issued. A new process is created (child).

Assuming that the parent gets to run, what happens to it?

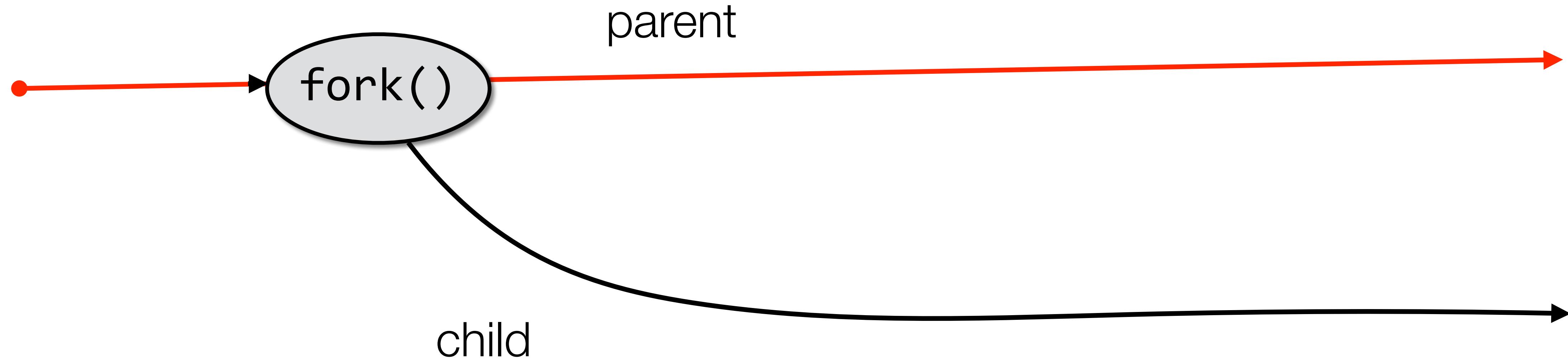




Why didn't the value of `x` in the child process change?



The parent process terminates. What happens to the child?



A code block showing the state of the child process. It contains a variable declaration `x = 0` in a blue box, followed by the C code for `main()`:

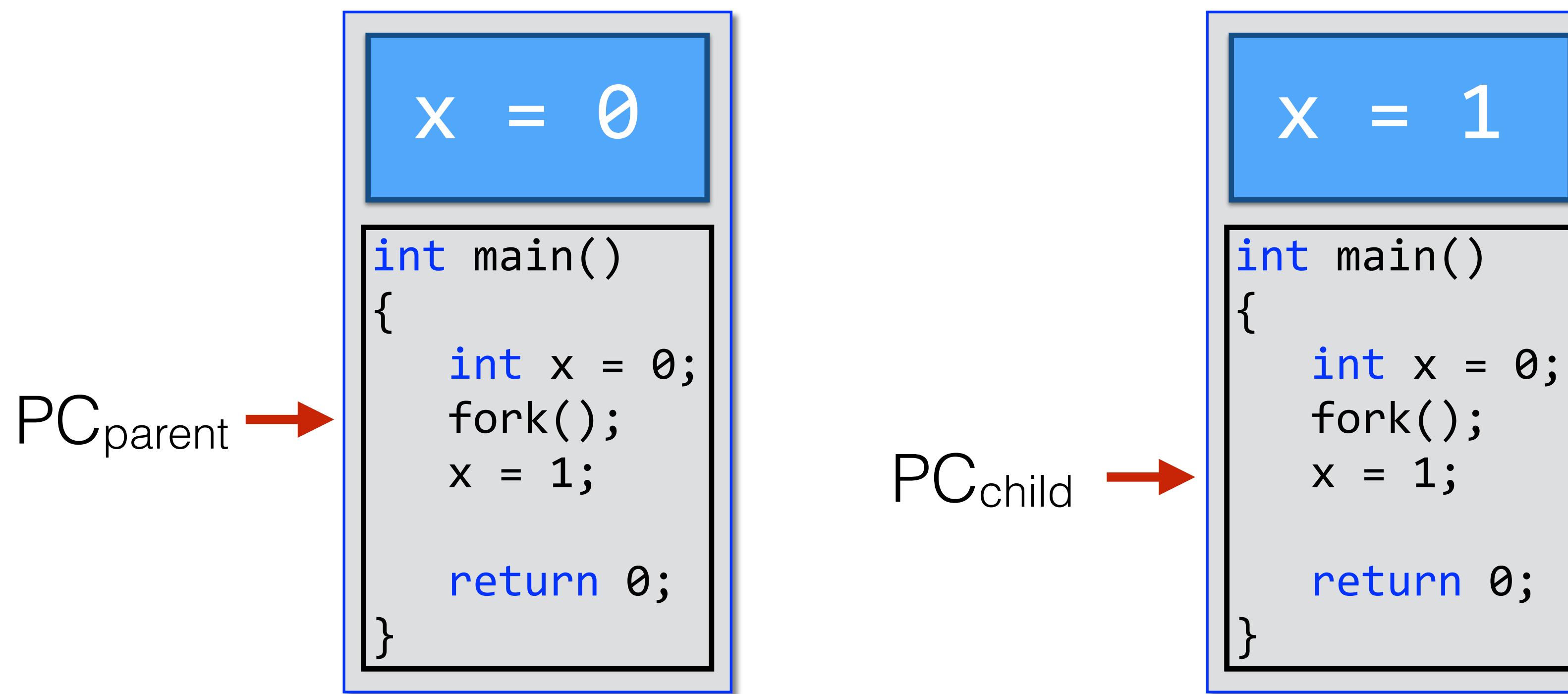
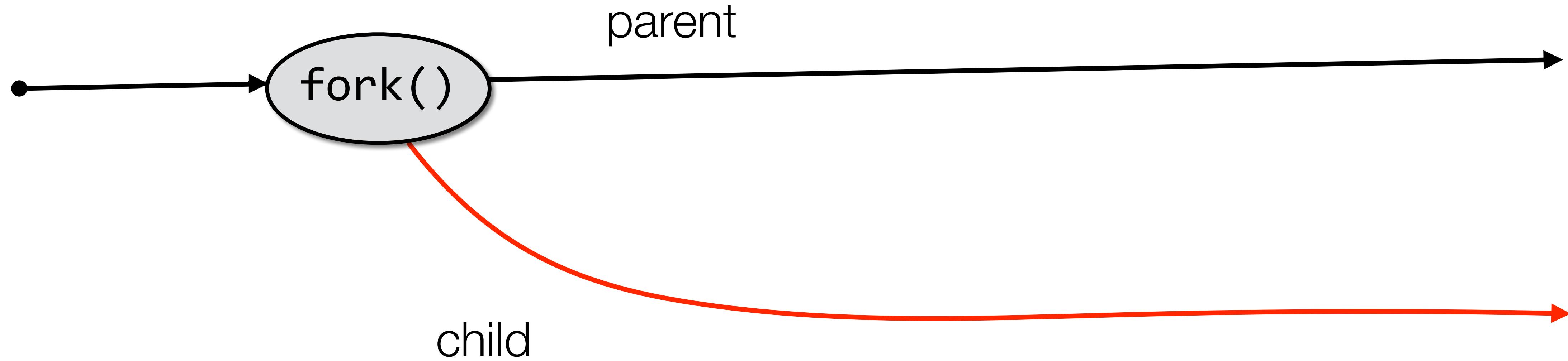
```
x = 0
int main()
{
    int x = 0;
    fork();
    x = 1;

    return 0;
}
```

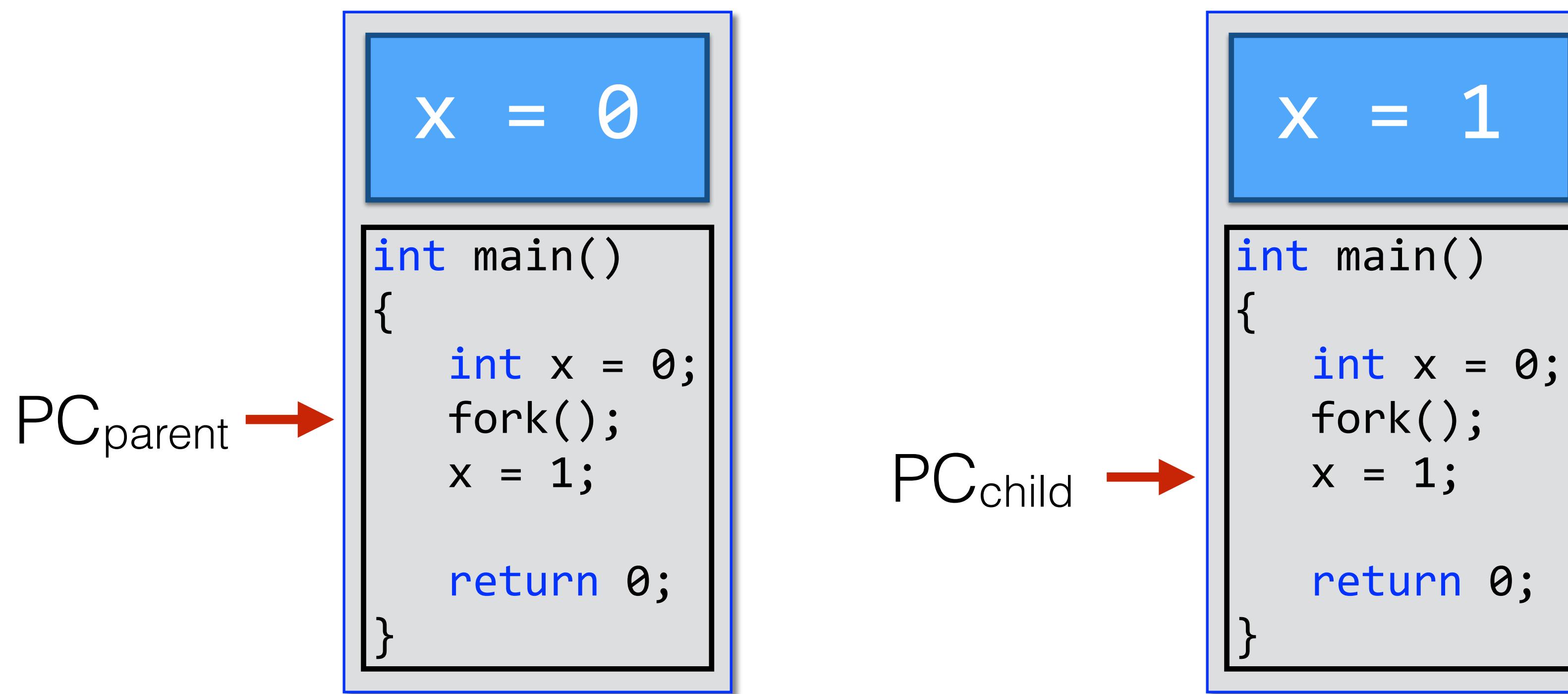
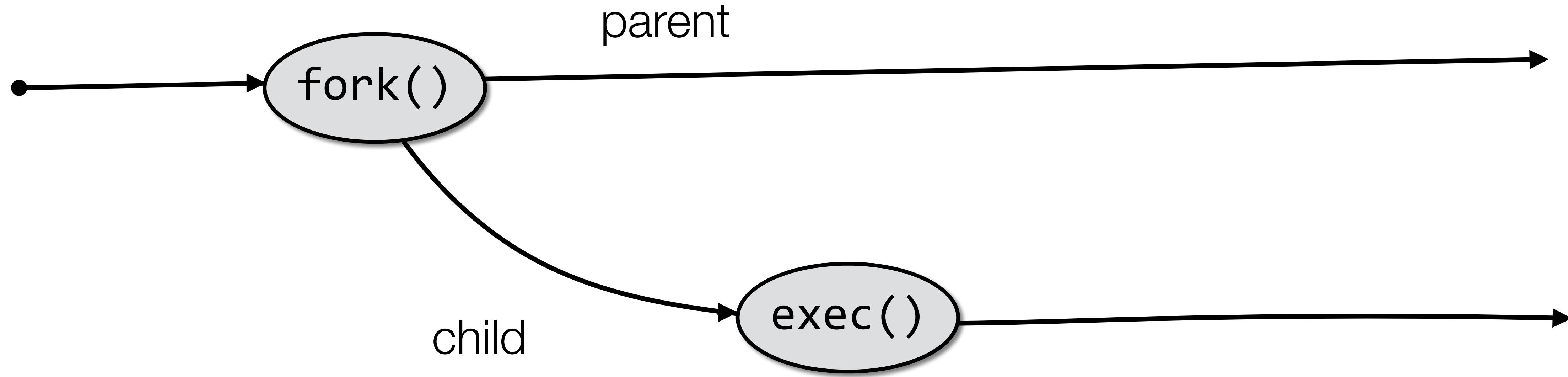
An arrow labeled PC_{child} points to the `fork();` line.

The parent process terminates. What happens to the child?

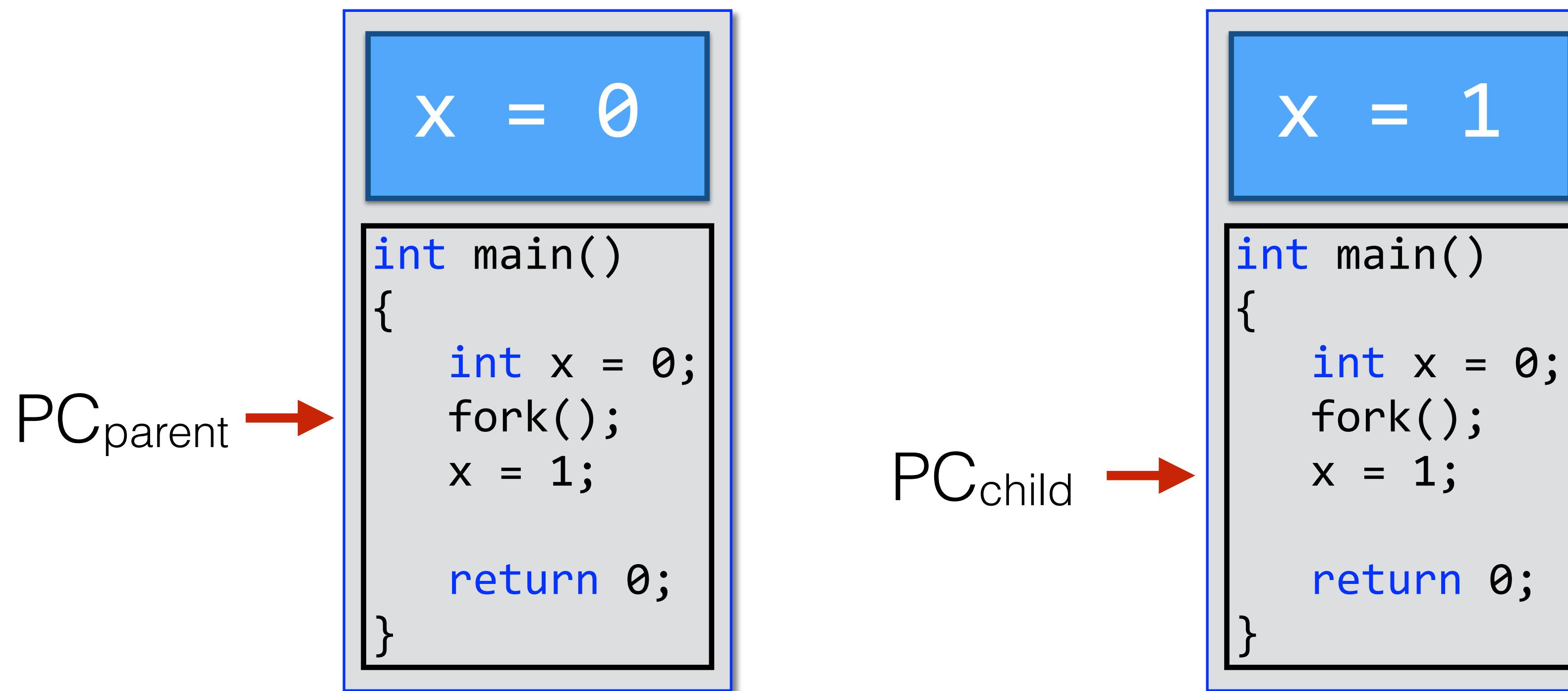
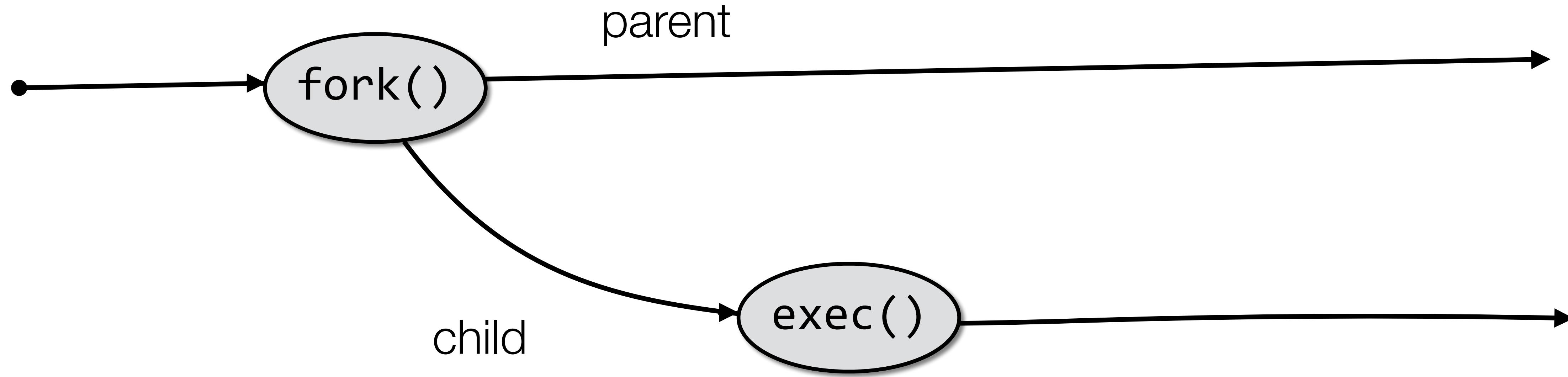
Who is the parent of the child now?



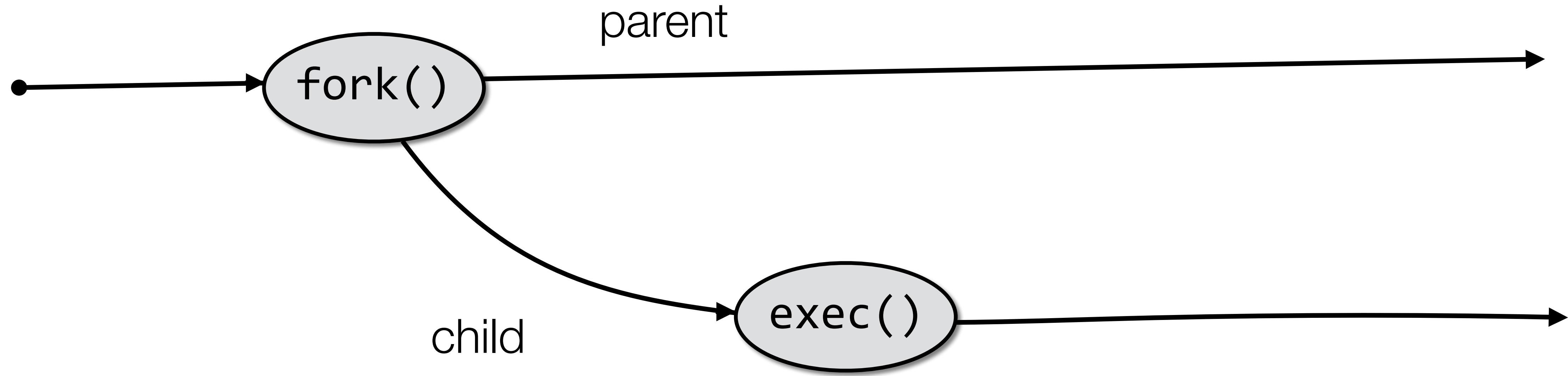
If the child process is scheduled to run, it executes just like the parent, i.e., it starts from the next instruction after the `fork()`. Then, it eventually terminates.



If we want the child process to execute code that is different from the parent's, we can call `exec()` in the child process.



If we want the child process to execute code that is different from the parent's, we can call `exec()` in the child process. **But how can we tell parent from child?**



But how can we tell parent from child?

x = 0

```

int main()
{
    int x = 0;
    pid_t r = fork();
    if(r==0)
        exec(ls);
    return 0;
}

```

PC_{parent}

x = 0

```

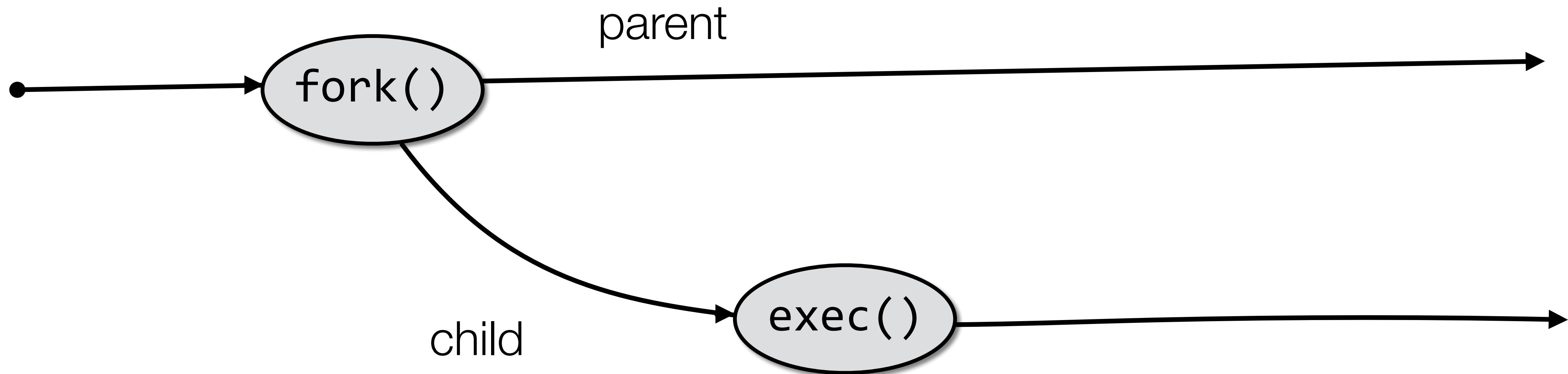
int main()
{
    int x = 0;
    pid_t r = fork();
    if(r==0)
        exec(ls);
    return 0;
}

```

PC_{child}

A red arrow points to the `fork()` call in both code snippets. In the parent's code, the condition `r == 0` is true, so it executes `exec(ls);`. In the child's code, the condition `r == 0` is false, so it does not execute `exec(ls);`.

`fork()` returns 0 to the child process and a non-zero value to parent process.



But how can we tell parent from child?

`x = 0, r=10`

```

int main()
{
    int x = 0;
    pid_t r = fork();
    if(r==0)
        exec(ls);
    return 0;
}

```

PC_{parent}

`x = 0, r=0`

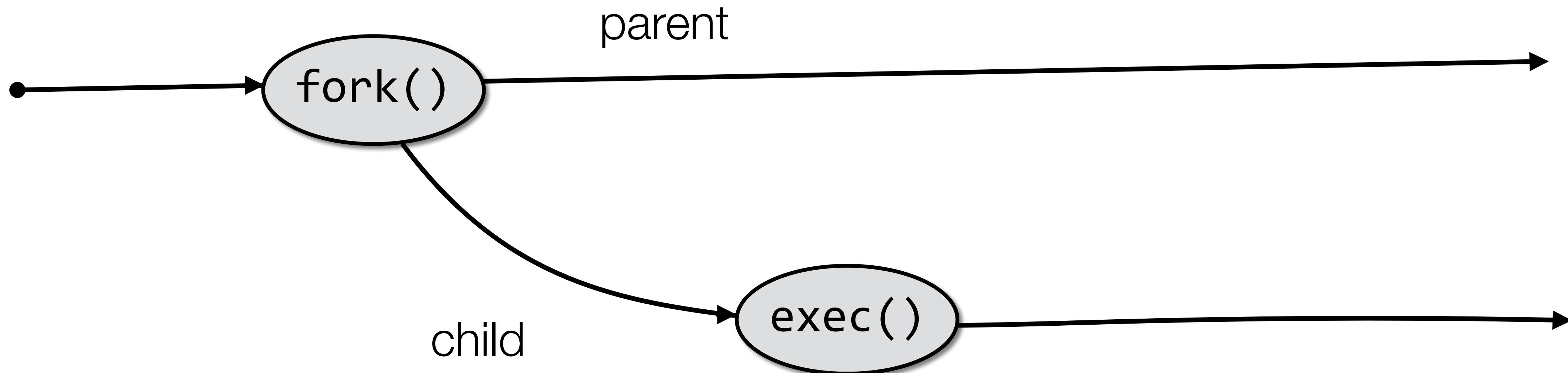
```

int main()
{
    int x = 0;
    pid_t r = fork();
    if(r==0)
        exec(ls);
    return 0;
}

```

PC_{child}

To the parent,
`fork()` returns the
 child's PID or a
 negative number
 (i.e., child couldn't
 be created).



PC_{parent}

```

x = 0, r=10
int main()
{
    int x = 0;
    pid_t r = fork();
    if(r==0)
        exec(ls);
    return 0;
}

```

PC_{child}

PC_{child} →

```

int main(args){
    struct file *flist=
    nil, **aflist=
    &flist;
    enum depth depth;

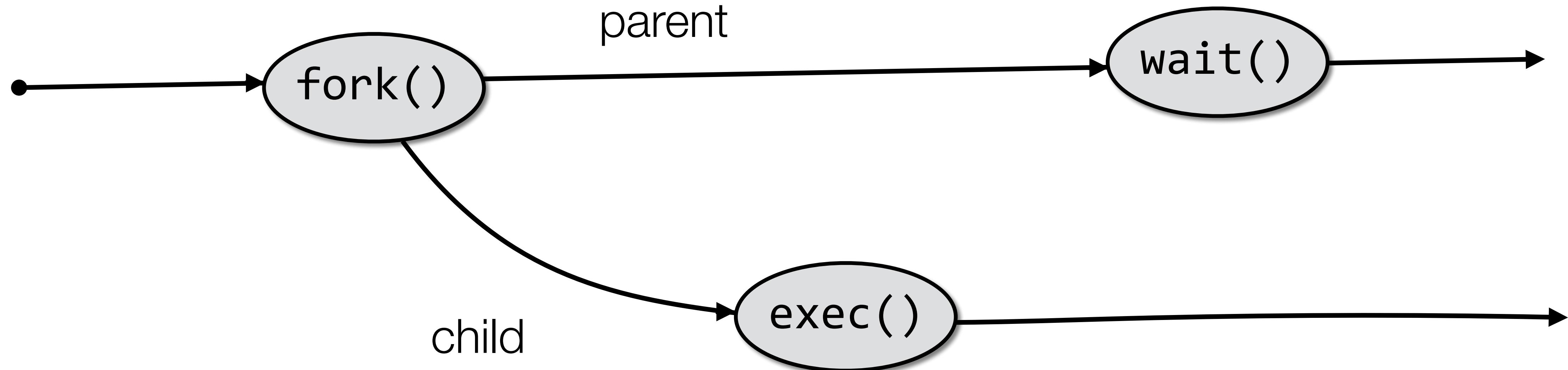
    (...)

}

```

The exec() function replaces the code section of the child process with the code of the new program.

The PC is reset to the first instruction.



`x = 0, r=10`

```

int main()
{
    int x = 0;
    pid_t r = fork();
    if(r==0)
        exec(ls);
    else
        wait();
    return 0;
}

```

`PCchild`

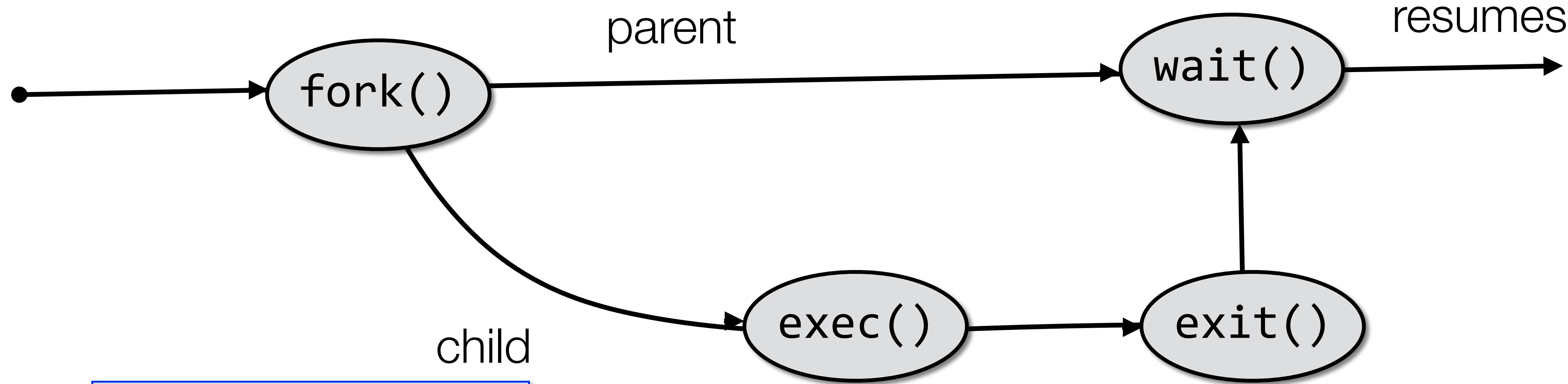
```

int main(args){
    struct file *flist=
    nil, **aflist=
    &flist;
    enum depth depth;

    (...)
}

```

Parent can issue a `wait()`. This will make parent wait until the child process terminates.



`x = 0, r=10`

```

int main()
{
    int x = 0;
    pid_t r = fork();
    if(r==0)
        exec(ls);
    else
        wait();
    return 0;
}
  
```

PC_{child}

```

int main(args){
    struct file *flist=
    nil, **aflist=
    &flist;
    enum depth depth;

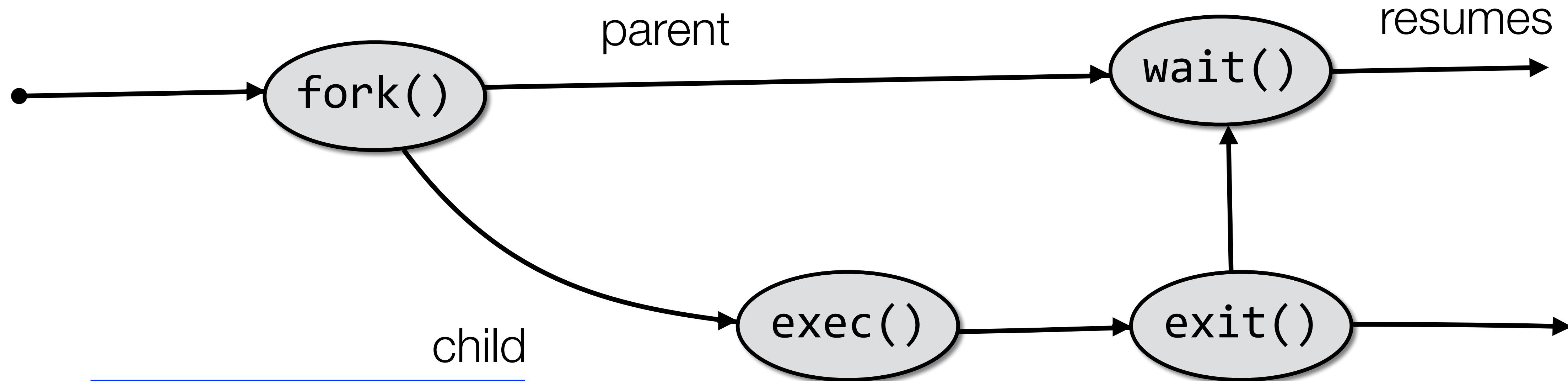
    (...)

}
  
```

Parent can issue a `wait()`. This will make parent wait until the child process terminates.

PC_{parent}

A callout box containing the code for the parent process, with a red arrow pointing to the `wait()` statement.



`x = 0, r=10`

```

int main()
{
    int x = 0;
    pid_t r = fork();
    if(r==0)
        exec(ls);
    else
        wait();
    return 0;
}
  
```

PC_{child}

```

int main(args){
    struct file *flist=
    nil, **aflist=
    &flist;
    enum depth depth;

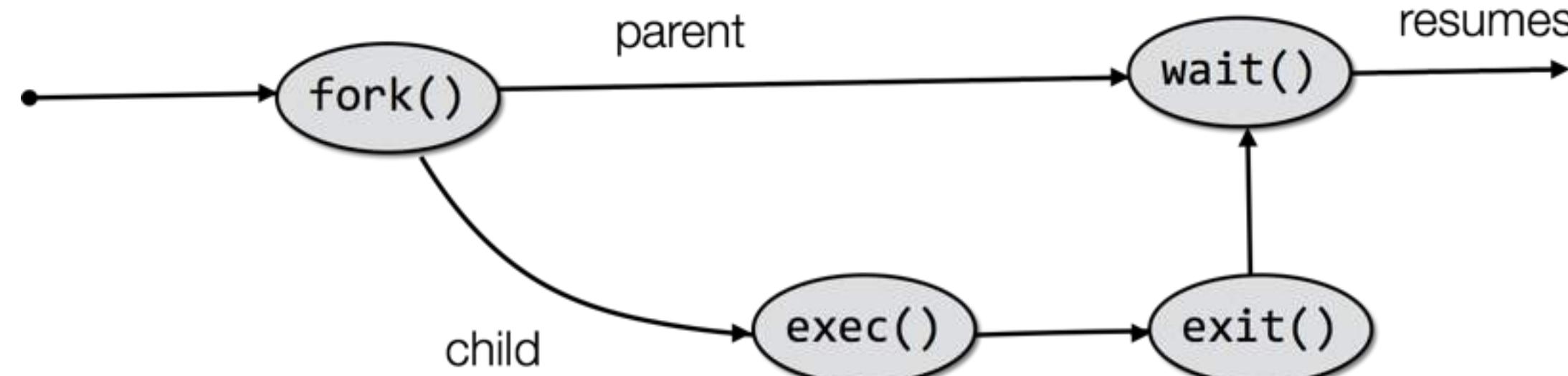
    (...)

}
  
```

PC_{parent}

What are the advantages of this apparently complex interface?

Standard fork pattern



```
int main()
{
pid_t pid;
/* fork another process */
pid = fork();
if (pid < 0) { /* error occurred */
    fprintf(stderr, "Fork Failed");
    exit(-1);
}
else if (pid == 0) { /* child process */
    execlp("/bin/ls", "ls", NULL);
}
else { /* parent process */
    /* parent will wait for the child to
complete */
    wait(NULL);
    printf ("Child Complete");
    exit(0);
}
}
```

Process Creation in Unix

- Process creation is by means of the system call fork().
- This causes the OS, in Kernel Mode, to:
 1. Allocate a slot in the process table for the new process.
 2. Assign a unique process ID to the child process.
 3. Copy of process image of the parent, with the exception of any shared memory.
 4. Increment the counters for any files owned by the parent, to reflect that an additional process now also owns those files.
 5. Assign the child process to the Ready to Run state.
 6. Returns the ID number of the child to the parent process, and a 0 value to the child process.

Why have fork() at all?

Why make a copy of the parent process?

Don't you usually want to start a new program instead?

Where might “cloning” the parent be useful?

- Web server – make a copy for each incoming connection
- Parallel processing – set up initial state, fork off multiple copies to do work

UNIX philosophy: System calls should be minimal.

- Don't overload system calls with extra functionality if it is not always needed.
- Better to provide a flexible set of simple primitives and let programmers combine them in useful ways.

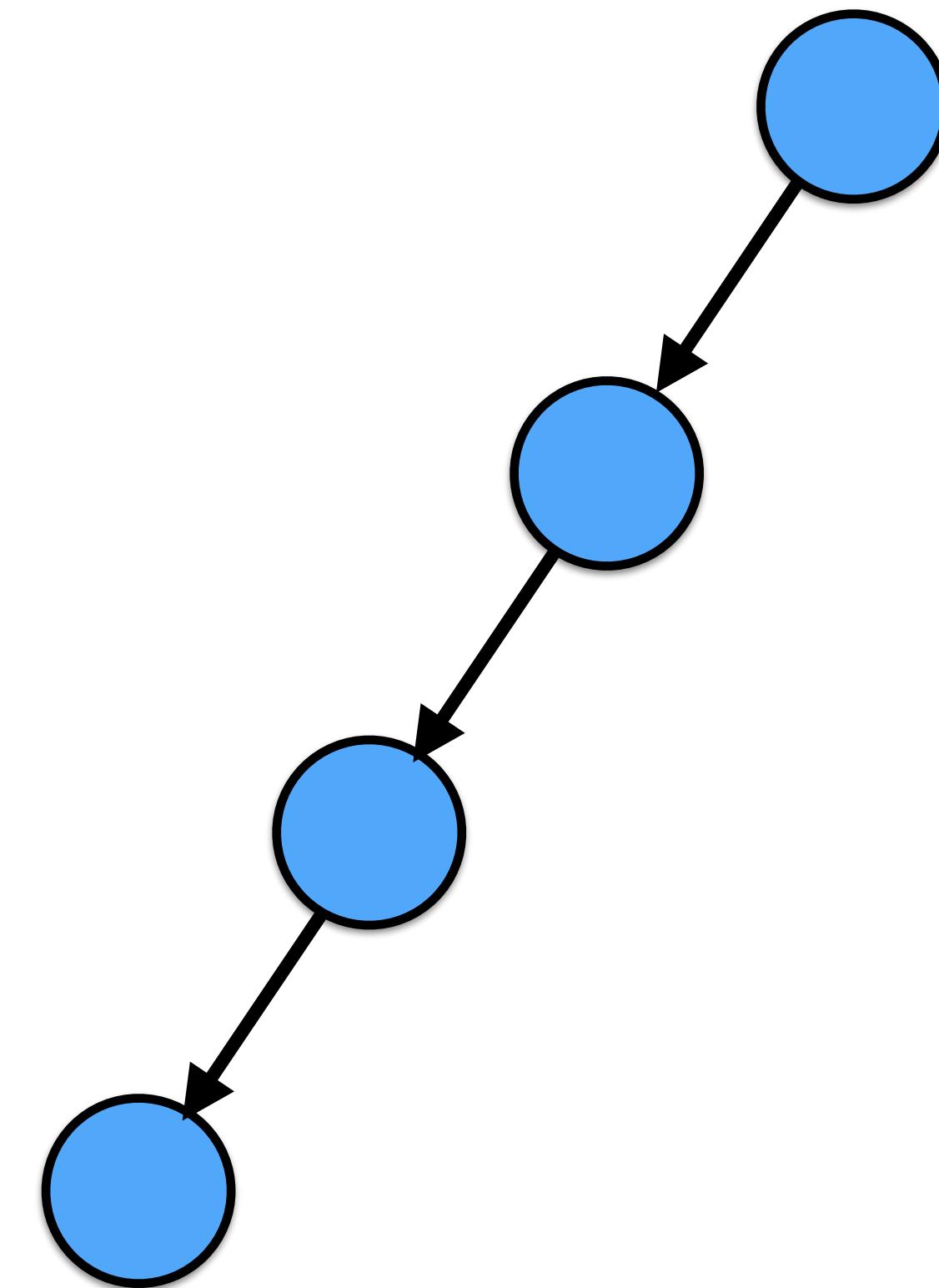
Output of sample program

```
Process 4530: value is 0
Process 4530: value is 1
Process 4530: value is 2
Process 4530: value is 3
Process 4530: value is 4
Process 4530: value is 5
Process 4530: About to do a fork...
Process 4531: value is 6
Process 4530: value is 6
Process 4530: value is 7
Process 4531: value is 7
Process 4530: value is 8
Process 4531: value is 8
Process 4530: value is 9
Process 4531: value is 9
```

*What determines the order in which
the two processes run???*

queue of processes

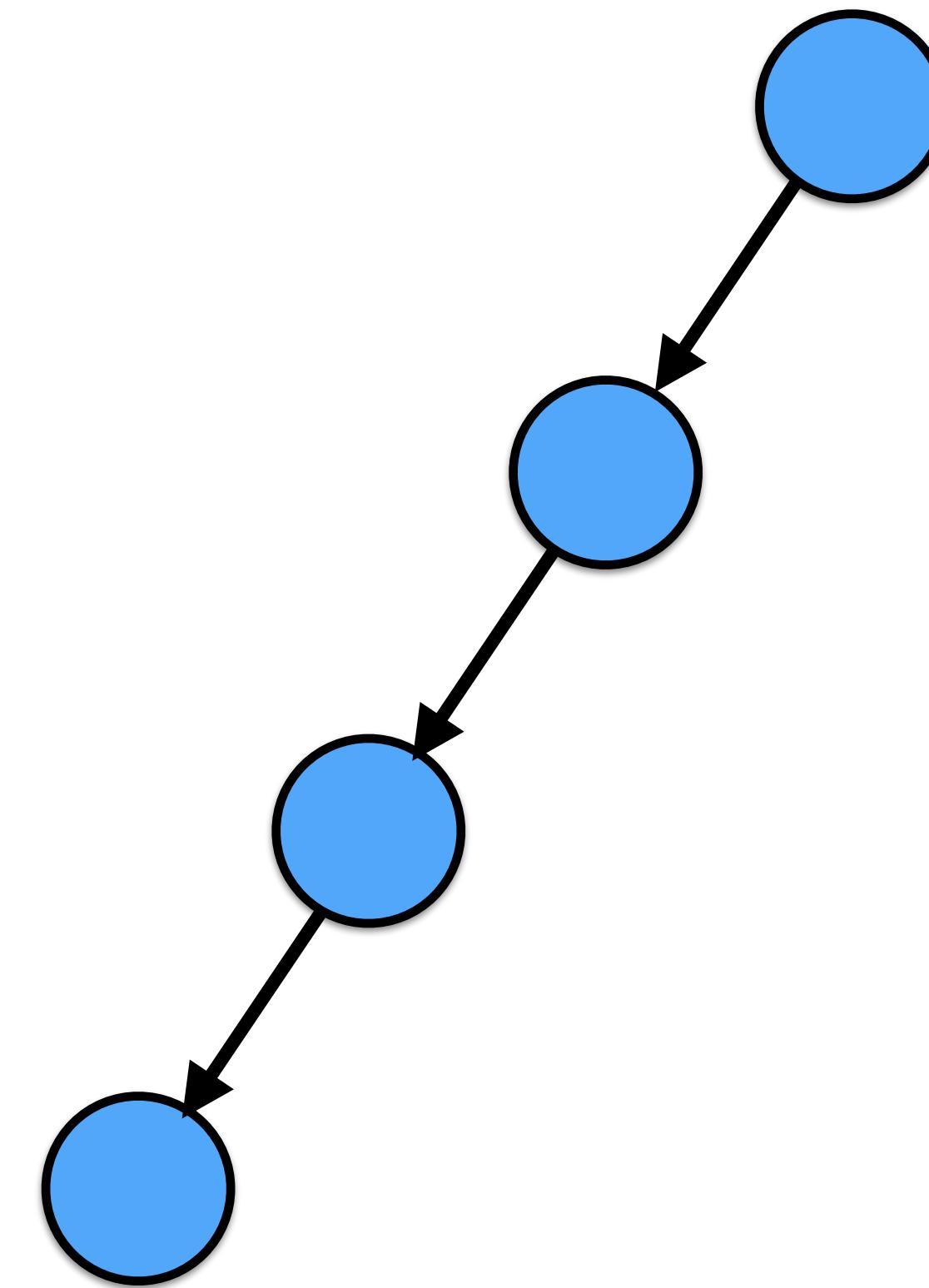
```
int i, n = 4;  
pid_t pid;  
for ( i = 1; i < n; ++i )  
    if ( pid = fork() )  
        break;
```



Parent breaks because when the child process is created, `fork()` returns the PID of the child to the parent process. The PID of any process is nonzero.

queue of processes

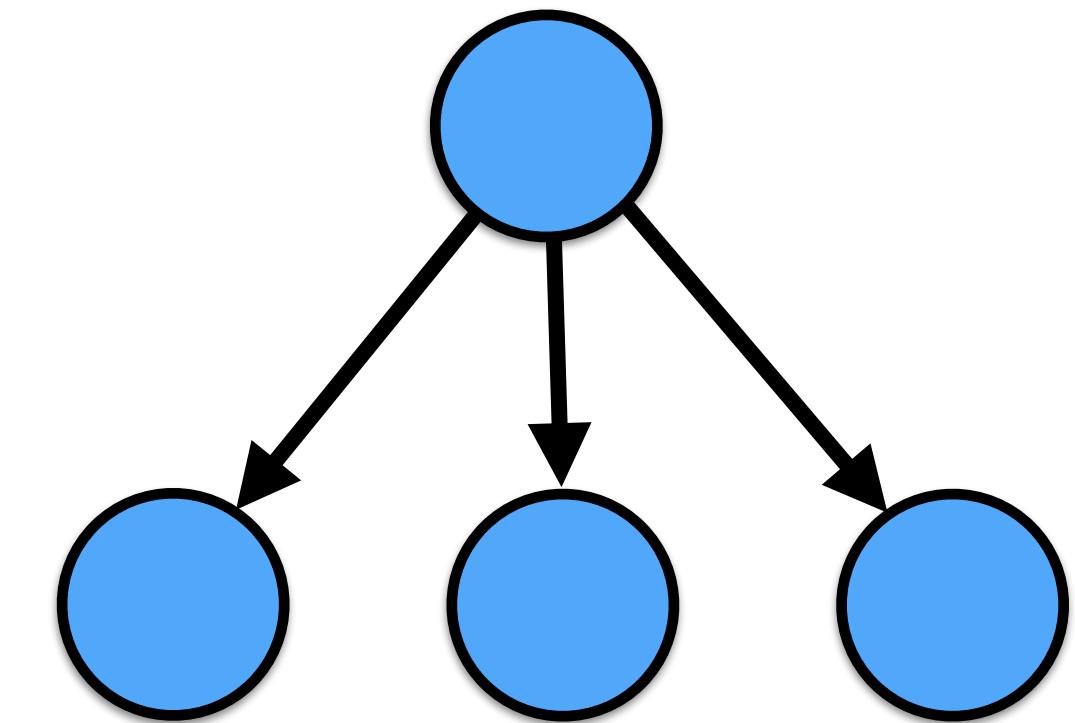
```
int i, n = 4;  
pid_t pid;  
for ( i = 1; i < n; ++i )  
    if ( pid = fork() )  
        break;
```



Child processes run the for-loop because `fork()` returns 0 (zero) to the child process.

fan of processes

```
int i, n = 4;  
pid_t pid;  
for ( i = 1; i < n; ++i )  
    if ( ( pid = fork() ) <= 0 )  
        break;
```



This time, child processes break, and parent runs the loop.

exit()

- When calling **exit()**, a process voluntarily release all its resources, e.g., address space returned, files closed, etc.
- But not everything can be cleaned by the process itself, it has to be cleaned by someone else.
- Also, I **should not** be all cleaned by the process because the parent process may be waiting for the return value.
- So, there is a separate state for the period after the process calls exit() and before someone else cleans it up.
- In Unix, it's called **zombie** state.

Zombie

- When a process exits, almost all of its resources are deallocated
- Address space is returned, files are closed, etc.
- PCB retains information about the process's exit state
- The process retains its PID
- The process is a **zombie** until its parent cleans it up



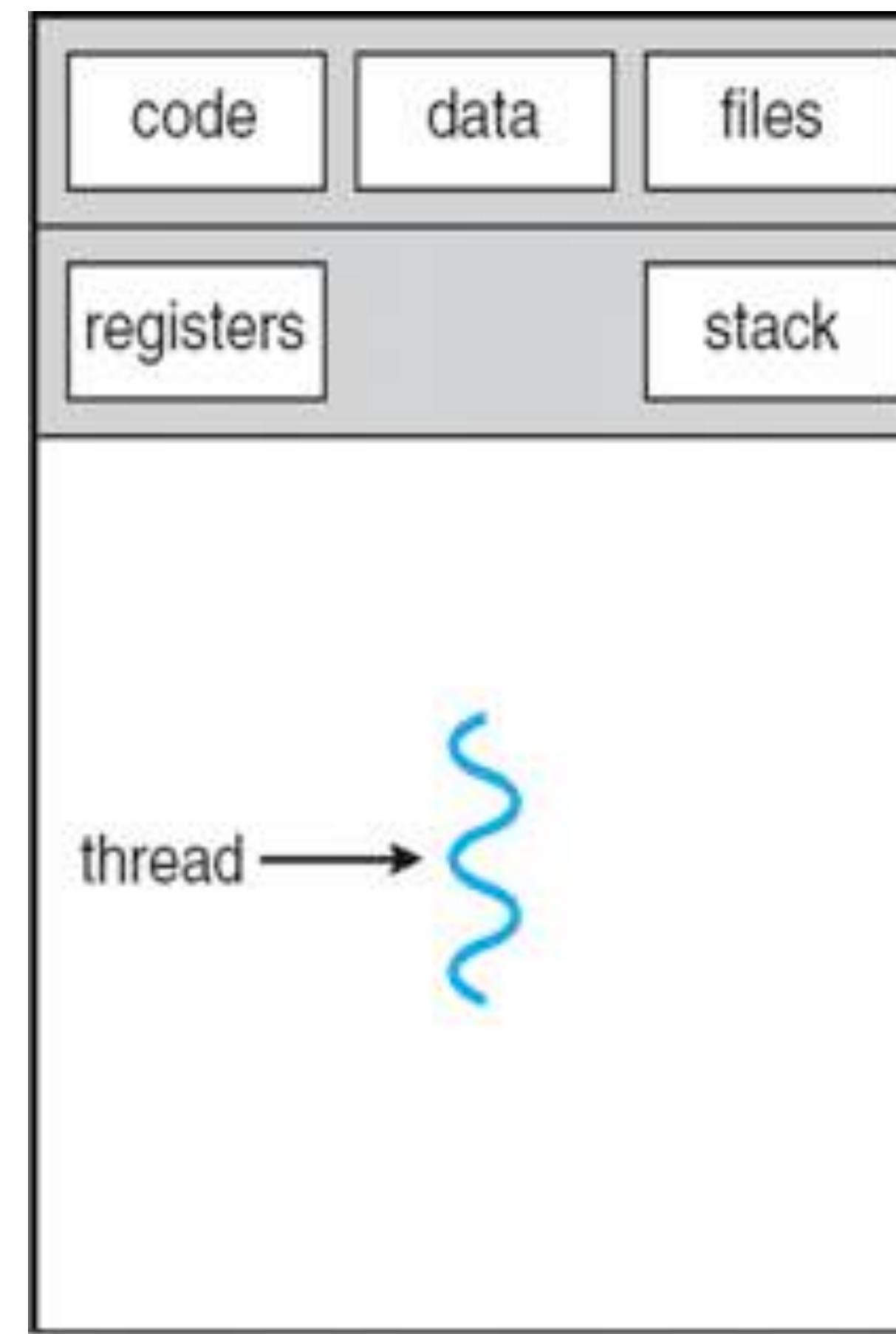
Threads

CSE4001 Operating Systems Concepts

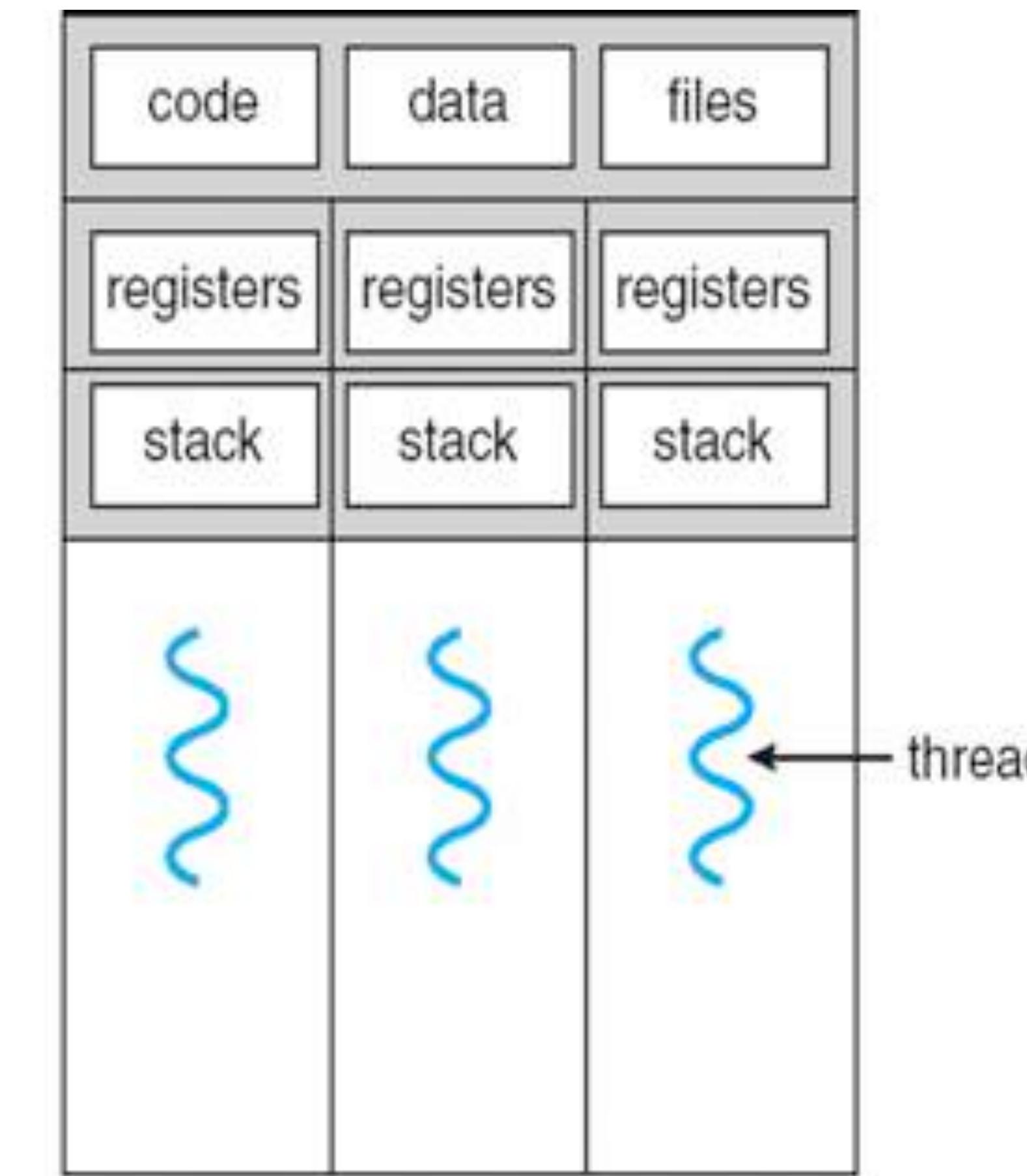
E. Ribeiro

Single and Multithreaded Processes

- ▶ **Thread** – a fundamental unit of CPU utilization that forms the basis of multithreaded computer systems



single-threaded process



multithreaded process

Threads

- » Threads are discrete processing units that allow functions to execute concurrently (i.e., simultaneous execution of functions while taking turns in the CPU).
- » Useful when functions take too long to complete their tasks as they should not block other functions.
- » When an application is launched, it contains only one thread (i.e., executes the `main()` function). This type of application is called a *single-threaded application*.

Threads

- » *Multi-threaded applications* create new threads to execute multiple functions.
- » Modern computer architecture offers multiple processing cores by default. Threads allow programmers to use the available processing capacity.
- » Having multi-core machines by default means that knowing how to develop multi-threaded programs has become a key skill in modern programming.

Single and Multithreaded Processes

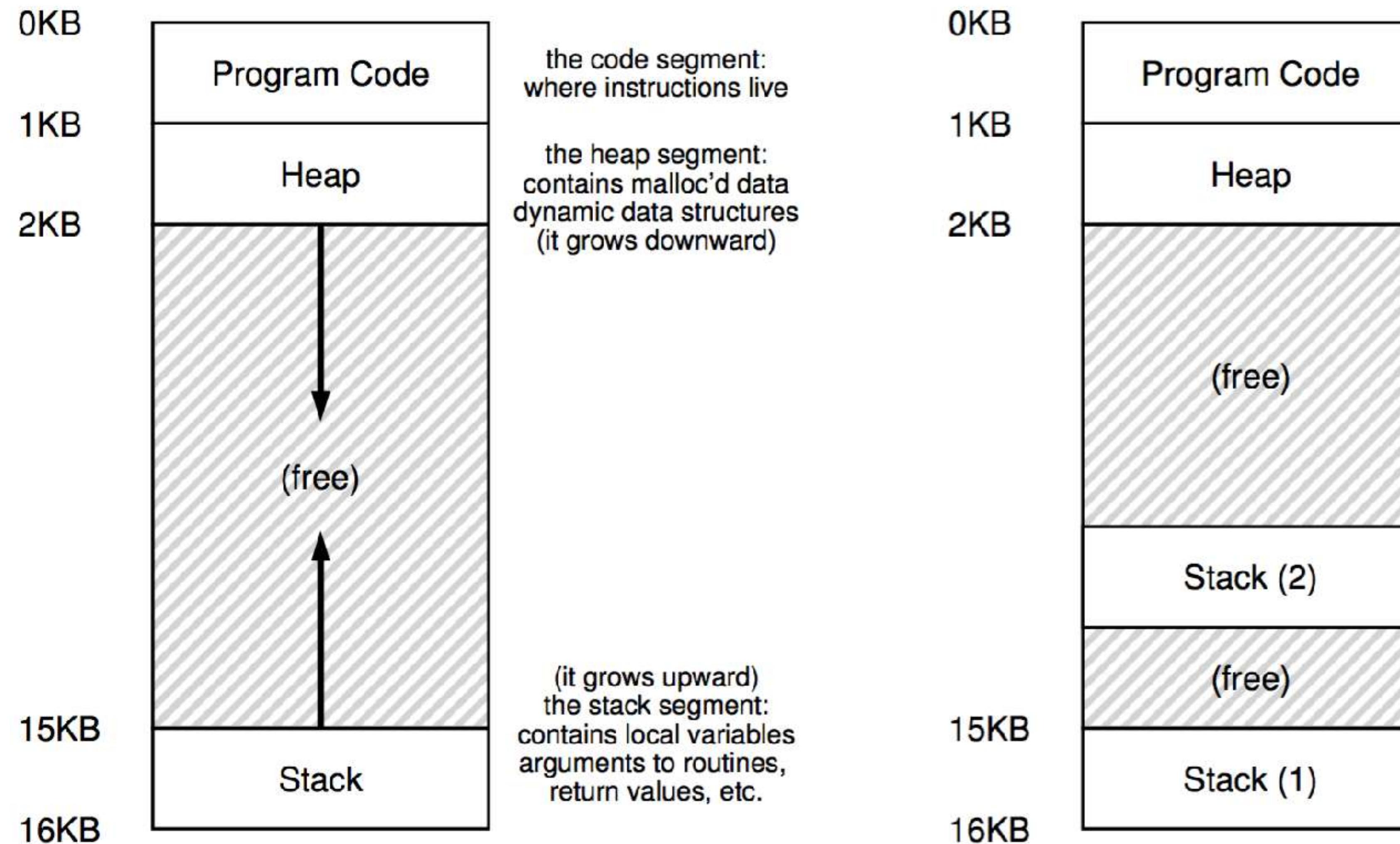
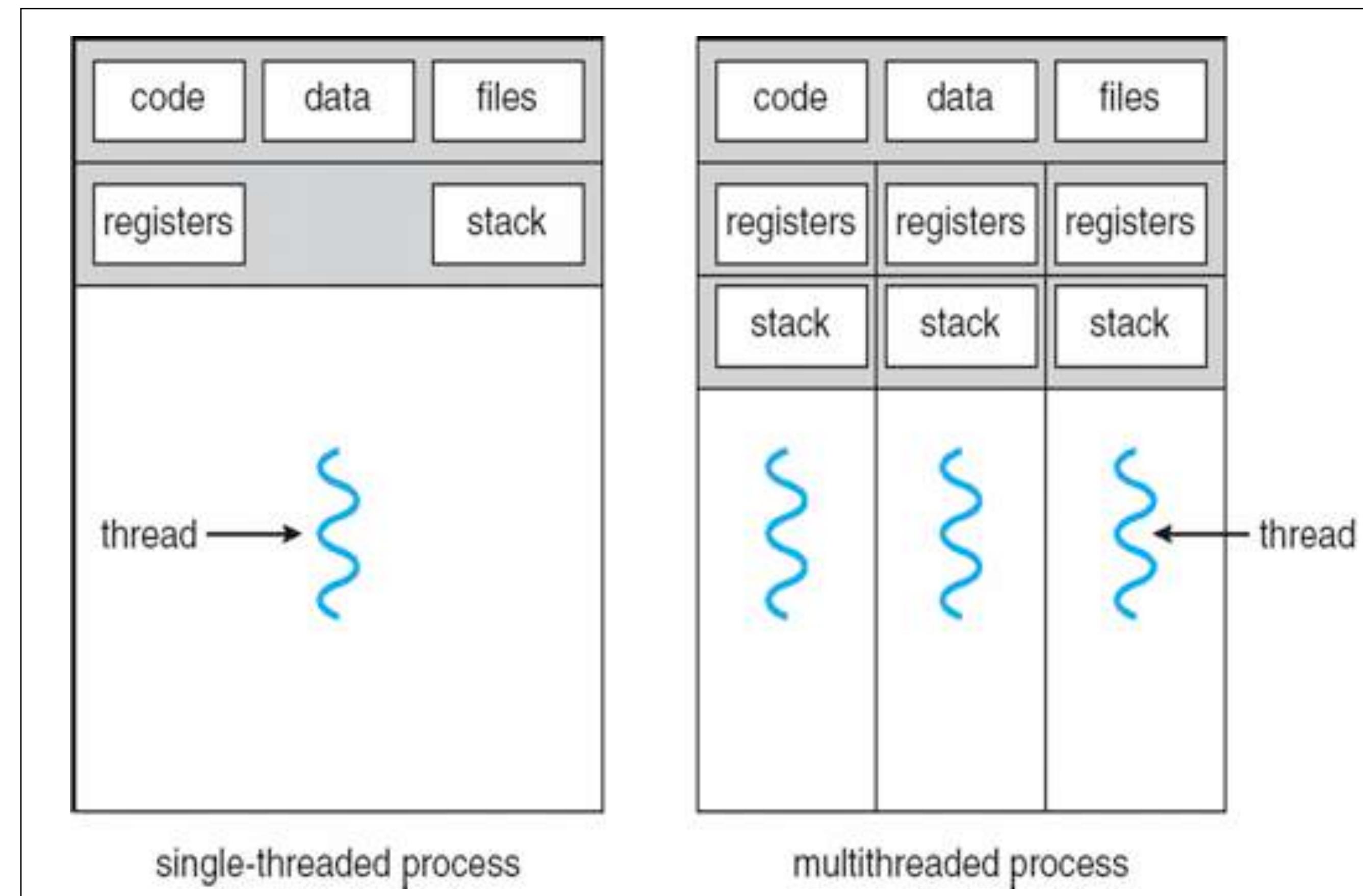


Figure 26.1: Single-Threaded And Multi-Threaded Address Spaces

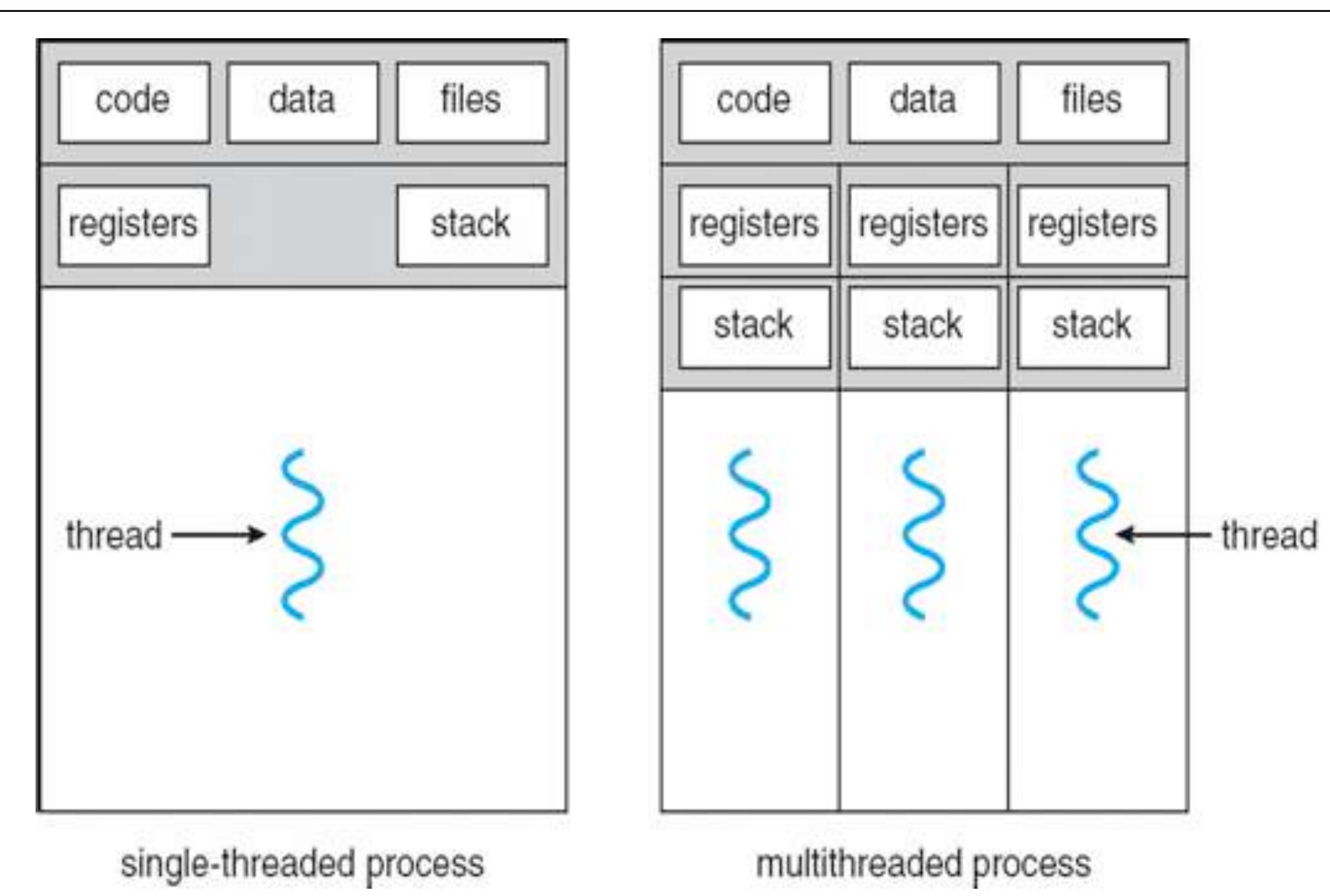
Single and Multithreaded Processes

- A thread is a basic unit of CPU utilization. It comprises:
 - a thread ID
 - a program counter
 - a register set
 - a stack



Single and Multithreaded Processes

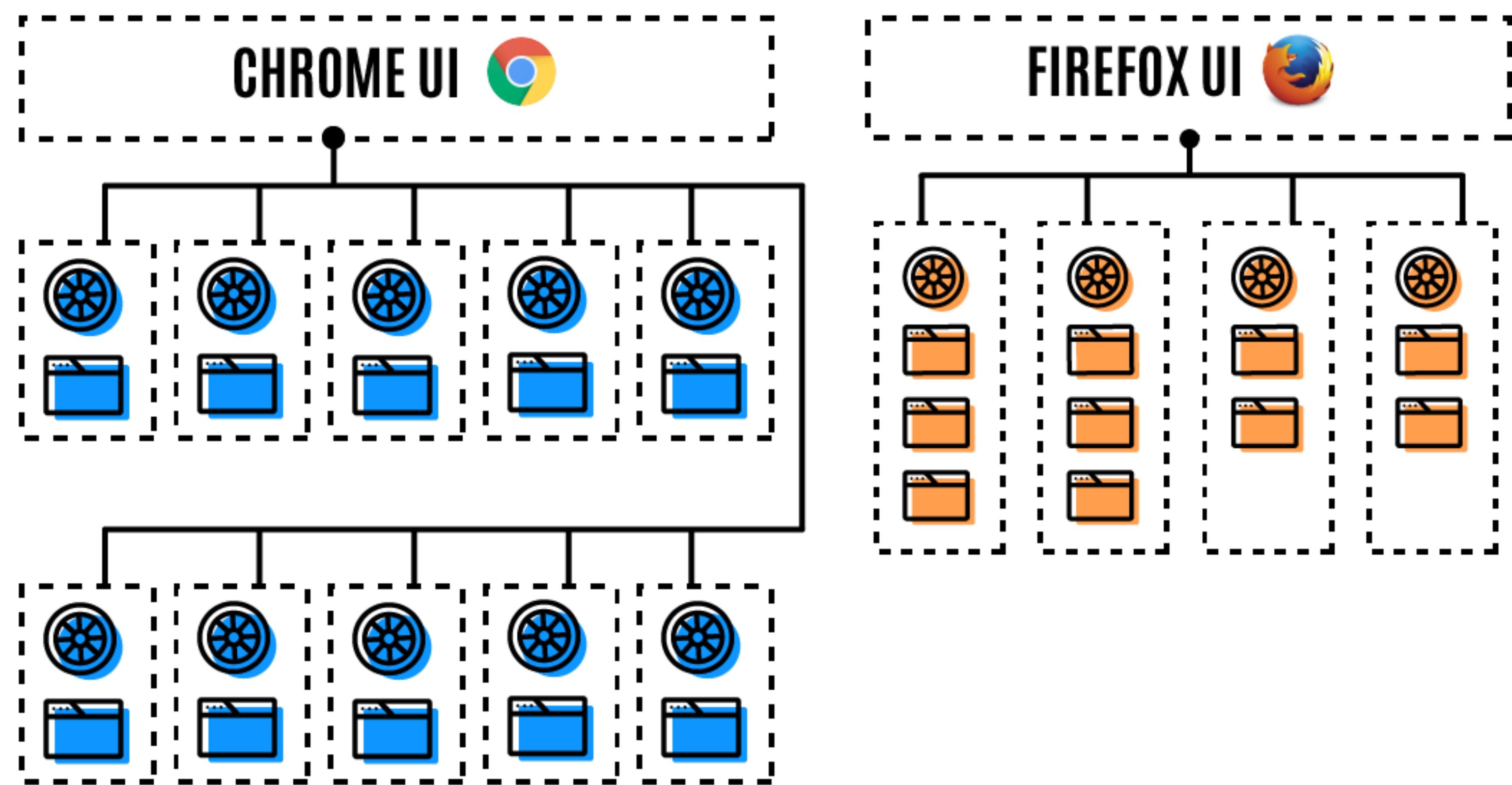
- A traditional process has a single thread of control.
- Processes that have multiple threads can perform multiple tasks concurrently.
- Software packages are usually multithreaded. They are implemented as a process with several threads of control.



Some multi-threaded packages

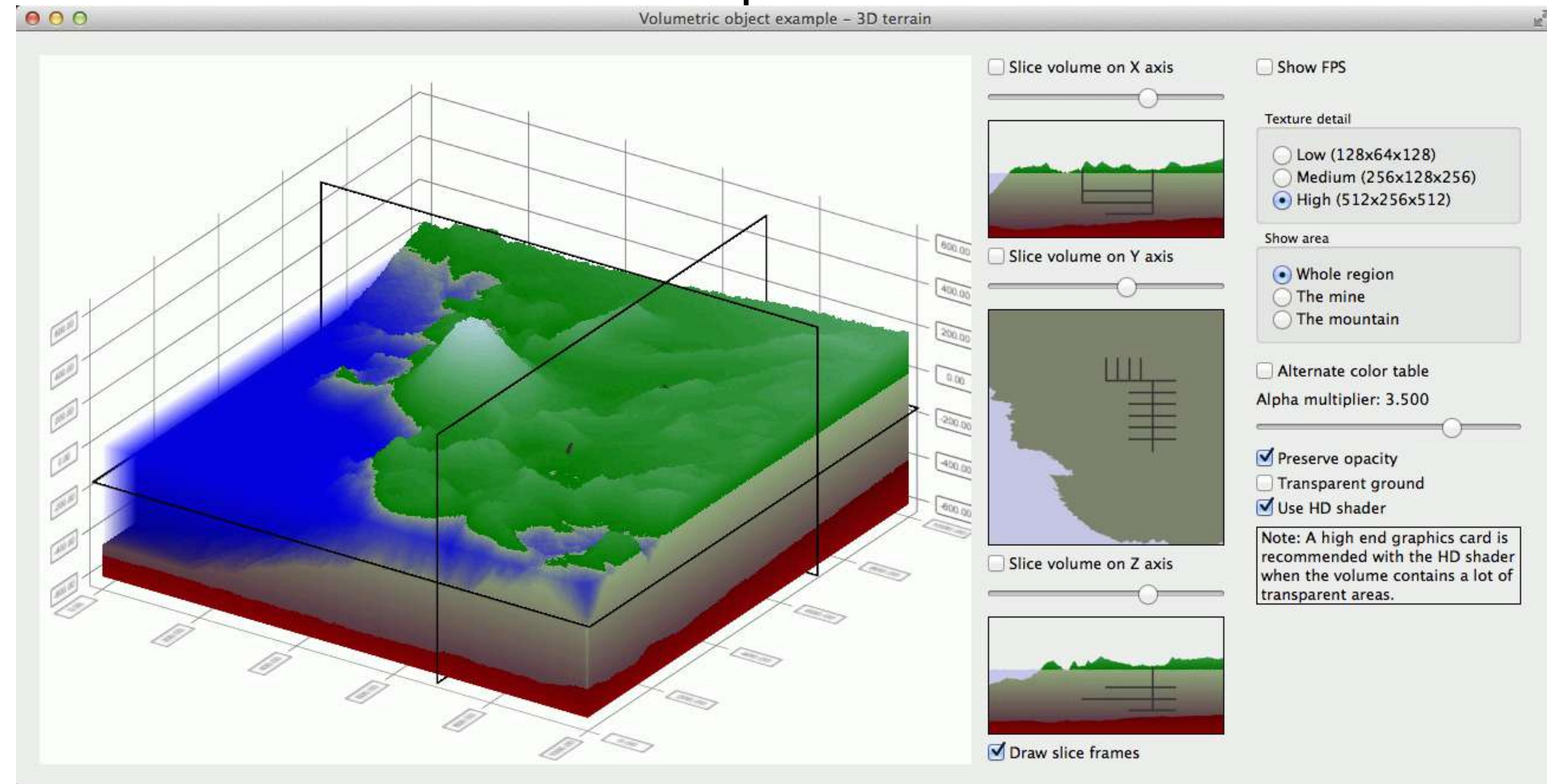
A web browser has threads for showing text, threads for showing images, threads to retrieve data from the network.

BROWSER ARCHITECTURE

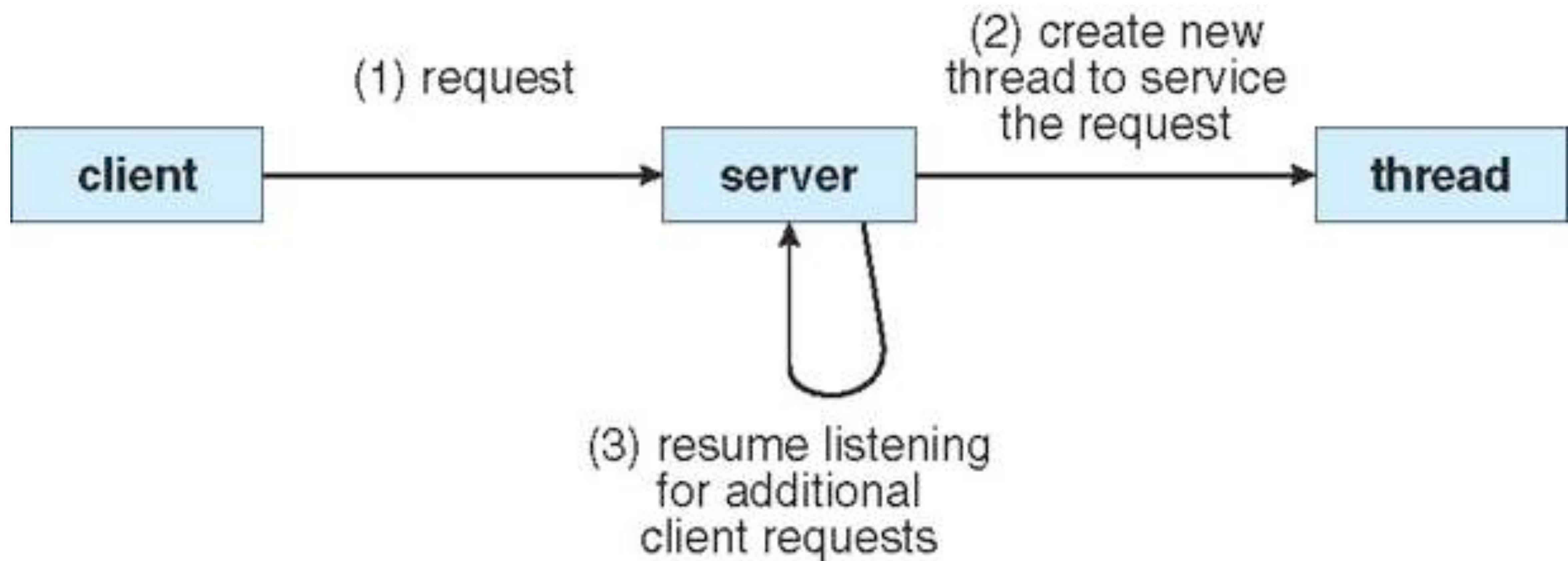


Some multi-threaded packages

Most graphical user interfaces are multi-threaded programs.
 Scientific software also uses multiple threads

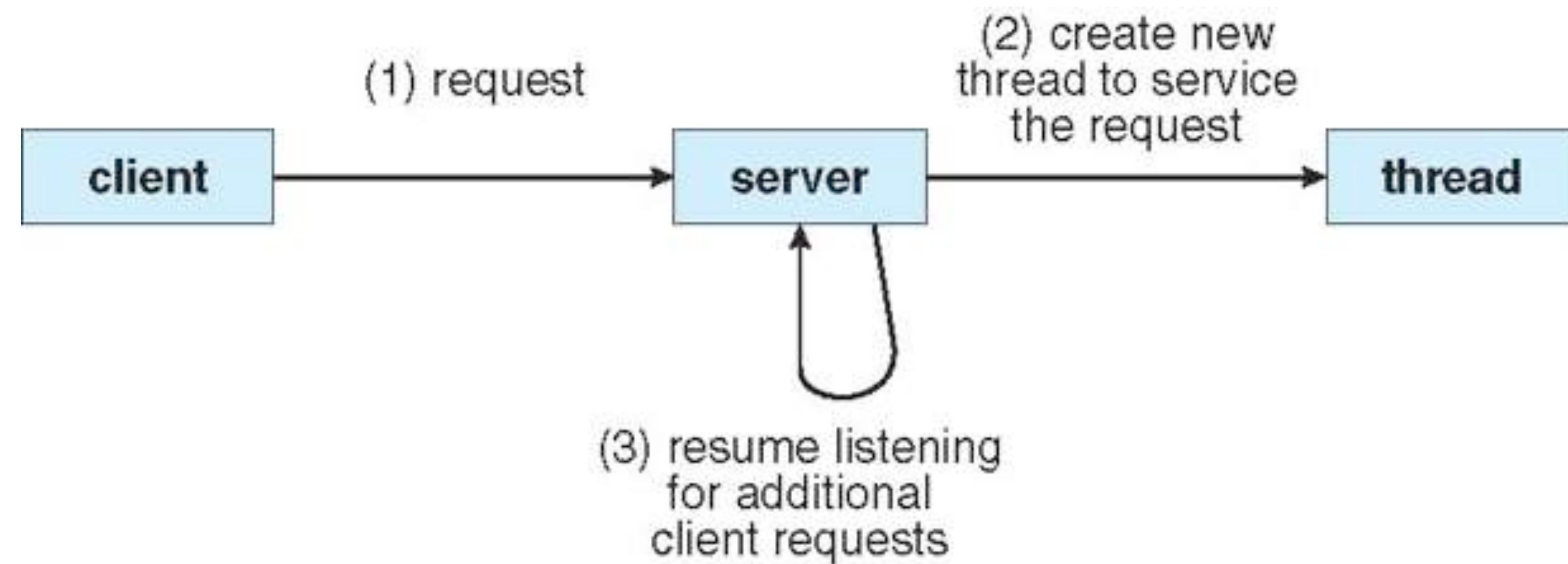


Multithreaded Server Architecture



Multithreaded Server Architecture

- For a large number of clients, a single-threaded server implementation would take too long to respond.
- Processes were used often to solve this problem until threads became popular. Threads are known as light-weight processes.



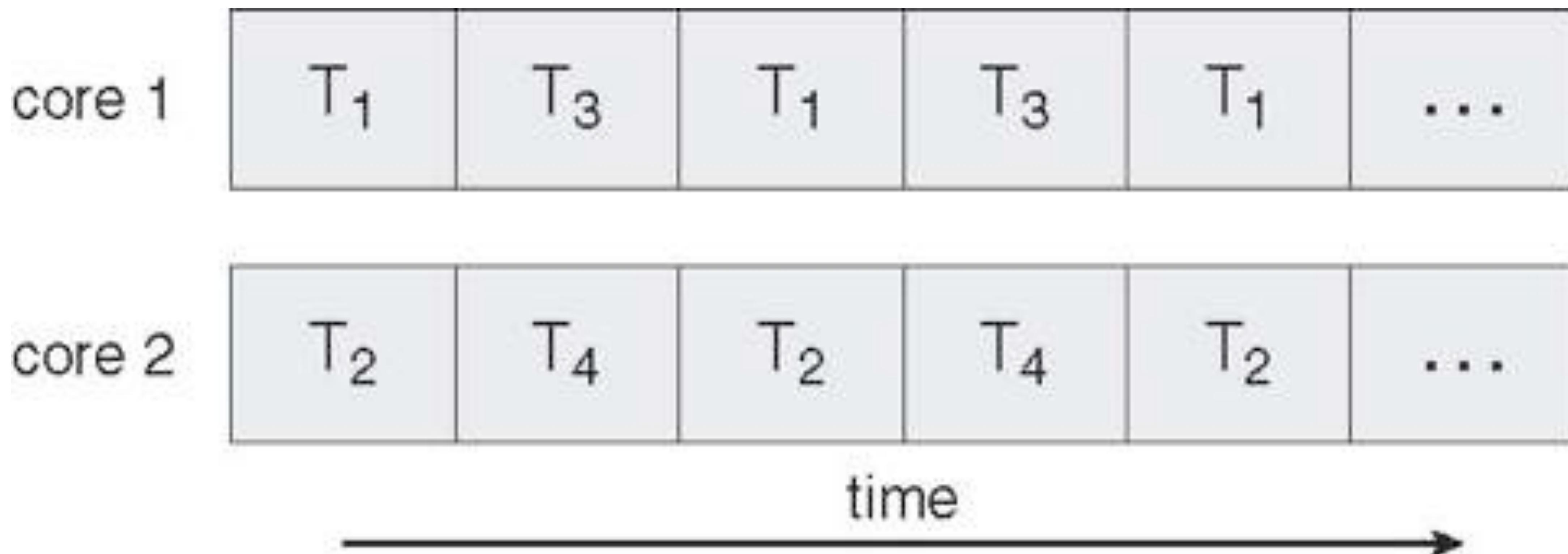
Modern OSs are mult0-threaded

- **Most OSs are now multithreaded:** several threads operate in the kernel managing devices and handling interrupts. For example, Linux uses a kernel thread for managing the amount of free memory in the system.

Concurrent Execution on a Single-core System



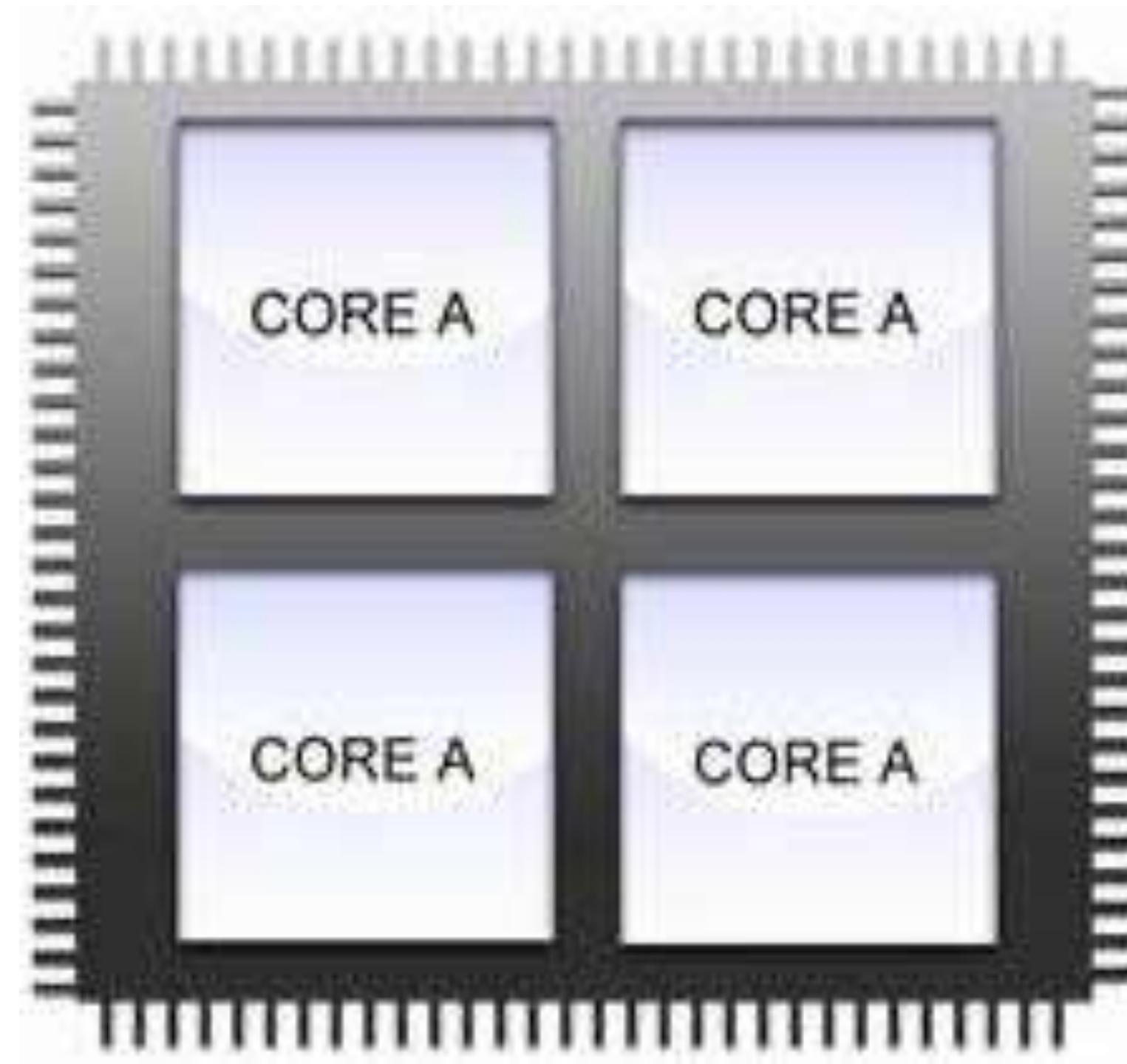
Parallel Execution on a Multicore System



Multi-core Programming

Challenges:

- ▶ Dividing activities
- ▶ Balance
- ▶ Data splitting
- ▶ Data dependency
- ▶ Testing and debugging



The challenges of developing software for multi-core systems may require an entirely new approach to designing software systems.

User Threads

- ▶ Thread management done by user-level threads library
- ▶ Three primary thread libraries:
 - ▶ POSIX Pthreads
 - ▶ Win32 threads
 - ▶ Java threads

Kernel Threads

- ▶ Supported by the Kernel
- ▶ Examples:
 - ▶ Windows XP / 2000
 - ▶ Solaris
 - ▶ Linux
 - ▶ Tru64 UNIX
 - ▶ Mac OS X

Multithreading Models

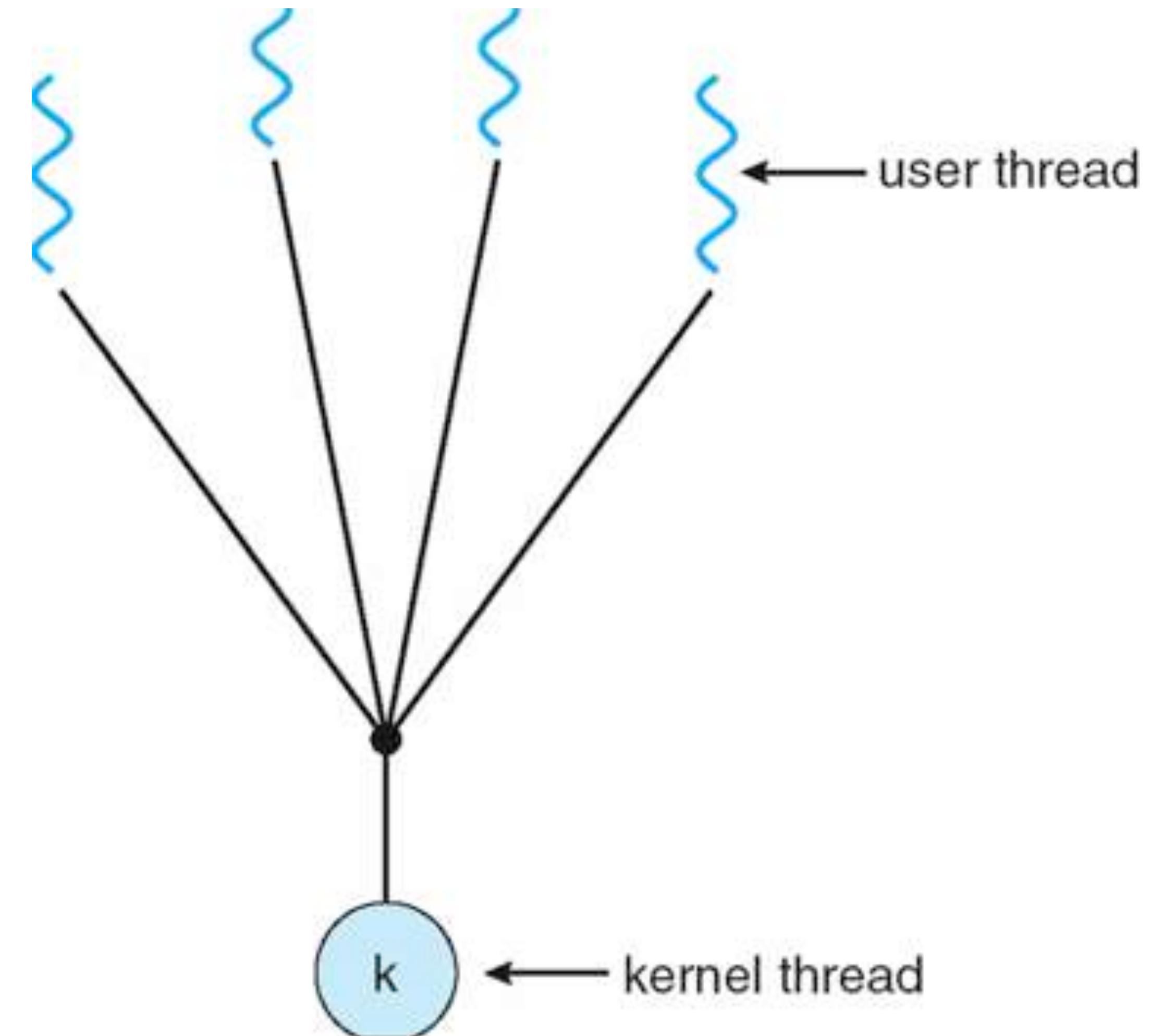
- ▶ Many-to-one
- ▶ One-to-one
- ▶ Many-to-many

Many-to-one model

Many user-level threads mapped to single kernel thread

Examples:

- ▶ Solaris Green Threads
- ▶ GNU Portable Threads

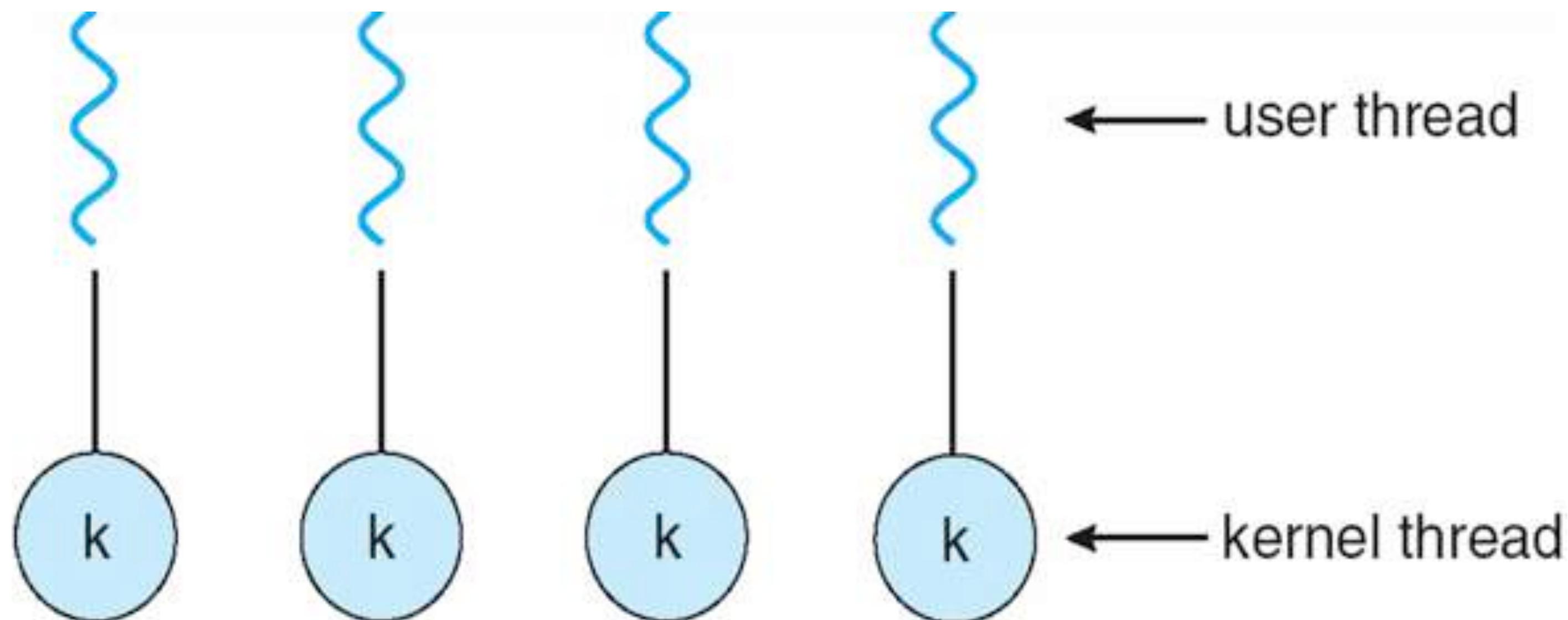


One-to-one model

Each user-level thread maps to kernel thread

Examples:

- ▶ Windows NT/XP/2000
- ▶ Linux
- ▶ Solaris 9 and later

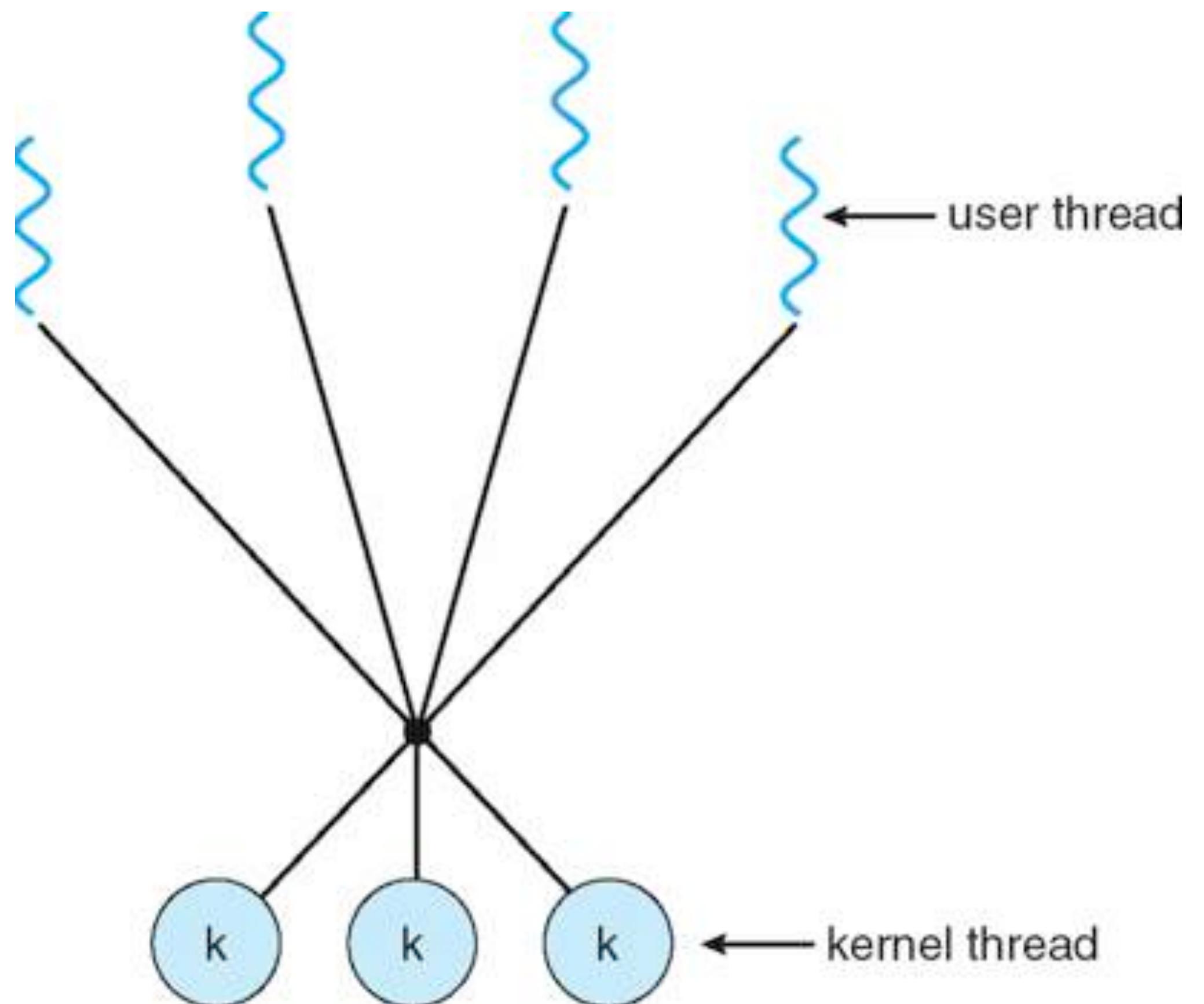


Many-to-many model

- ▶ Allows many user level threads to be mapped to many kernel threads
- ▶ Allows the operating system to create a sufficient number of kernel threads

Examples:

- ▶ Solaris prior to version 9
- ▶ Windows NT/2000 with the *ThreadFiber* package



Thread Libraries: pthreads

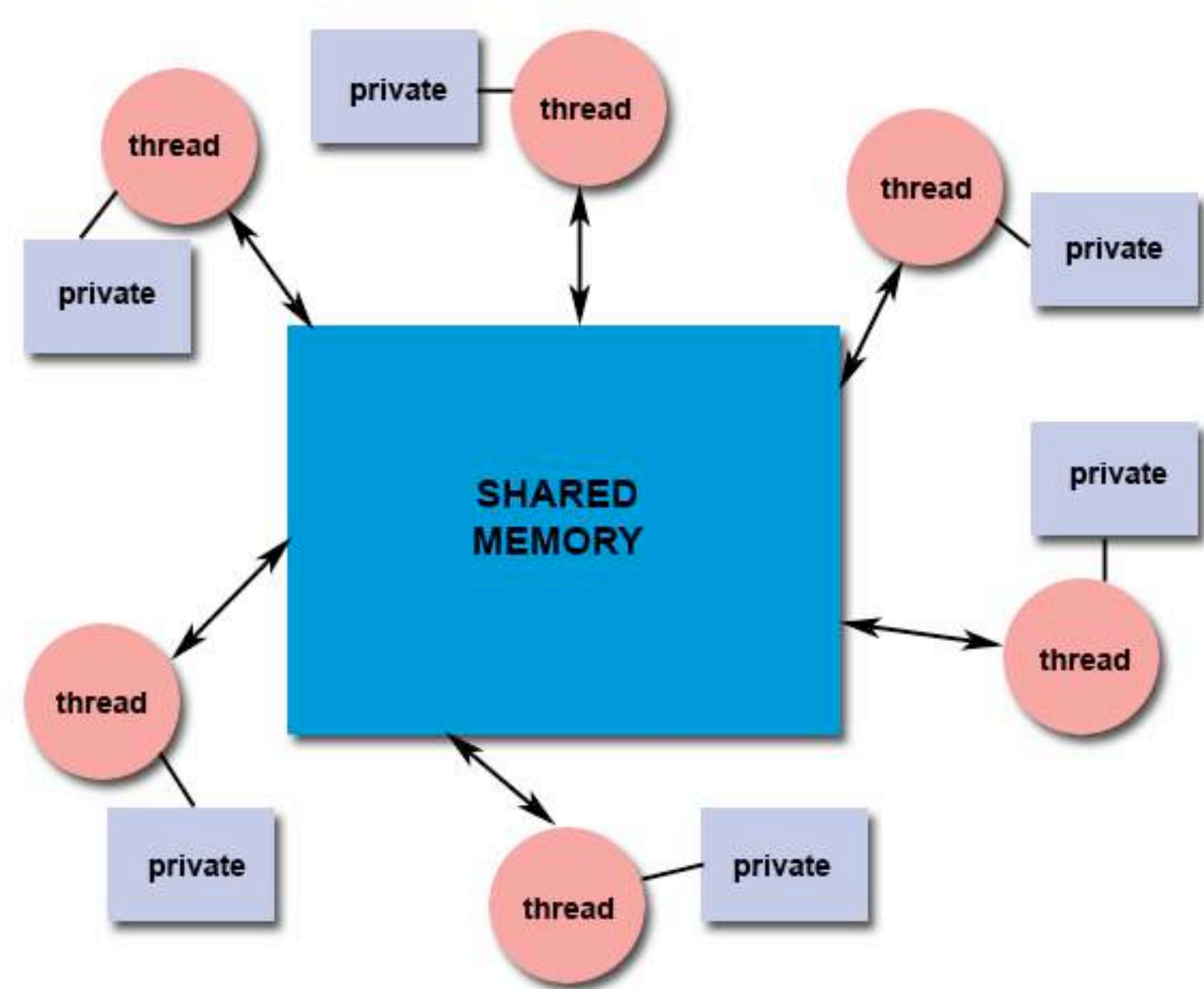
Thread library provides programmer with API for creating and managing threads

- ▶ A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- ▶ API specifies behavior of the thread library, implementation is up to the developer
- ▶ Common in UNIX operating systems (Solaris, Linux, Mac OS X)

Thread Programming

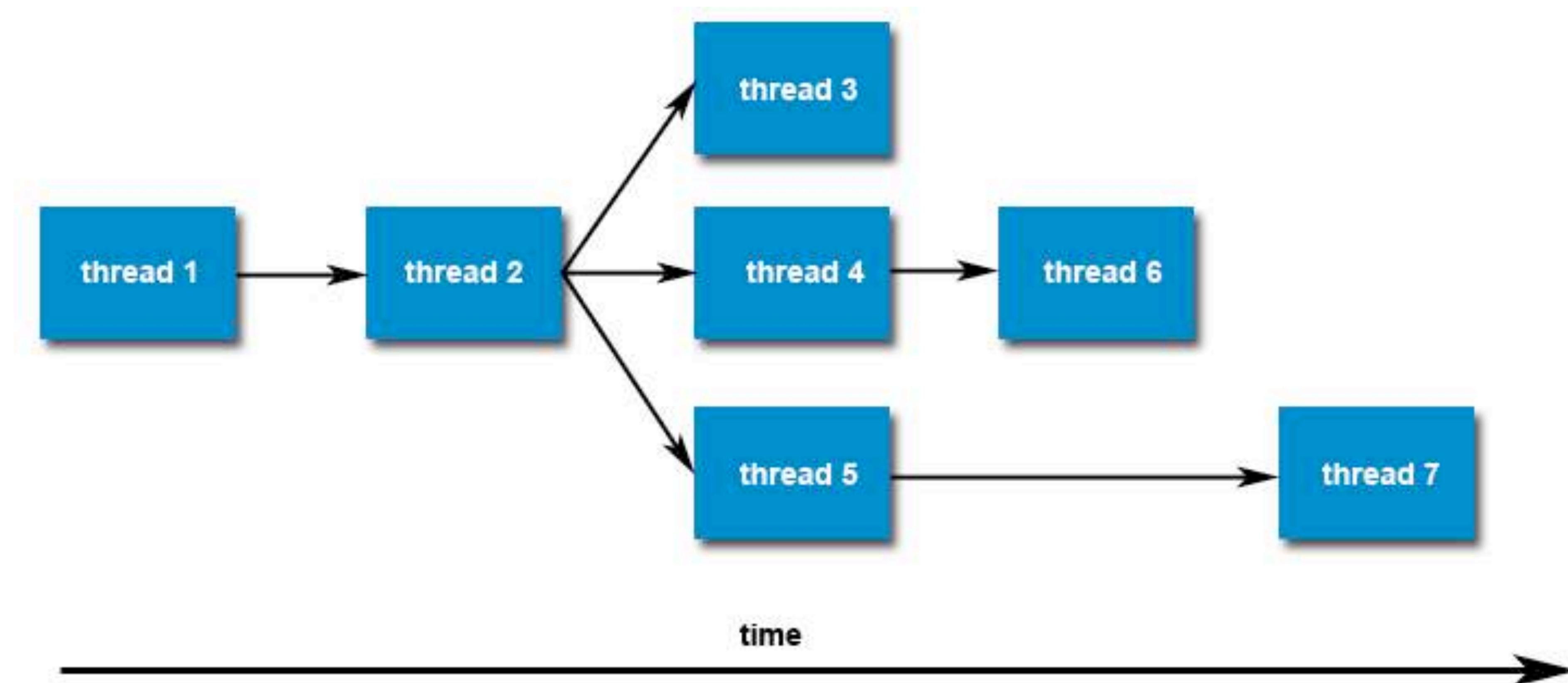
Shared Memory Model:

- ▶ All threads have access to the same global, shared memory
- ▶ Threads also have their own private data
- ▶ Programmers are responsible for synchronizing access (protecting) globally shared data.



Creating and Terminating Threads

- ▶ The maximum number of threads that may be created by a process is implementation dependent.
- ▶ Once created, threads are peers, and may create other threads. There is no implied hierarchy or dependency between threads.



Creating and Terminating Threads

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5

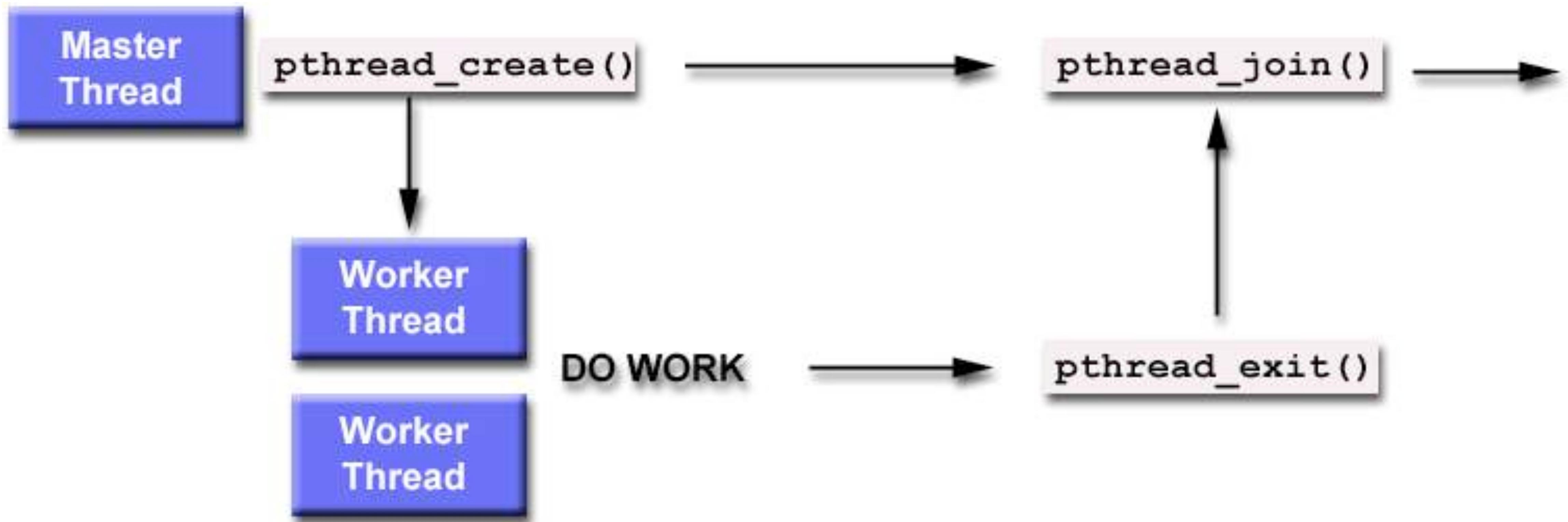
void *PrintHello( void *threadid ){
    long tid;
    tid = (long)threadid;
    printf( "Hello World! It's me, thread #%ld!\n", tid );
    pthread_exit( NULL );
}
```

Creating and Terminating Threads

```
int main ( int argc , char *argv [] ) {
    pthread_t threads [ NUM_THREADS ];
    int rc;
    long t;
    for ( t = 0; t < NUM_THREADS ; t ++ ){
        printf ( "In main: creating thread %ld\n" , t );
        rc = pthread_create ( &threads [ t ] , NULL , PrintHello , ( void * ) t );
        if ( rc ){
            printf ( "ERROR; return code from pthread_create () : %d\n" , rc );
            exit ( -1 );
        }
    }
    pthread_exit ( NULL );
}
```

Thread Management - Joining and Detaching Threads

“Joining” is one way to accomplish synchronization between threads.



Example – Joining and Detaching Threads

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define NUM_THREADS 4

void *BusyWork( void *t )
{
    int i;
    long tid;
    double result=0.0;
    tid = (long)t;
    printf("Thread %ld starting...\n",tid);
    for (i=0; i<1000000; i++)
        result = result + sin(i) * tan(i);

    printf("Thread %ld done. Result = %e\n",tid, result);
    pthread_exit((void*) t );
}
```

Worker Function

Main Function

```

int main ( int argc , char *argv[] ){
    pthread_t thread[ NUM_THREADS ];
    pthread_attr_t attr;
    int rc; long t; void *status;

    /* Initialize and set thread detached attribute */
    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr , PTHREAD_CREATE_JOINABLE);

    for( t=0; t<NUM_THREADS; t++ ) {
        printf( "Main: creating thread %ld\n" , t );
        rc = pthread_create(&thread[t] , &attr , BusyWork , ( void * )t );
        if ( rc ) exit(-1);
    }

    /* Free attribute and wait for the other threads */
    pthread_attr_destroy(&attr);
    for( t=0; t<NUM_THREADS; t++ ) {
        rc = pthread_join(thread[t] , &status );
        if ( rc ) exit(-1);

        printf( "Main: completed join with thread %ld having a status
               of %ld\n" , t ,( long )status );
    }

    printf( "Main: program completed. Exiting.\n" );
    pthread_exit( NULL );
}

```

Boost Libraries

Example 1

```
#include <iostream>
#include <boost/thread.hpp>
#include <boost/date_time.hpp>

void workerFunc()
{
    boost::posix_time::seconds workTime(3);
    std::cout << "Worker: running" << std::endl;

    // Pretend to do something useful...
    boost::this_thread::sleep(workTime);
    std::cout << "Worker: finished" << std::endl;
}

int main(int argc, char* argv[])
{
    std::cout << "main: startup" << std::endl;
    boost::thread workerThread(workerFunc);

    std::cout << "main: waiting for thread" << std::endl;
    workerThread.join();

    std::cout << "main: done" << std::endl;
    return 0;
}
```

```
main: startup
main: waiting for thread
Worker: running
Worker: finished
main: done
```

Example 2

```
#include <boost/thread.hpp>
#include <iostream>
```

```
void wait(int seconds)
{
    boost::this_thread::sleep(boost::posix_time::seconds(seconds));
}
```

```
void thread()
{
    for (int i = 0; i < 5; ++i)
    {
        wait(1);
        std::cout << i << std::endl;
    }
}
```

```
int main()
{
    boost::thread t(thread);
    t.join();
}
```

Declares a variable `t` of type
`boost::thread`

```
#include <boost/thread.hpp>
#include <iostream>
```

```
void wait(int seconds)
{
    boost::this_thread::sleep(boost::posix_time::seconds(seconds));
}
```

```
void thread() ←
{
    for (int i = 0; i < 5; ++i)
    {
        wait(1);
        std::cout << i << std::endl;
    }
}
```

```
int main()
{
    boost::thread t(thread);
    t.join();
}
```

This is the function we want to be executed within the thread.

```
#include <boost/thread.hpp>
#include <iostream>

void wait(int seconds)
{
    boost::this_thread::sleep(boost::posix_time::seconds(seconds));
}

void thread()
{
    for (int i = 0; i < 5; ++i)
    {
        wait(1);
        std::cout << i << std::endl;
    }
}

int main()
{
    boost::thread t(thread);
    t.join();
}
```

Name of the function to be executed within the thread is passed to the constructor of `boost::thread`

```
#include <boost/thread.hpp>
#include <iostream>

void wait(int seconds)
{
    boost::this_thread::sleep(boost::posix_time::seconds(seconds));
}

void thread()
{
    for (int i = 0; i < 5; ++i)
    {
        wait(1);
        std::cout << i << std::endl;
    }
}

int main()
{
    boost::thread t(thread);
    t.join();
}
```

Upon creation, the `thread` function starts executing in its own thread *immediately*. Function `main()` is also executing in its own thread. Here, we say that these functions are executing *concurrently*.

```
#include <boost/thread.hpp>
#include <iostream>
```

```
void wait(int seconds)
{
    boost::this_thread::sleep(boost::posix_time::seconds(seconds));
}
```

```
void thread()
{
    for (int i = 0; i < 5; ++i)
    {
        wait(1);
        std::cout << i << std::endl;
    }
}
```

```
int main()
{
    boost::thread t(thread);
    t.join();
}
```

The method `join()` blocks the calling thread until thread `t` terminates. Basically, it forces `main()` to wait for `t`.

```
#include <boost/thread.hpp>
#include <iostream>

void wait(int seconds)
{
    boost::this_thread::sleep(boost::posix_time::seconds(seconds));
}

void thread()
{
    for (int i = 0; i < 5; ++i)
    {
        wait(1);
        std::cout << i << std::endl;
    }
}

int main()
{
    boost::thread t(thread);
    t.join(); ←
}
```

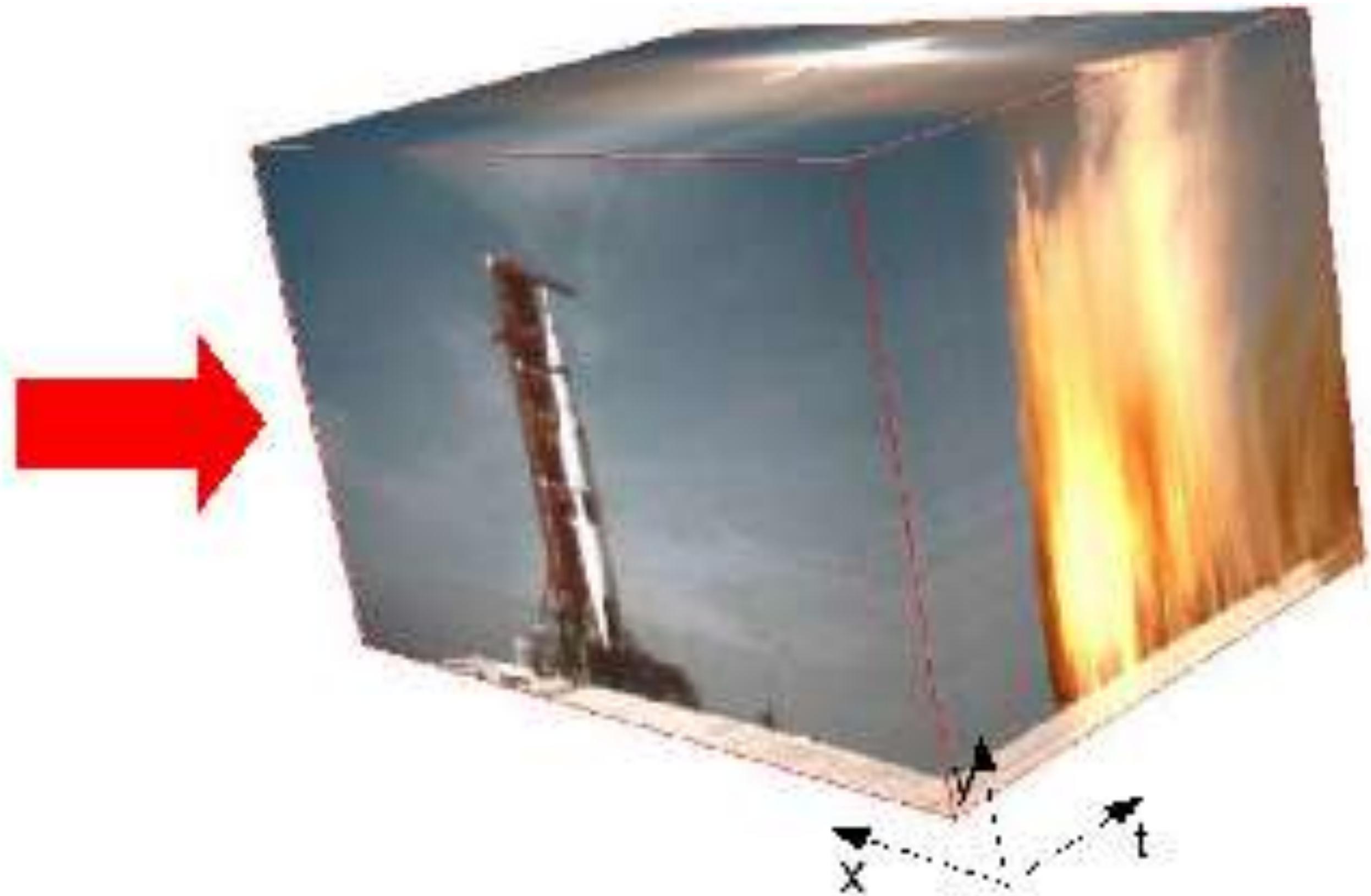
What happens if we don't call join()?

Example: Video processing

- » Video as a 3-D array (volume)



Source: NASA

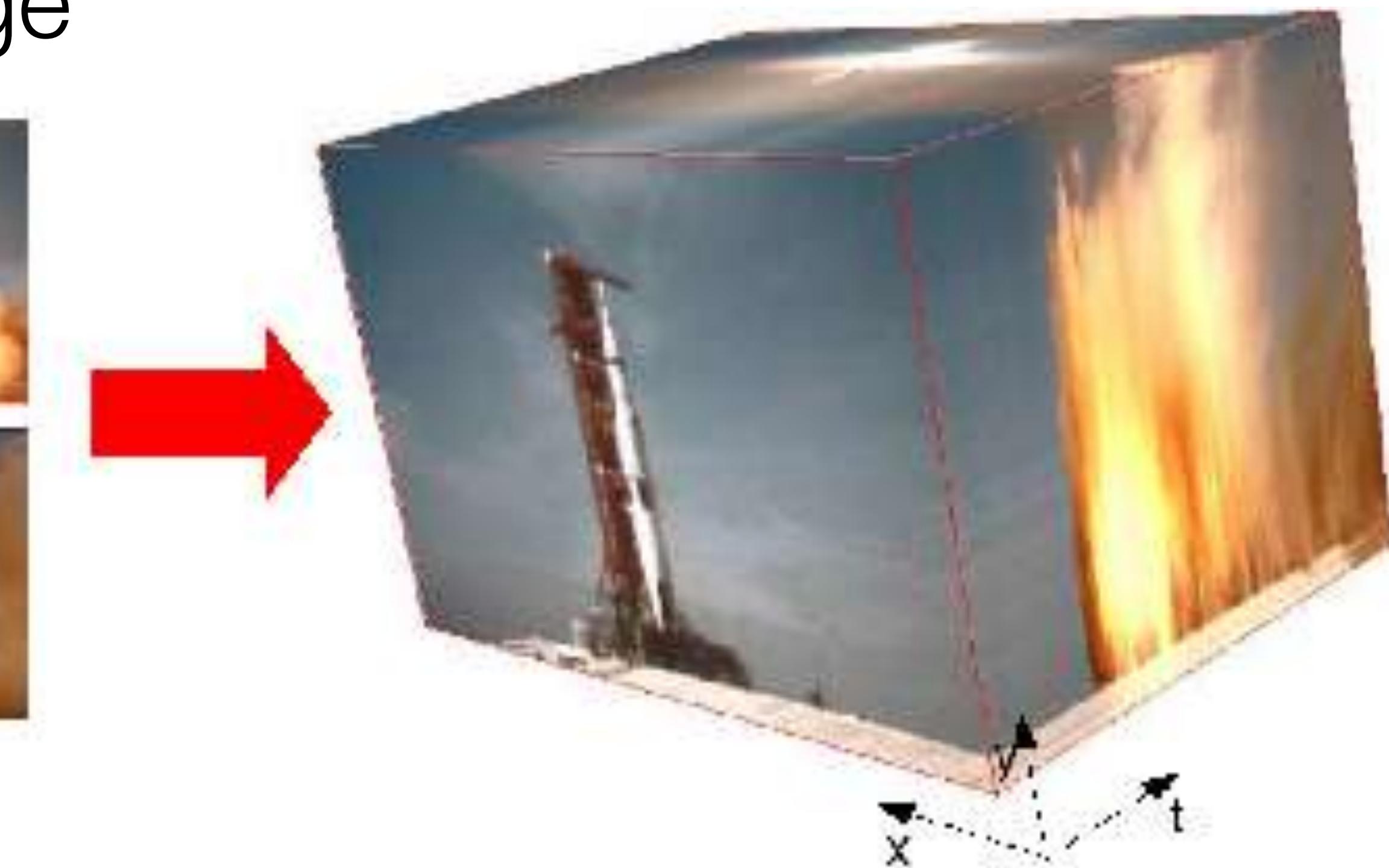


Example: Video processing

- » Some tasks of a video-processing software can be done concurrently by separate threads.
 - Calculate the average image
 - Calculate the median image

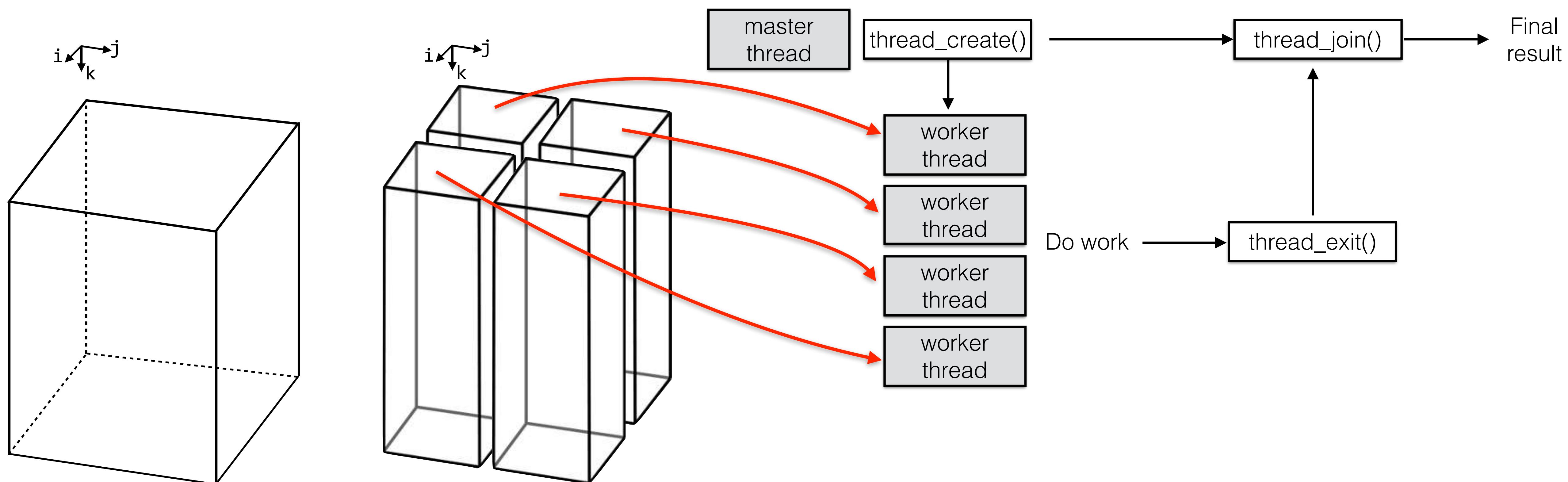


Source: NASA



Example: Video processing

- » We can also speed up each task by first dividing the video into sub volumes, and then assign each sub volume to be processed by a separate thread.



An aerial photograph of a city street. In the foreground, several cars are parked along the curb. A yellow double-decker bus is driving away from the camera on the left side of the street. In the background, there are buildings, trees, and a bridge. The image is slightly blurred.

Synchronization

Synchronizing threads

- » Multi-threaded programming can increase performance of applications. But, complexity is also increased.
- » Access to shared resources must be controlled by trying to synchronize access.

```
void wait(int seconds)
{
    boost::this_thread::sleep(boost::posix_time::seconds(seconds));
}

boost::mutex mutex;

void thread()
{
    for (int i = 0; i < 5; ++i)
    {
        wait(1);
        mutex.lock();
        std::cout << "Thread " << boost::this_thread::get_id() << ":" << i << std::endl;
        mutex.unlock();
    }
}

int main()
{
    boost::thread t1(thread);
    boost::thread t2(thread);
    t1.join();
    t2.join();
}
```

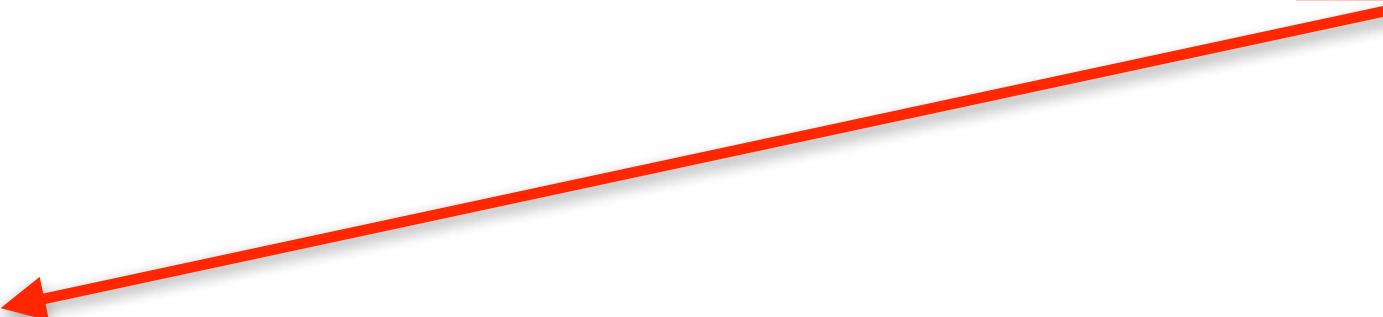
```
void wait(int seconds)
{
    boost::this_thread::sleep(boost::posix_time::seconds(seconds));
}

boost::mutex mutex;

void thread()
{
    for (int i = 0; i < 5; ++i)
    {
        wait(1);
        mutex.lock();
        std::cout << "Thread " << boost::this_thread::get_id() << ":" << i << std::endl;
        mutex.unlock();
    }
}

int main()
{
    boost::thread t1(thread);
    boost::thread t2(thread);
    t1.join();
    t2.join();
}
```

Creates two threads, both execution the `thread()` function.



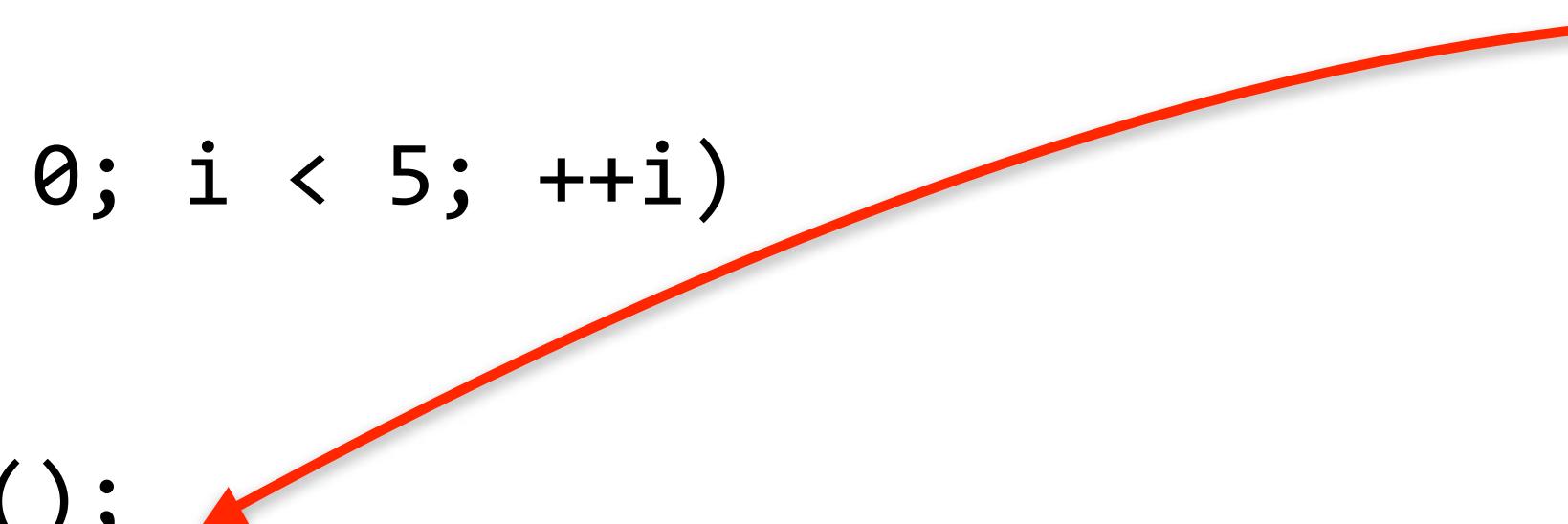
```
void wait(int seconds)
{
    boost::this_thread::sleep(boost::posix_time::seconds(seconds));
}

boost::mutex mutex;

void thread()
{
    for (int i = 0; i < 5; ++i)
    {
        wait(1);
        mutex.lock();
        std::cout << "Thread " << boost::this_thread::get_id() << ":" << i << std::endl;
        mutex.unlock();
    }
}

int main()
{
    boost::thread t1(thread);
    boost::thread t2(thread);
    t1.join();
    t2.join();
}
```

The `thread()` function writes on the standard output stream (on the console). This stream is a *global object* shared by all threads.



```
void wait(int seconds)
{
    boost::this_thread::sleep(boost::posix_time::seconds(seconds));
}
```

```
boost::mutex mutex;
void thread()
{
    for (int i = 0; i < 5; ++i)
    {
        wait(1);
        mutex.lock();
        std::cout << "Thread " << boost::this_thread::get_id() << ":" << i << std::endl;
        mutex.unlock();
    }
}

int main()
{
    boost::thread t1(thread);
    boost::thread t2(thread);
    t1.join();
    t2.join();
}
```

We need to synchronize access to this *shared* resource otherwise messages from multiple threads will overlap on the console.



```
void wait(int seconds)
{
    boost::this_thread::sleep(boost::posix_time::seconds(seconds));
}
```

```
boost::mutex mutex;
void thread()
{
    for (int i = 0; i < 5; ++i)
    {
        wait(1);
        mutex.lock();
        std::cout << "Thread " << boost::this_thread::get_id() << ":" << i << std::endl;
        mutex.unlock();
    }
}

int main()
{
    boost::thread t1(thread);
    boost::thread t2(thread);
    t1.join();
    t2.join();
}
```

Here, we declare a global **mutex** (i.e., mutual-exclusion object)

```

void wait(int seconds)
{
    boost::this_thread::sleep(boost::posix_time::seconds(seconds));
}

boost::mutex mutex;

void thread()
{
    for (int i = 0; i < 5; ++i)
    {
        wait(1);
        mutex.lock(); ←
            std::cout << "Thread " << boost::this_thread::get_id() << ":" << i << std::endl;
        mutex.unlock();
    }
}

int main()
{
    boost::thread t1(thread);
    boost::thread t2(thread);
    t1.join();
    t2.join();
}

```

A **mutex** works like a “traffic semaphore” or lock. Multiple threads will see it but only one thread can get hold of it. Once one thread locks the mutex, all other threads that “try it” will need to wait until the lock is released by the thread that was holding it.



Install the boost library (Ubuntu)

```
sudo apt-get install libboost-all-dev
```

To learn more about boost threads

» Tutorial:

<http://theboostcpplibraries.com/boost.thread>

OpenMP (Open Multi-Processing)



- » OpenMP is a set of compiler directives as well as an API for programs written in C, C++, or Fortran that provides support for parallel programming in shared-memory environments.

OpenMP (Open Multi-Processing)



- » OpenMP identifies **parallel regions** as blocks of code that may run in parallel. Application developers insert compiler directives into their code at parallel regions, and these directives instruct the OpenMP run-time library to execute the region in parallel.

OpenMP (Open Multi-Processing)

This program will print a message which will be getting executed by various threads.

```
// OpenMP header
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char* argv[])
{
    int nthreads, tid;

    // Begin of parallel region
    #pragma omp parallel private(nthreads, tid)
    {
        // Getting thread number
        tid = omp_get_thread_num();
        printf("Welcome to GFG from thread = %d\n", tid);

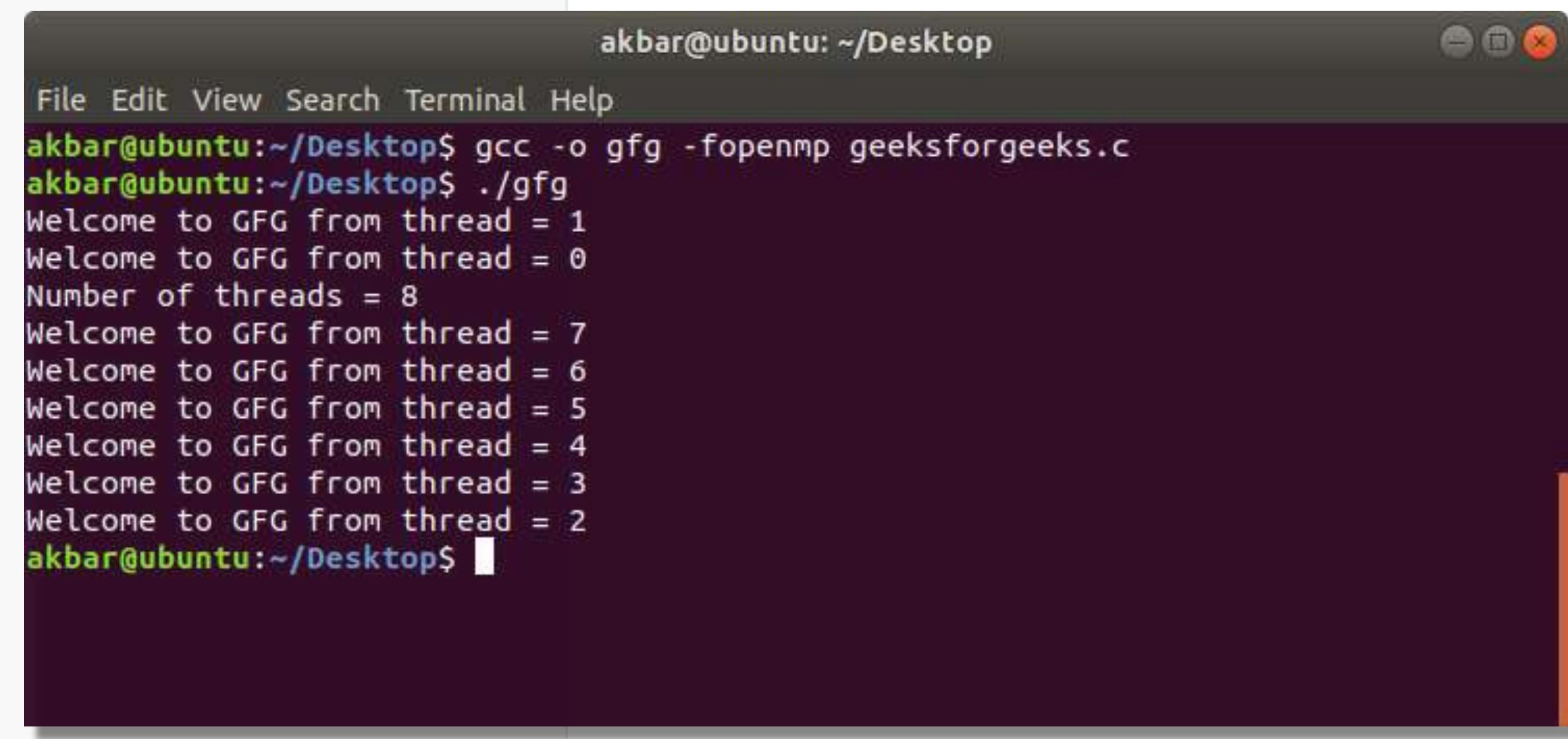
        if (tid == 0) {
            // only master thread does this
            nthreads = omp_get_num_threads();
            printf("Number of threads = %d\n", nthreads);
        }
    }
}
```

Compile:

```
gcc -o gfg -fopenmp geeksforgeeks.c
```

Execute:

```
./gfg
```



The terminal window shows the following session:

```
akbar@ubuntu:~/Desktop$ gcc -o gfg -fopenmp geeksforgeeks.c
akbar@ubuntu:~/Desktop$ ./gfg
Welcome to GFG from thread = 1
Welcome to GFG from thread = 0
Number of threads = 8
Welcome to GFG from thread = 7
Welcome to GFG from thread = 6
Welcome to GFG from thread = 5
Welcome to GFG from thread = 4
Welcome to GFG from thread = 3
Welcome to GFG from thread = 2
akbar@ubuntu:~/Desktop$
```

* To find out how many CPUs, type **lscpu** on the command line.



Happy Multi-thread Programming!!



Thread Synchronization

CSE4001 Operating Systems Concepts

E. Ribeiro

Thread Synchronization (Part 1)

CSE 4001

The Little Book of Semaphores

Allen B. Downey

Version 2.2.1

Book URL: <https://greenteapress.com/wp/semaphores/>

Non-deterministic execution order

- Concurrent programs are often non-deterministic as order of execution depends on the scheduler.

Single program with two threads

Thread A

```
1 print "yes"
```

Thread B

```
1 print "no"
```

Concurrent writes on shared variables

- The value that gets printed depends on the order in which the statements are executed (i.e., the execution path).

Single program with two threads

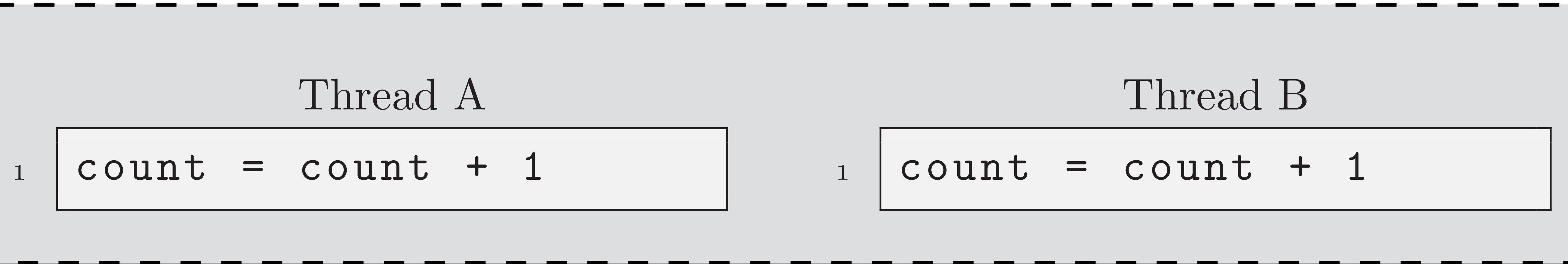
Thread A

```
1 x = 5
2 print x
```

Thread B

```
1 x = 7
```

Concurrent updates on shared variables



Translation to machine language



Semaphores

A semaphore is like an integer, with three differences:

1. When you create the semaphore, you can initialize its value to any integer, but after that the only operations you are allowed to perform are increment (increase by one) and decrement (decrease by one). You cannot read the current value of the semaphore.
2. When a thread decrements the semaphore, if the result is negative, the thread blocks itself and cannot continue until another thread increments the semaphore.
3. When a thread increments the semaphore, if there are other threads waiting, one of the waiting threads gets unblocked.

Semaphore implementation

If semaphore is closed, block the thread that called `wait()` on a queue associated with the semaphore. Otherwise, let the thread that called `wait()` continue into the critical section.

The `wait()` function:

```
wait() {  
    value = value - 1  
    if (value < 0) {  
        add this thread to list  
        block thread  
    }  
}
```

Semaphore implementation

Wake up one of the threads that called wait(s), and run it so that it can continue into the critical section.

The signal() function:

```
signal() {  
    value = value + 1  
    if (value <= 0) {  
        remove a thread from list  
        wakeup thread  
    }  
}
```

Semaphores: syntax

Semaphore initialization syntax

```
1 fred = Semaphore(1)
```

Semaphore operations

```
1 fred.signal()  
2 fred.wait()
```

Synchronization Constraints

Serialization: Event A must happen before Event B.

Mutual exclusion: Events A and B must not happen at the same time.

- We will use a combination of these two constraints to solve most thread-synchronization problems

Basic synchronization patterns: Signaling

- **a1** must happen before **b1**

Thread A

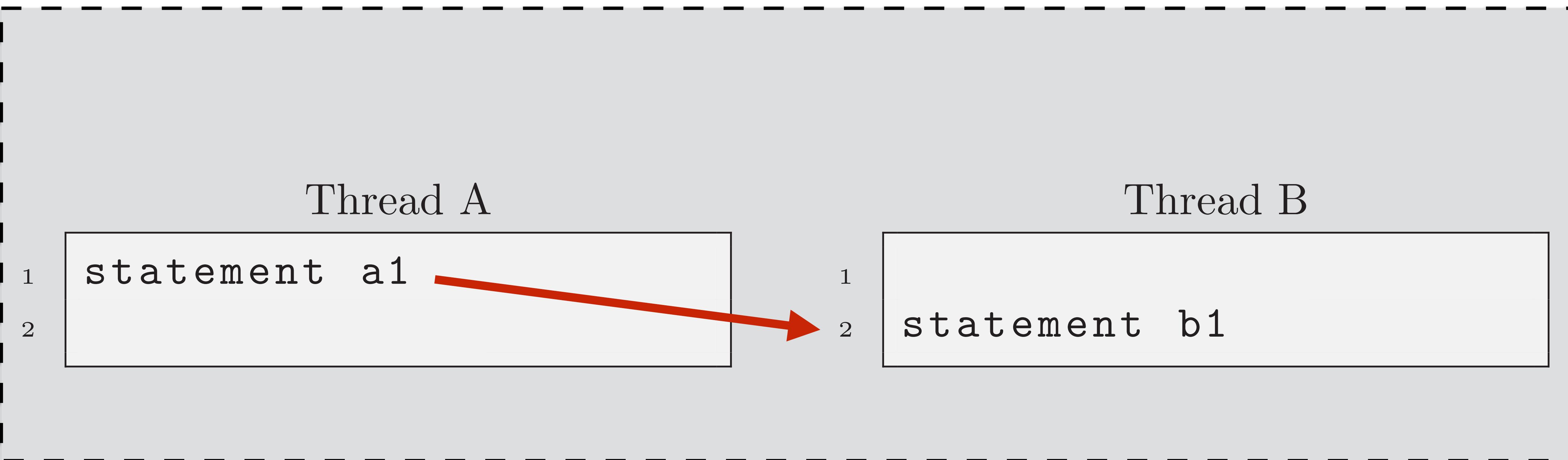
```
1 statement a1  
2
```

Thread B

```
1  
2 statement b1
```

Basic synchronization patterns: Signaling

- **a1** must happen before **b1**



Basic synchronization patterns: Signaling

- **a1** must happen before **b1**

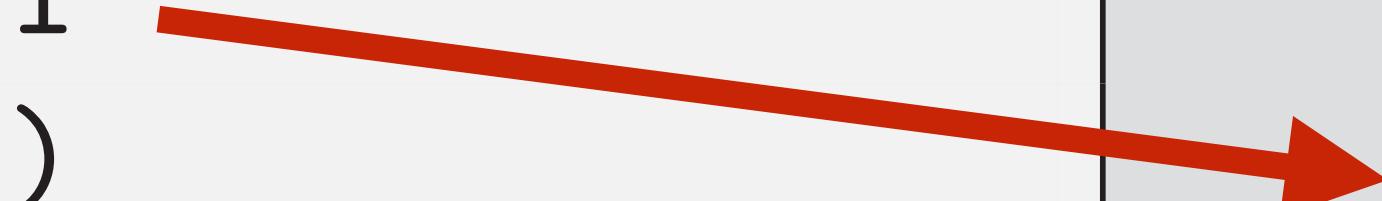
```
sem = semaphore(0)
```

Thread A

```
1 statement a1
2 sem.signal()
```

Thread B

```
1 sem.wait()
2 statement b1
```



Basic synchronization patterns: Signaling

- **a1** must happen before **b1**

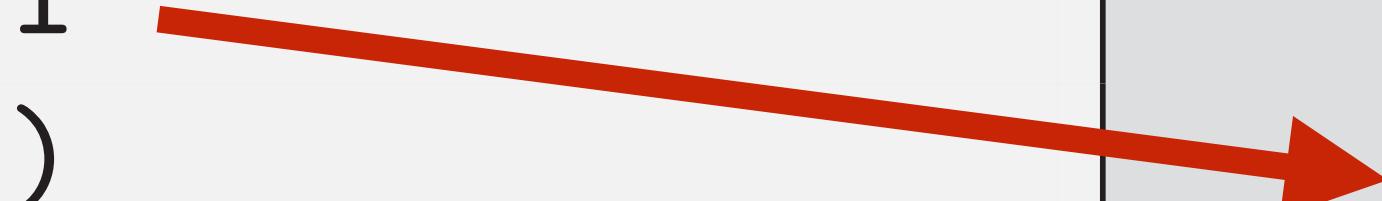
```
sem = semaphore(0)
```

Thread A

```
1 statement a1
2 sem.signal()
```

Thread B

```
1 sem.wait()
2 statement b1
```



Basic synchronization patterns: Signaling

- **a1** must happen before **b1**
- Same solution using better semaphore naming

```
a1Done = semaphore(0)
```

Thread A

```
1 statement a1  
2 a1Done.signal()
```

Thread B

```
1 a1Done.wait()  
2 statement b1
```

Basic synchronization patterns: Rendezvous

- **a1** must happen before **b2**
- **b1** must happen before **a2**

Thread A

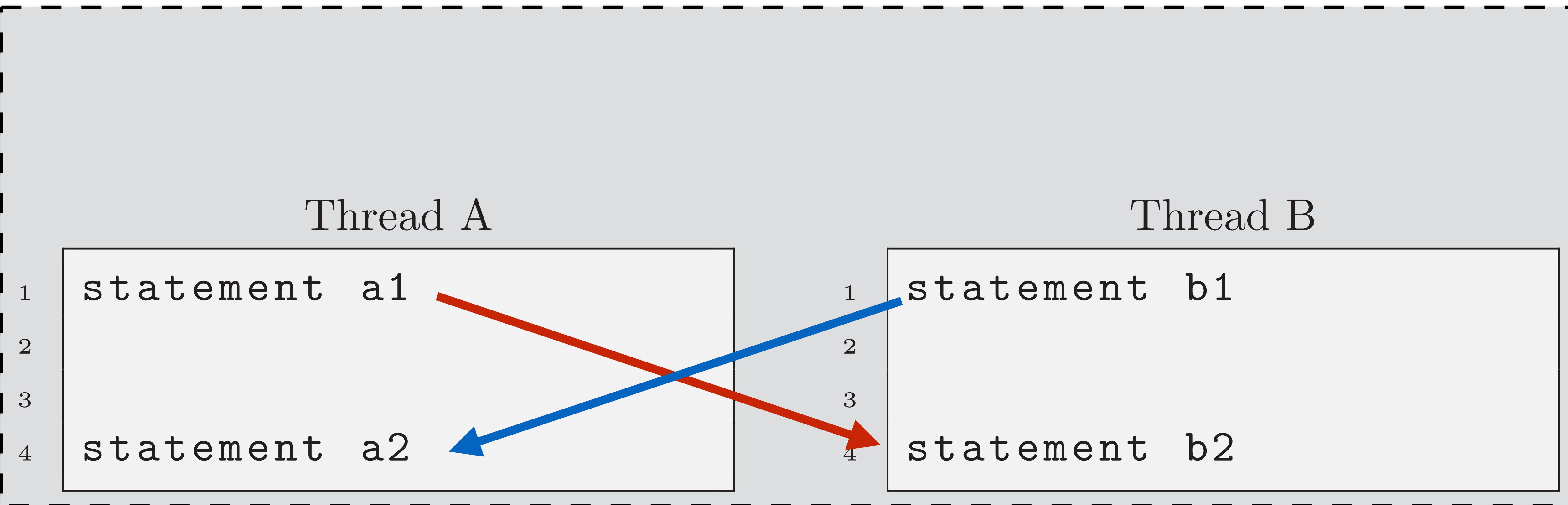
```
1 statement a1  
2 statement a2
```

Thread B

```
1 statement b1  
2 statement b2
```

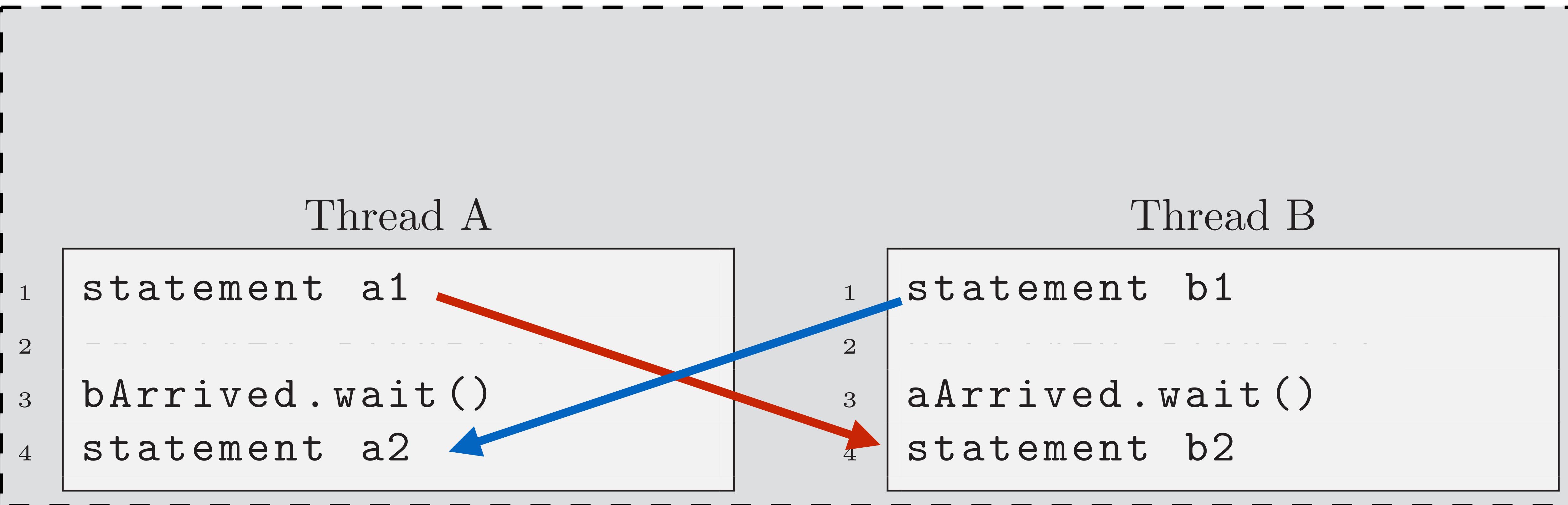
Basic synchronization patterns: Rendezvous

- **a1** must happen before **b2**
- **b1** must happen before **a2**



Basic synchronization patterns: Rendezvous

- **a1** must happen before **b2**
- **b1** must happen before **a2**



Basic synchronization patterns: Rendezvous

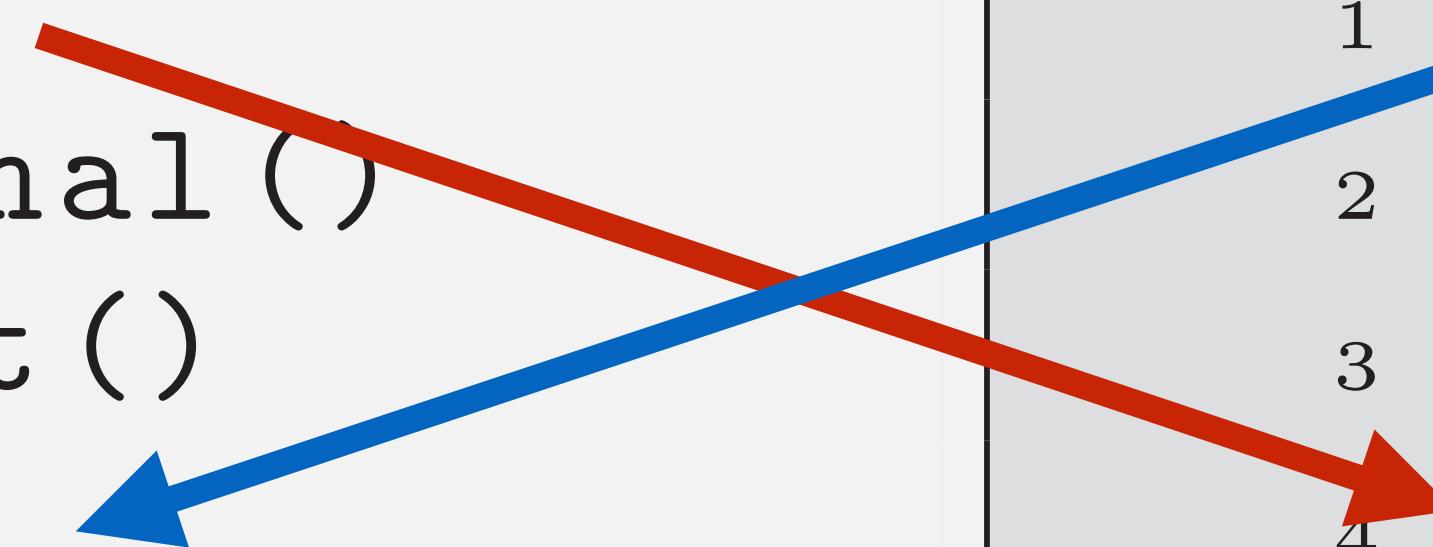
- **a1** must happen before **b2**
- **b1** must happen before **a2**

Thread A

```
1 statement a1
2 aArrived.signal()
3 bArrived.wait()
4 statement a2
```

Thread B

```
1 statement b1
2 bArrived.signal()
3 aArrived.wait()
4 statement b2
```



Basic synchronization patterns: Rendezvous

- **a1** must happen before **b2**
- **b1** must happen before **a2**

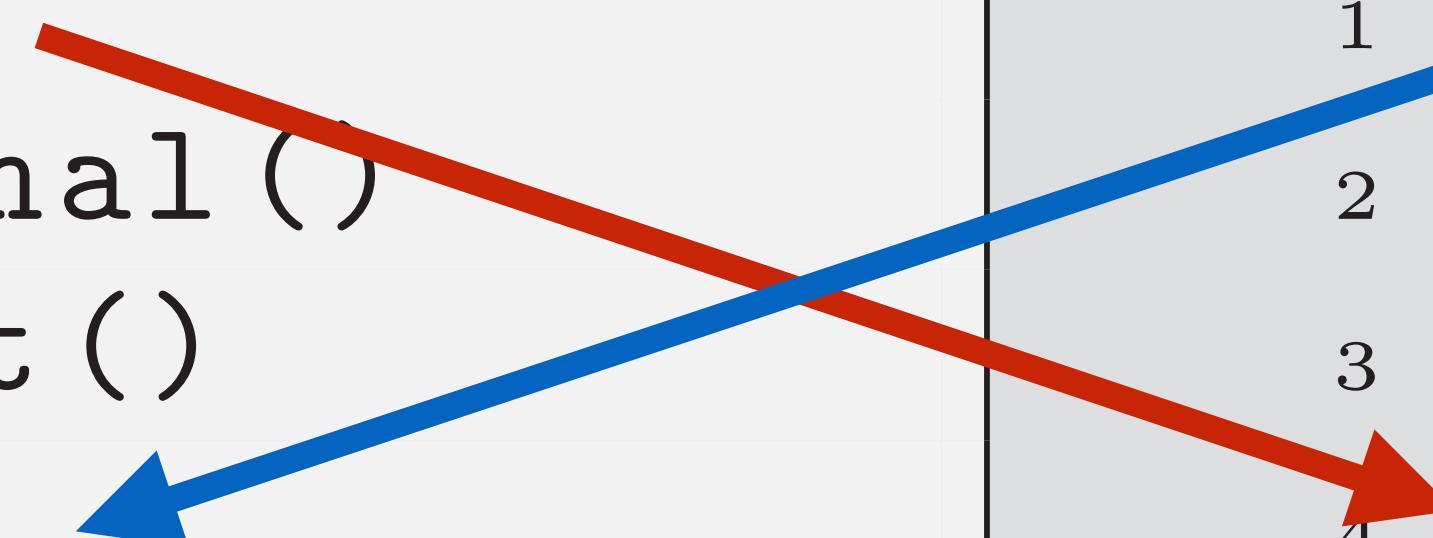
```
aArrived = semaphore(0)  
bArrived = semaphore(0)
```

Thread A

```
1 statement a1  
2 aArrived.signal()  
3 bArrived.wait()  
4 statement a2
```

Thread B

```
1 statement b1  
2 bArrived.signal()  
3 aArrived.wait()  
4 statement b2
```



Example

```
#include "semaphore_class.h"

/* prototype for thread routine */
void *threadB ( void *ptr );
void *threadA ( void *ptr );

/* global vars */
Semaphore B_Done(0);

int main()
{
    int i[3];
    pthread_t thread_a;
    pthread_t thread_b;
    i[0] = 0; i[1] = 1; /* argument to threads */

    pthread_create (&thread_a, NULL, threadA, (void *) &i[0]);
    pthread_create (&thread_b, NULL, threadB, (void *) &i[1]);

    exit(0);
} /* main() */
```

```
void *threadA ( void *ptr )
{
    int x;
    x = *((int *) ptr);

    B_Done.wait();

    printf("Thread %d: Statement A: Must run after Statement B. \n", x);
    fflush(stdout);

    B_Done.signal();

    pthread_exit(0); /* exit thread */
}
```

```
void *threadB ( void *ptr )
{
    int x;
    x = *((int *) ptr);

    printf("Thread %d: Statement B: Must run before Statement A. \n", x);
    fflush(stdout);

    B_Done.signal();

    pthread_exit(0); /* exit thread */
}
```

Example

```
#include "semaphore_class.h"

/* prototype for thread routine */
void *threadB ( void *ptr );
void *threadA ( void *ptr );

/* global vars */
Semaphore B_Done(0);

int main()
{
    int i[3];
    pthread_t thread_a;
    pthread_t thread_b;
    i[0] = 0; i[1] = 1; /* argument to threads */

    B_Done.wait();
    printf("Thread %d: statem
ation A. \n", x);
    fflush(stdout);

    B_Done.signal();

    exit(0);

    pthread_exit(0); /* exit */
} /* main() */
```

Example

```
#include "semaphore_class.h"

/*
void *threadA ( void *ptr )
/*
Semaphore *B_Done;
{
    int x;
    x = *((int *) ptr);

    B_Done.wait();

    printf("Thread %d: Statement A: Must run after Statement B. \n", x);
    fflush(stdout);

    B_Done.signal();

    pthread_exit(0); /* exit thread */
}
```

}

Example

```
#include "semaphore_class.h"

/* prototype for thread routine */
void *threadB ( void *ptr );
void *threadA ( void *ptr );

/* global vars */
Semaphore *S;
int max_val = 10;
int current_val = 0;
int i;
int j;
int k;
int l;
int m;
int n;
int o;
int p;
int q;
int r;
int s;
int t;
int u;
int v;
int w;
int x;
int y;
int z;

void *threadB ( void *ptr )
{
    int x;
    x = *((int *) ptr);

    printf("Thread %d: Statement B: Must run before Statement A. \n", x);
    fflush(stdout);

    B_Done.signal();

    pthread_exit(0); /* exit thread */
}

void *threadA ( void *ptr )
{
    int x;
    x = *((int *) ptr);

    B_Done.wait();

    printf("Thread %d: Statement A: Must run after Statement B. \n", x);
    fflush(stdout);

    S->signal();
}
```

Thread Synchronization (Part 2)

CSE 4001

Contents

- Semaphore implementation
- The mutual exclusion constraint
- The producer-consumer problem

Semaphore implementation

If semaphore is closed, block the thread that called `wait()` on a queue associated with the semaphore. Otherwise, let the thread that called `wait()` continue into the critical section.

The `wait()` function:

```
wait() {  
    value = value - 1  
    if (value < 0) {  
        add this thread to list  
        block thread  
    }  
}
```

Semaphore implementation

Wake up one of the threads that called wait(s), and run it so that it can continue into the critical section.

The signal() function:

```
signal() {  
    value = value + 1  
    if (value <= 0) {  
        remove a thread from list  
        wakeup thread  
    }  
}
```

Mutual exclusion

- A second common use of semaphores: to enforce mutual exclusion and controlling concurrent access to shared variables.
- The mutex guarantees that only one thread accesses the shared variable at a time.

Thread A

```
count = count + 1
```

Thread B

```
count = count + 1
```

Mutual exclusion hint

- Create a semaphore named `mutex` that is initialized to 1.
- A value of one means that a thread may proceed and access the shared variable;
- A value of zero means that it has to wait for another thread to release the `mutex`.

Mutual exclusion solution

Thread A

```
mutex.wait()  
    # critical section  
    count = count + 1  
mutex.signal()
```

Thread B

```
mutex.wait()  
    # critical section  
    count = count + 1  
mutex.signal()
```

Multiplex

- It allows multiple threads to run in the critical section at the same time, but it enforces an upper limit on the number of concurrent threads.
- In other words, no more than n threads can run in the critical section at the same time.

hint: treat semaphores as a set of tokens. If no tokens are available when a thread arrives at the critical section, it waits until another thread releases one.

The producer-consumer problem

Producer

```
event = waitForEvent()  
buffer.add(event)
```

Consumer

```
event = buffer.get()  
event.process()
```

Access to the buffer has to be exclusive, but `waitForEvent()` and `event.process()` can run concurrently.

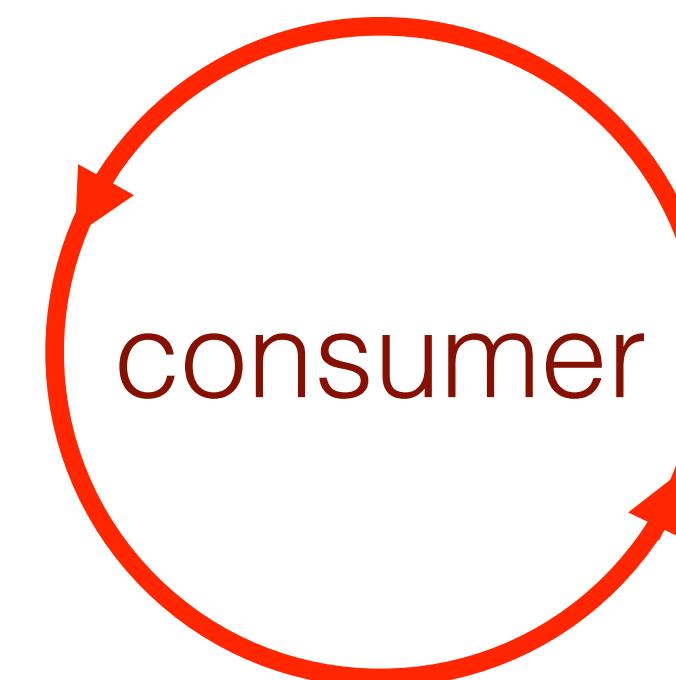
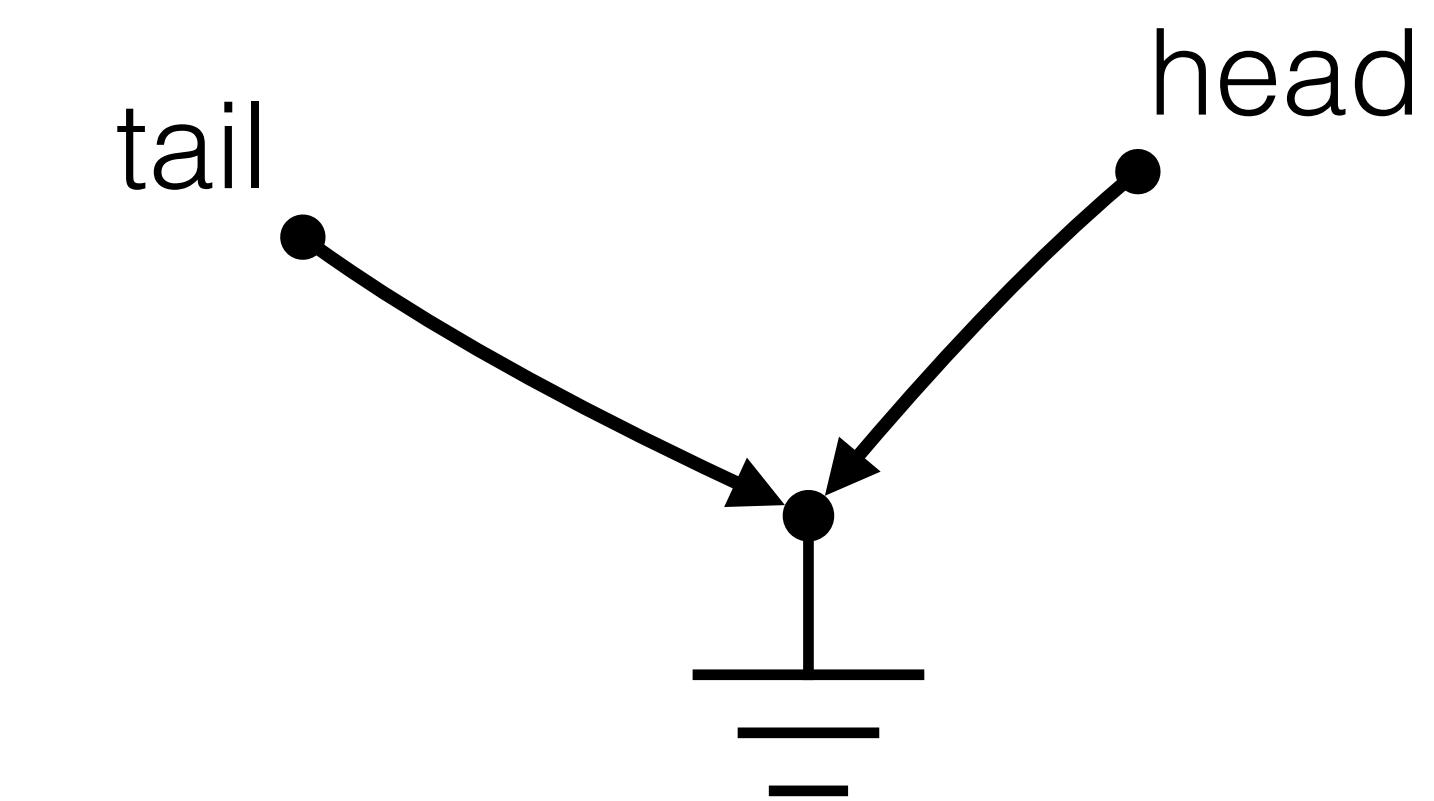
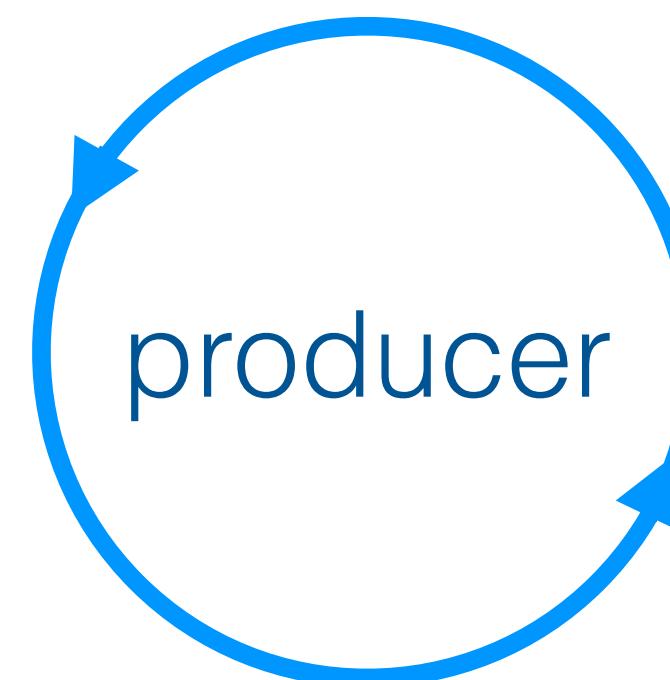
Empty buffer: consumer runs first

producer()

```
→ while true {  
    item = getEvent()  
    buffer.add(event)  
}
```

consumer()

```
→ while true {  
    event = buffer.get()  
    event.process()  
}
```



Empty buffer: consumer runs first

producer()

```
while true {
    item = getEvent()
    buffer.add(event)
}
```

signal()

consumer()

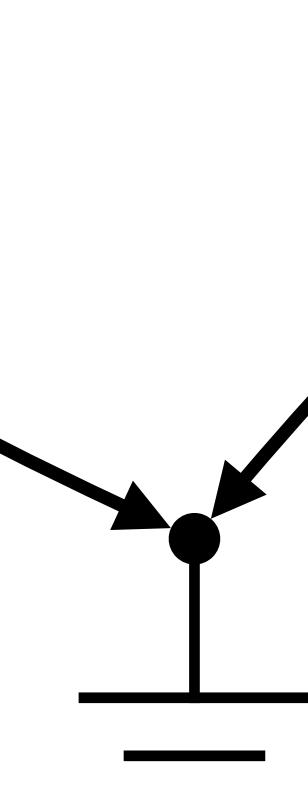
```
while true {
    event = buffer.get()
    event.process()
}
```

wait()



tail

head



Empty buffer: consumer runs first

producer()

```
while true {
    item = getEvent()
    buffer.add(event)
}
```

signal()

consumer()

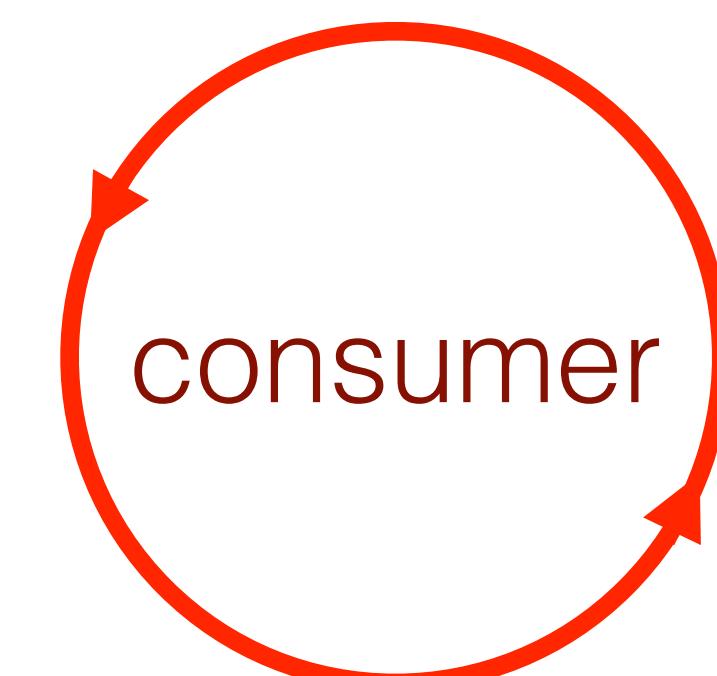
```
while true {
    items.wait()
    event = buffer.get()
    event.process()
}
```

wait()



tail

head



=

Empty buffer: consumer runs first

```
items = semaphore(0)
```

producer()

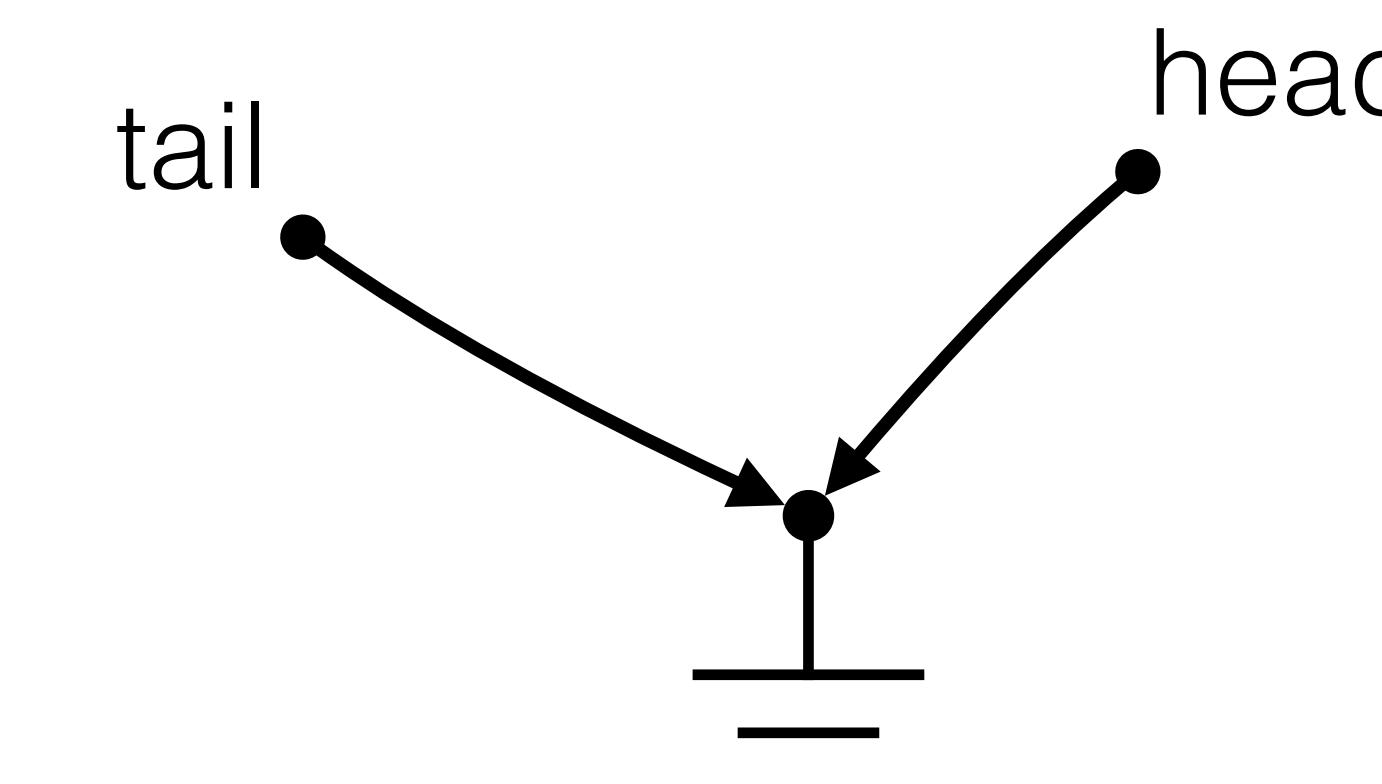
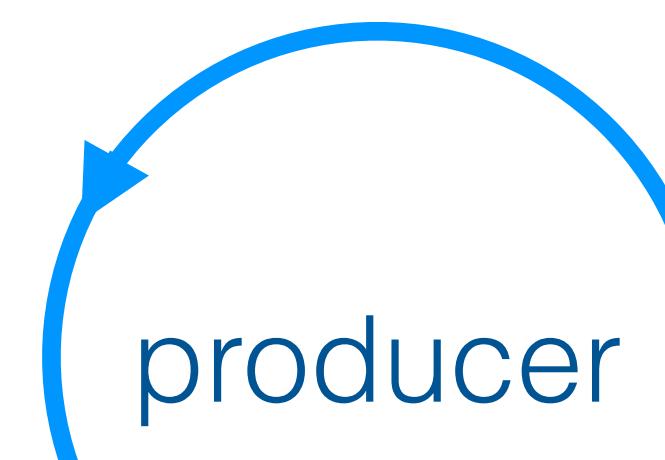
```
while true {
    item = getEvent()
    buffer.add(event)
}
```

signal()

consumer()

```
while true {
    items.wait()
    event = buffer.get()
    event.process()
}
```

wait()



Empty buffer: consumer runs first

```
items = semaphore(0)
```

producer()

```
while true {
    item = getEvent()
    buffer.add(event)
    items.signal()
}
```

signal()

consumer()

```
while true {
    items.wait()
    event = buffer.get()
    event.process()
}
```

wait()



tail

head



=

Empty buffer: consumer runs first

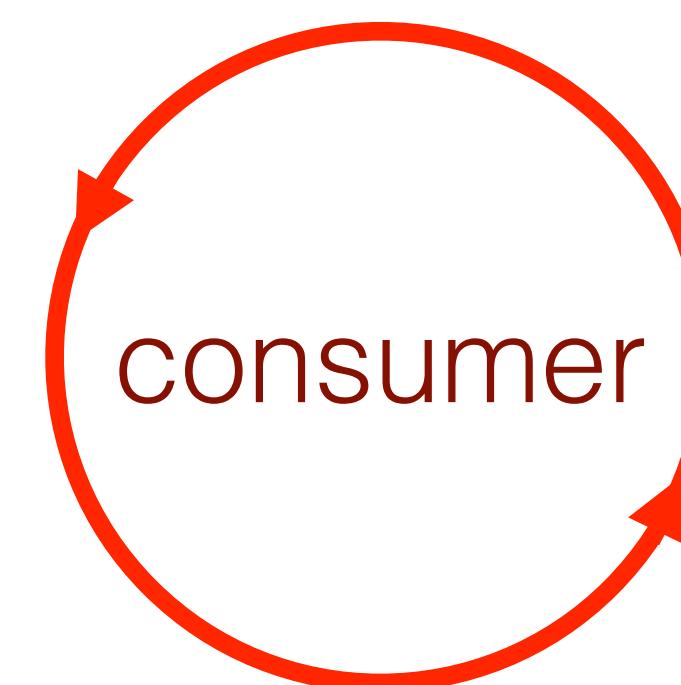
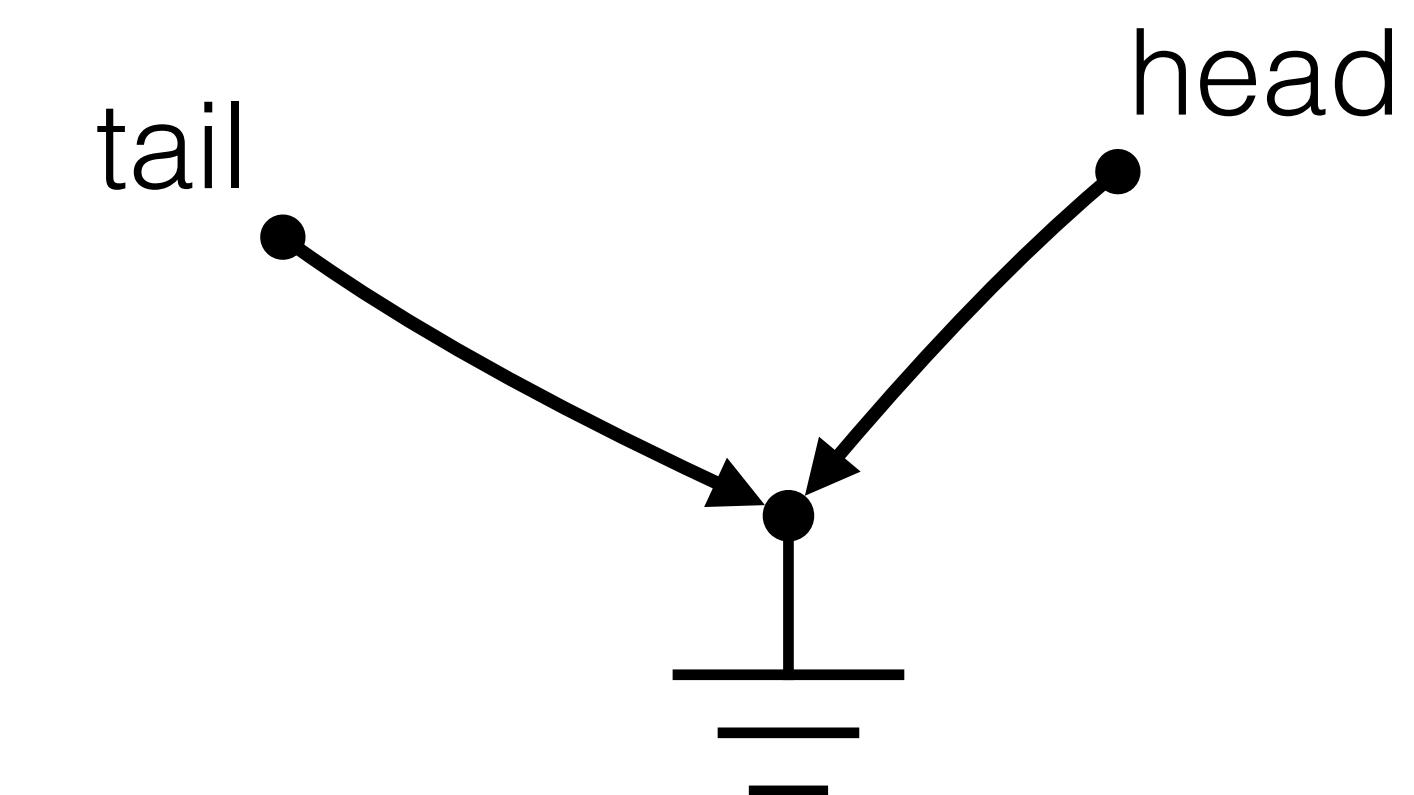
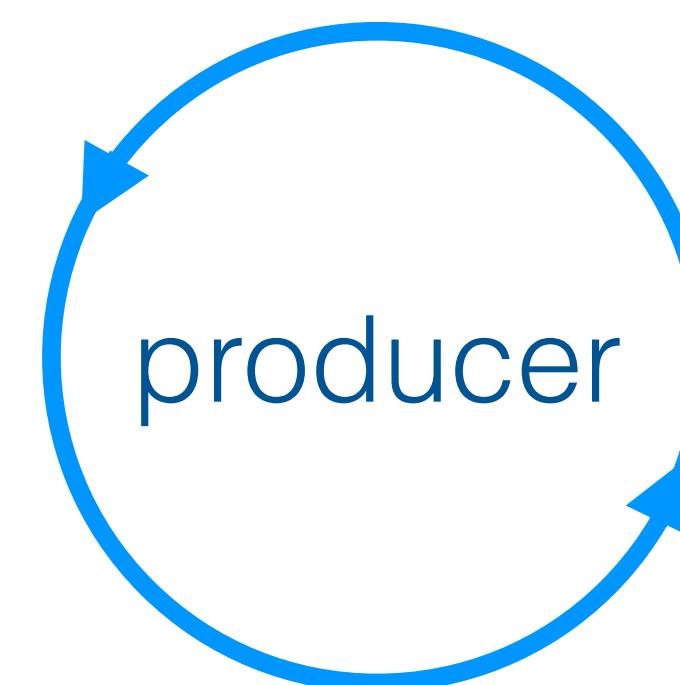
```
items = semaphore(0)
```

producer()

```
while true {
  item = getEvent()
  buffer.add(event)
  items.signal()
}
```

consumer()

```
while true {
  items.wait()
  event = buffer.get()
  event.process()
}
```



Empty buffer: consumer runs first

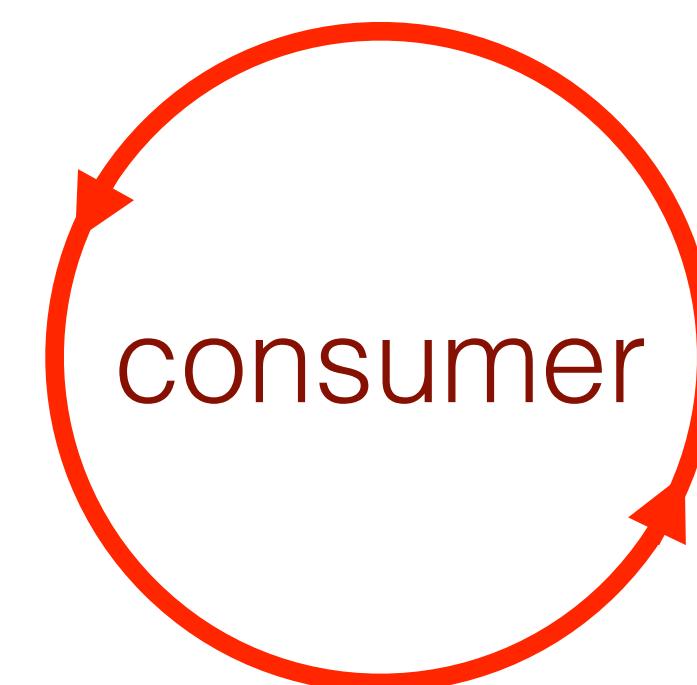
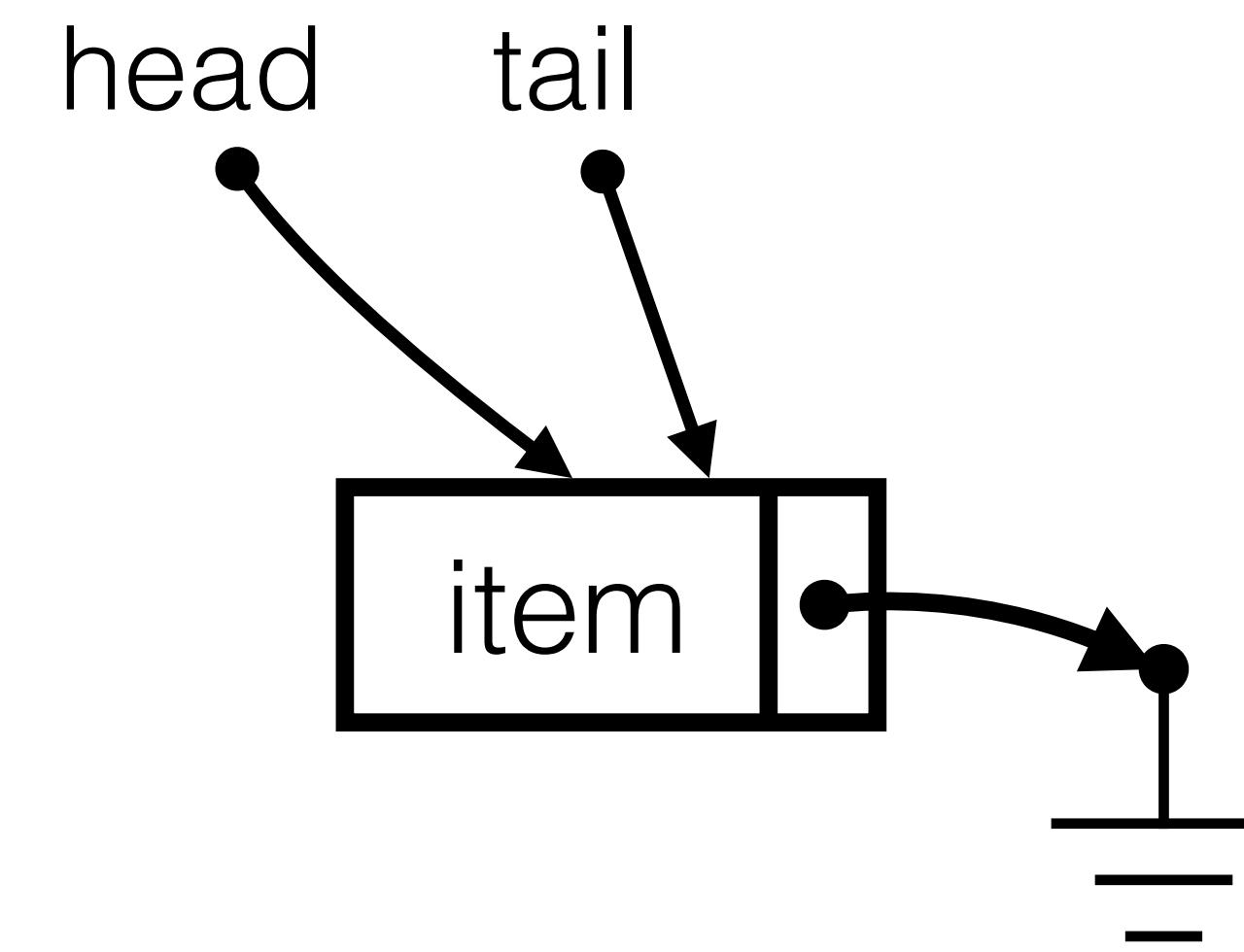
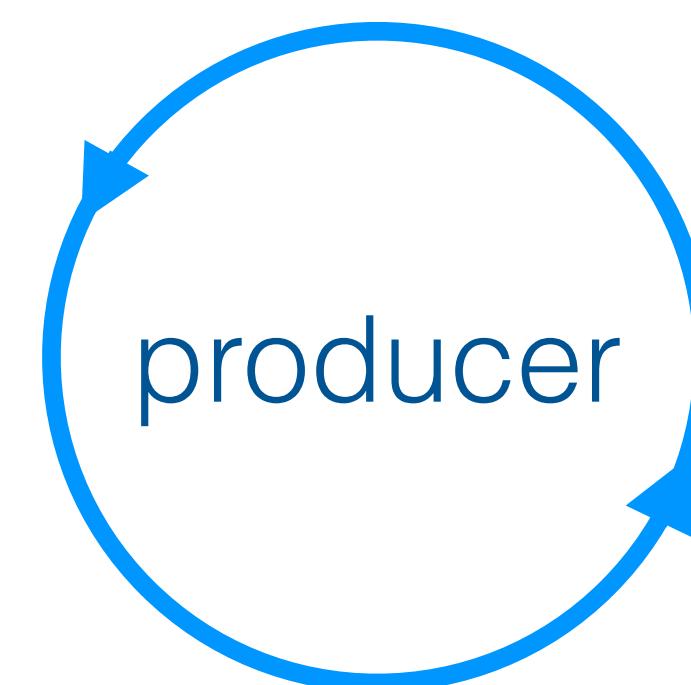
```
items = semaphore(0)
```

producer()

```
while true {
    item = getEvent()
    buffer.add(event)
    items.signal()
}
```

consumer()

```
while true {
    items.wait()
    event = buffer.get()
    event.process()
}
```



Concurrent writes: add and get cannot take place at the same time

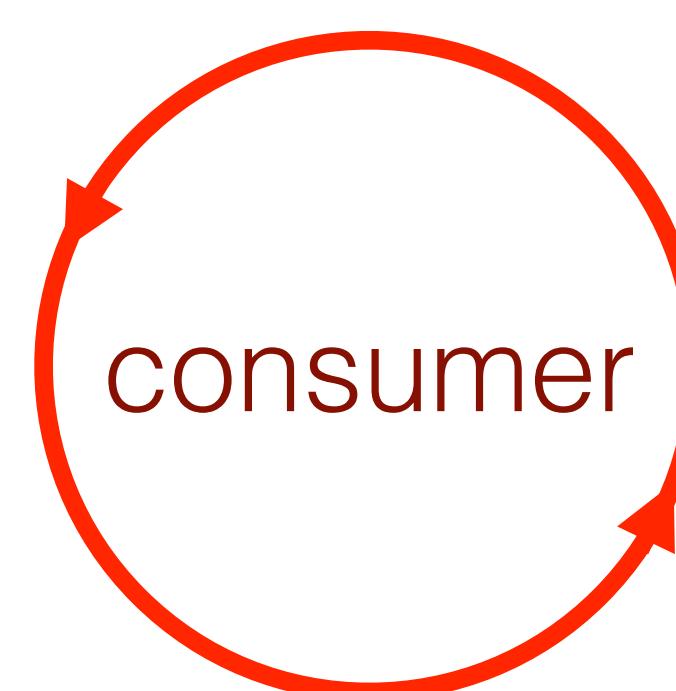
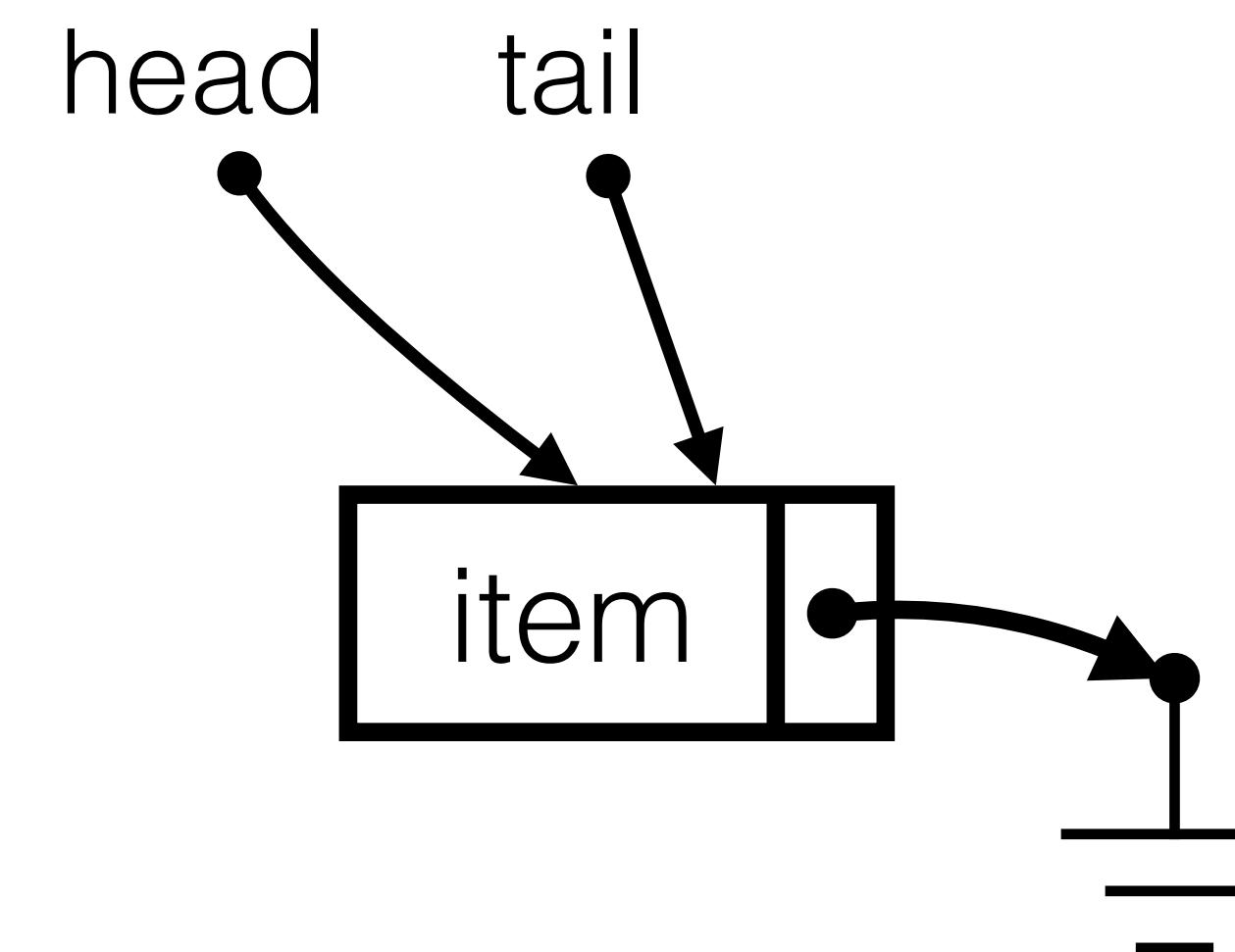
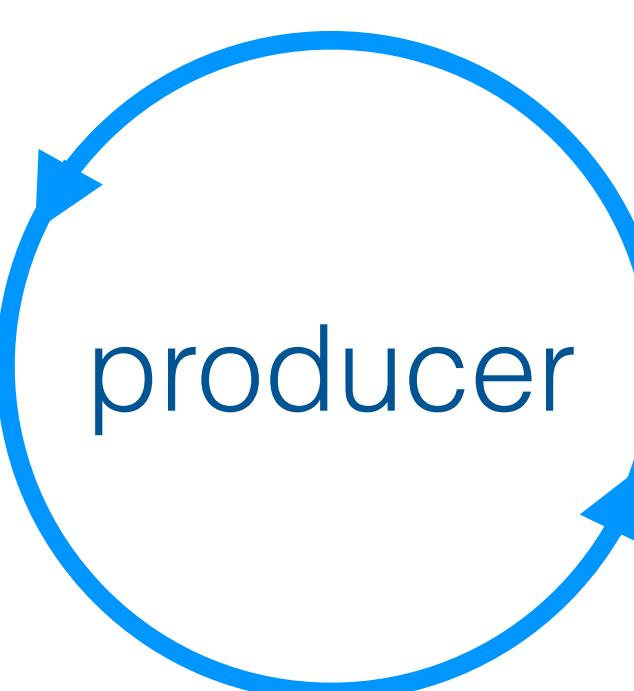
```
items = semaphore(0)
```

producer()

```
while true {  
    item = getEvent()  
    buffer.add(event)  
items.signal()  
}
```

consumer()

```
while true {  
items.wait()  
event = buffer.get()  
event.process()  
}
```



Concurrent writes: add and get cannot take place at the same time

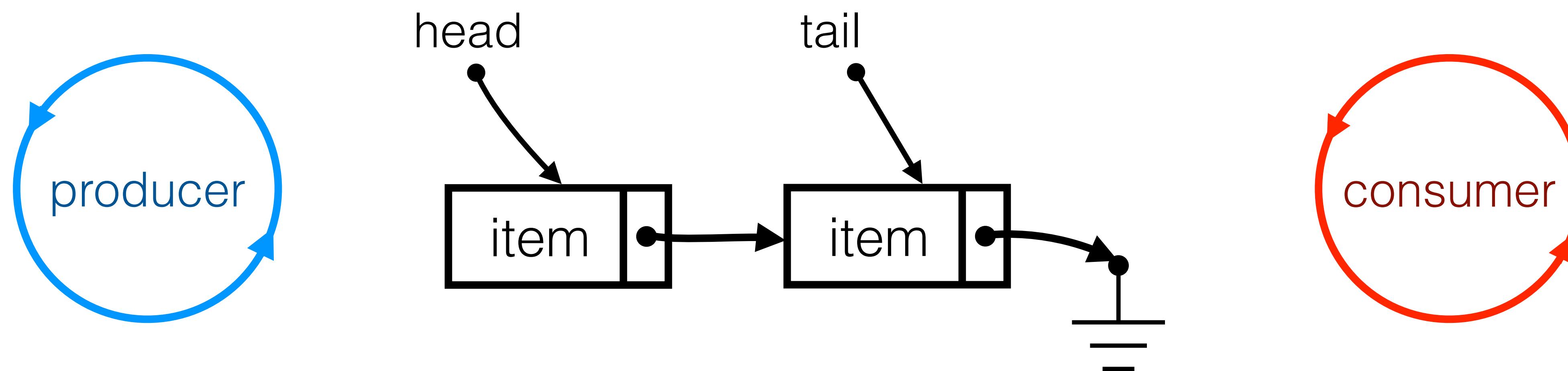
```
items = semaphore(0)
mutex = semaphore(1)
```

producer()

```
while true {
    item = getEvent()
    mutex.wait()
    buffer.add(event)
    mutex.signal()
    items.signal()
}
```

consumer()

```
while true {
    items.wait()
    mutex.wait()
    event = buffer.get()
    mutex.signal()
    event.process()
}
```



Limited buffer size: producer sleeps once the maximum buffer length is reached.

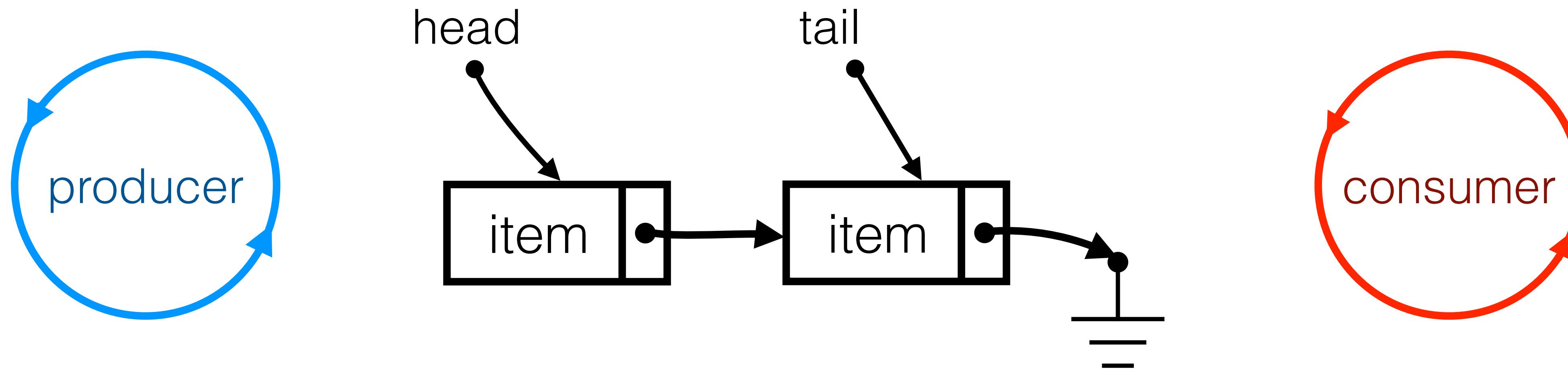
```
items = semaphore(0)
mutex = semaphore(1)
```

producer()

```
while true {
    item = getEvent()
    mutex.wait()
    buffer.add(event)
    mutex.signal()
    items.signal()
}
```

consumer()

```
while true {
    items.wait()
    mutex.wait()
    event = buffer.get()
    mutex.signal()
    event.process()
}
```



Limited buffer size: producer sleeps once the maximum buffer length is reached.

```
items = semaphore(0)
mutex = semaphore(1)
spaces = semaphore(buffer.size())
producer()
```

```
while true {
    item = getEvent()
    spaces.wait() ←
    mutex.wait()
    buffer.add(event)
    mutex.signal()
    items.signal()
}
```

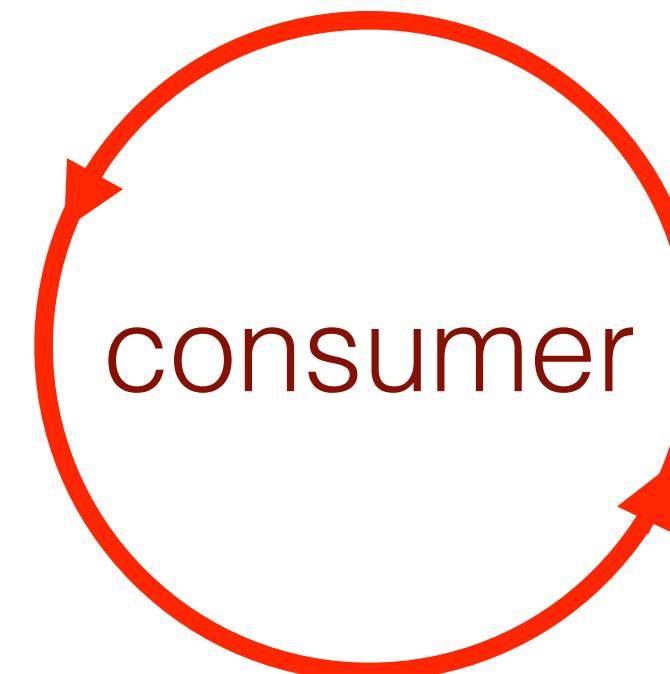
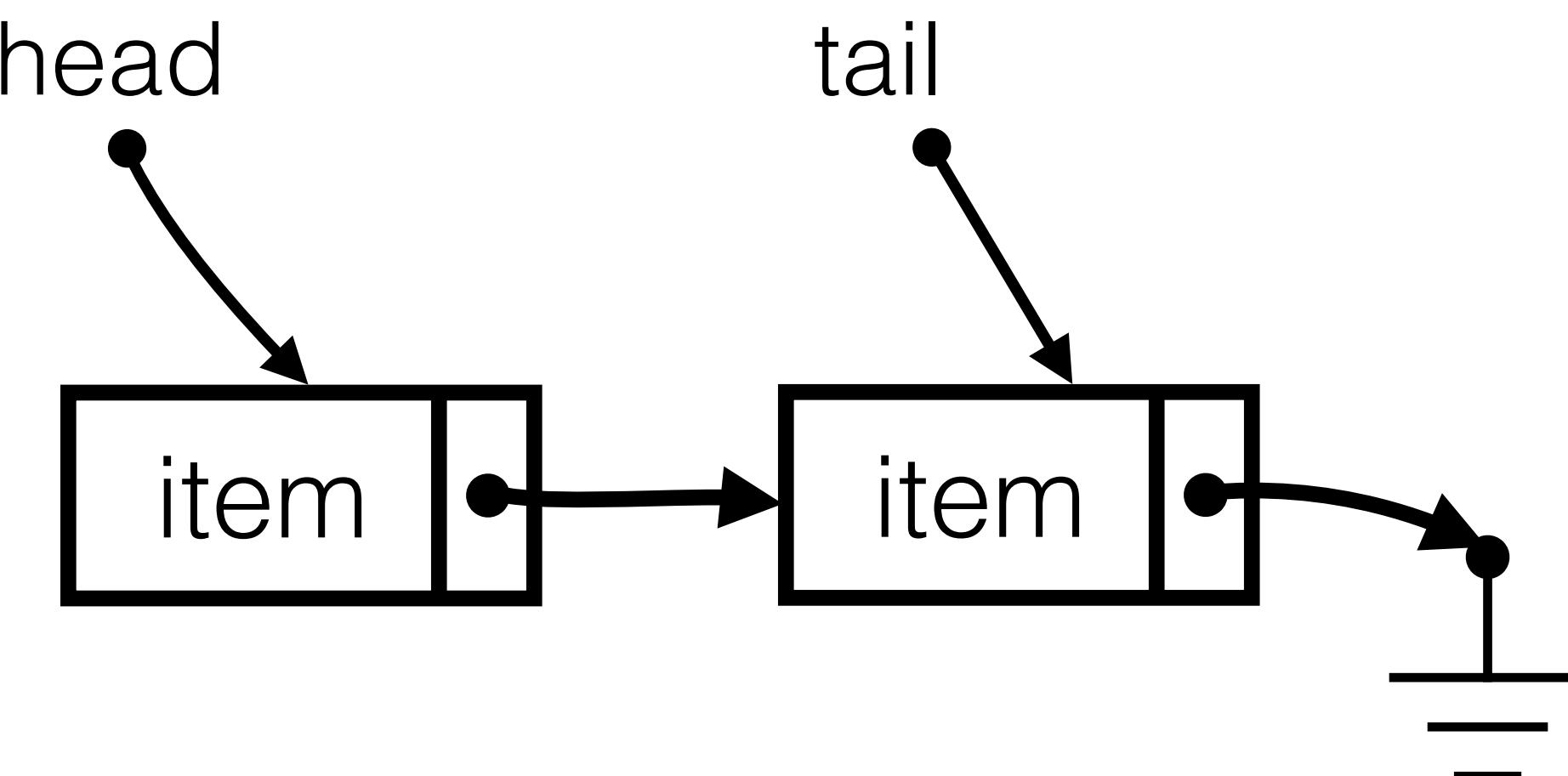
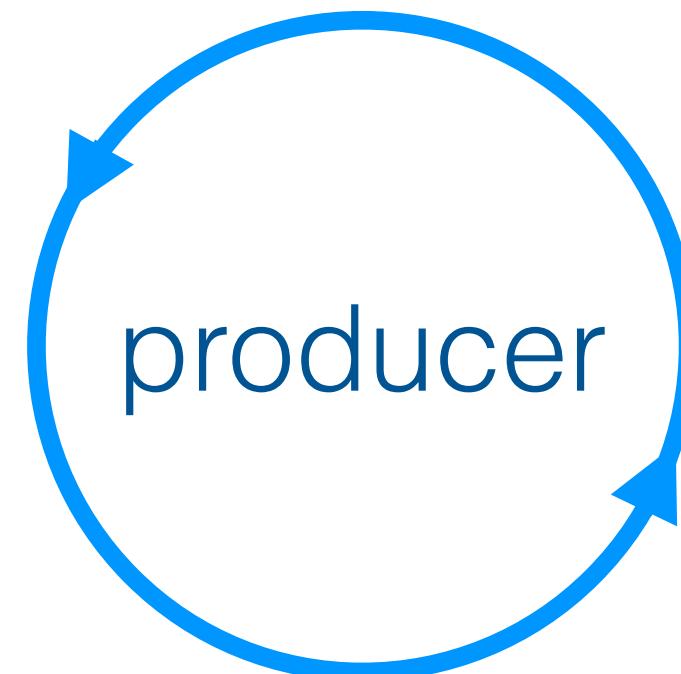
token interpretation
of the multiplex

wait()

signal()

consumer()

```
while true {
    items.wait()
    mutex.wait()
    event = buffer.get()
    mutex.signal()
    spaces.signal()
    event.process()
}
```



Limited buffer size: producer sleeps once the maximum buffer length is reached.

```
items = semaphore(0)
mutex = semaphore(1)
spaces = semaphore(buffer.size())
producer()
```

```
while true {
    item = getEvent()
    spaces.wait()
    mutex.wait()
    buffer.add(event)
    mutex.signal()
    items.signal()
}
```

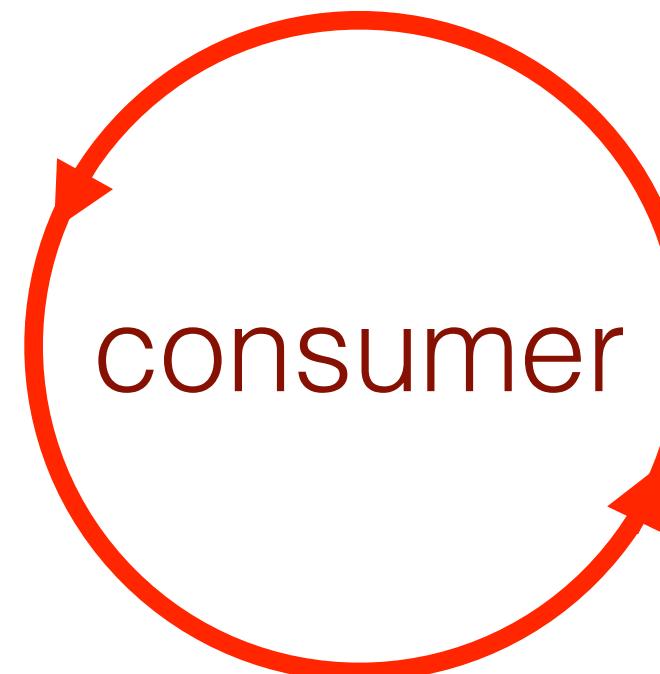
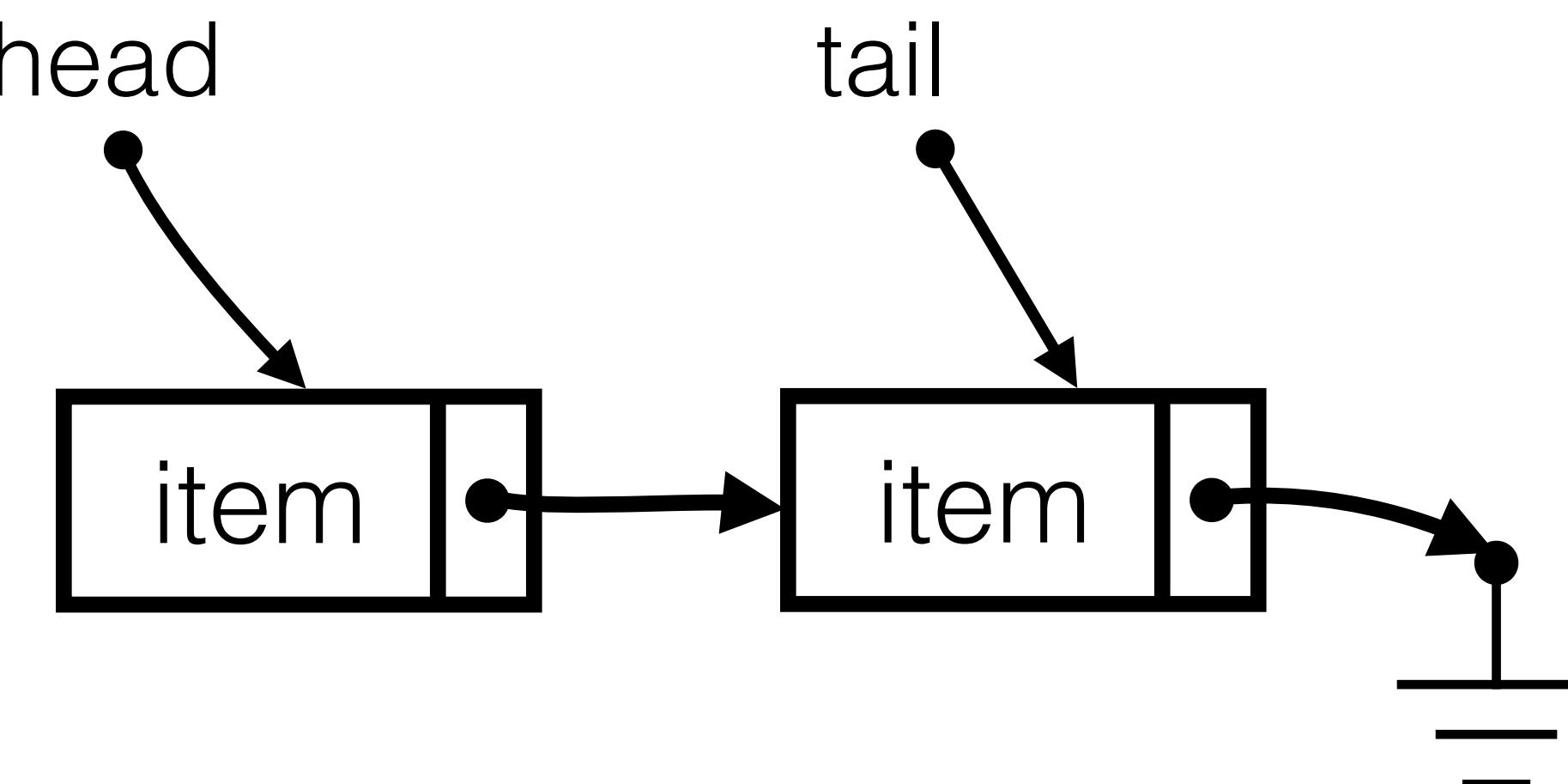
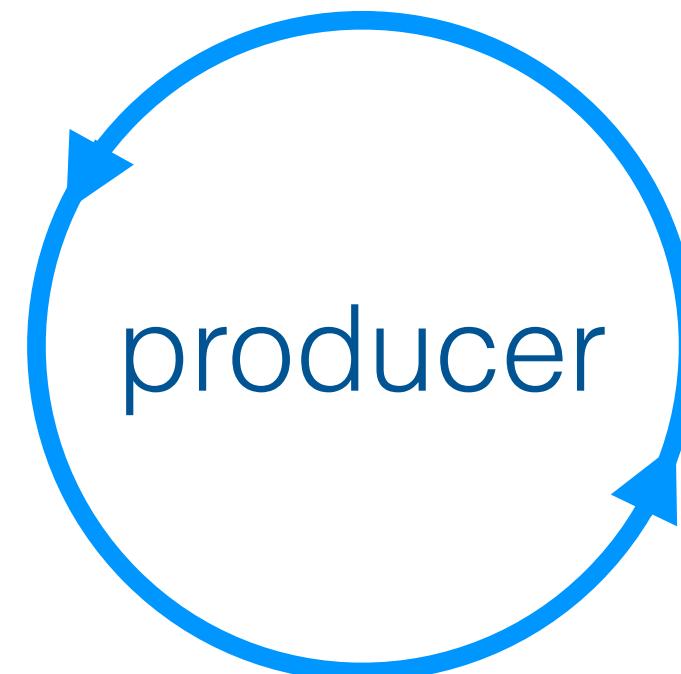
token interpretation
of the multiplex

wait()

signal()

consumer()

```
while true {
    items.wait()
    mutex.wait()
    event = buffer.get()
    mutex.signal()
    spaces.signal()
    event.process()
}
```



Implementation Example: Producer-Consumer

Global variables and semaphores

```
/* global vars */
const int bufferSize = 5;
const int numConsumers = 3;
const int numProducers = 3;

/* semaphores are declared global so they can be accessed
   in main() and in thread routine.*/
Semaphore Mutex(1);
Semaphore Spaces(bufferSize);
Semaphore Items(0);

int main(int argc, char **argv )
{
    pthread_t producerThread[ numProducers ];
    pthread_t consumerThread[ numConsumers ];
    ...
}
```

Implementation Example: Producer-Consumer

```
/*
    Producer function
*/
void *Producer ( void *threadID )
{
    // Thread number
    int x = (long)threadID;

    while( 1 )
    {
        sleep(3); // slow the thread down a bit so we can see what is going on
        Spaces.wait();
        Mutex.wait();
        printf("Producer %d adding item to buffer \n", x);
        fflush(stdout);
        Mutex.signal();
        Items.signal();
    }

}
```

Producer thread

Implementation Example: Producer-Consumer

```
/*
 * Consumer function
 */
void *Consumer ( void *threadID )
{
    // Thread number
    int x = (long)threadID;

    while( 1 )
    {
        Items.wait();
        Mutex.wait();
        printf("Consumer %d removing item from buffer \n", x);
        fflush(stdout);
        Mutex.signal();
        Spaces.signal();
        sleep(5);    // slow the thread down a bit so we can see what is going on
    }

}
```

Consumer thread

Thread Synchronization (Part 3)

CSE 4001

Contents

- The readers-writers problem

Readers-Writers

A data set is shared among a number of concurrent threads

- **Readers:** Only read the data set; they do not perform any updates
- **Writers:** Can both read and write

Problem: Allow multiple readers to read at the same time. Only one single writer can access the shared data at any time.

Readers-Writers

Here is a set of variables that is sufficient to solve the problem

```
int readers = 0           // no. of readers in the room
mutex = Semaphore(1)      // protects the counter
roomEmpty = Semaphore(1)  // 1 if room is empty
```

Readers-Writers

Writer

```
roomEmpty.wait()  
    critical section for writers  
roomEmpty.signal()
```

Readers-Writers

Reader

```
mutex.wait()
    readers += 1
    if readers == 1:
        roomEmpty.wait() # first in locks
    endif
mutex.signal()

    # critical section for readers

mutex.wait()
    readers -= 1
    if readers == 0:
        roomEmpty.signal() # last out unlocks
    endif
mutex.signal()
```

Readers-Writers

Reader

```

mutex.wait()
  readers += 1
  if readers == 1:
    roomEmpty.wait() # first in locks
  endif
mutex.signal()
# critical section for readers
mutex.wait()
  readers -= 1
  if readers == 0:
    roomEmpty.signal() # last out unlocks
  endif
mutex.signal()

```

Reader thread

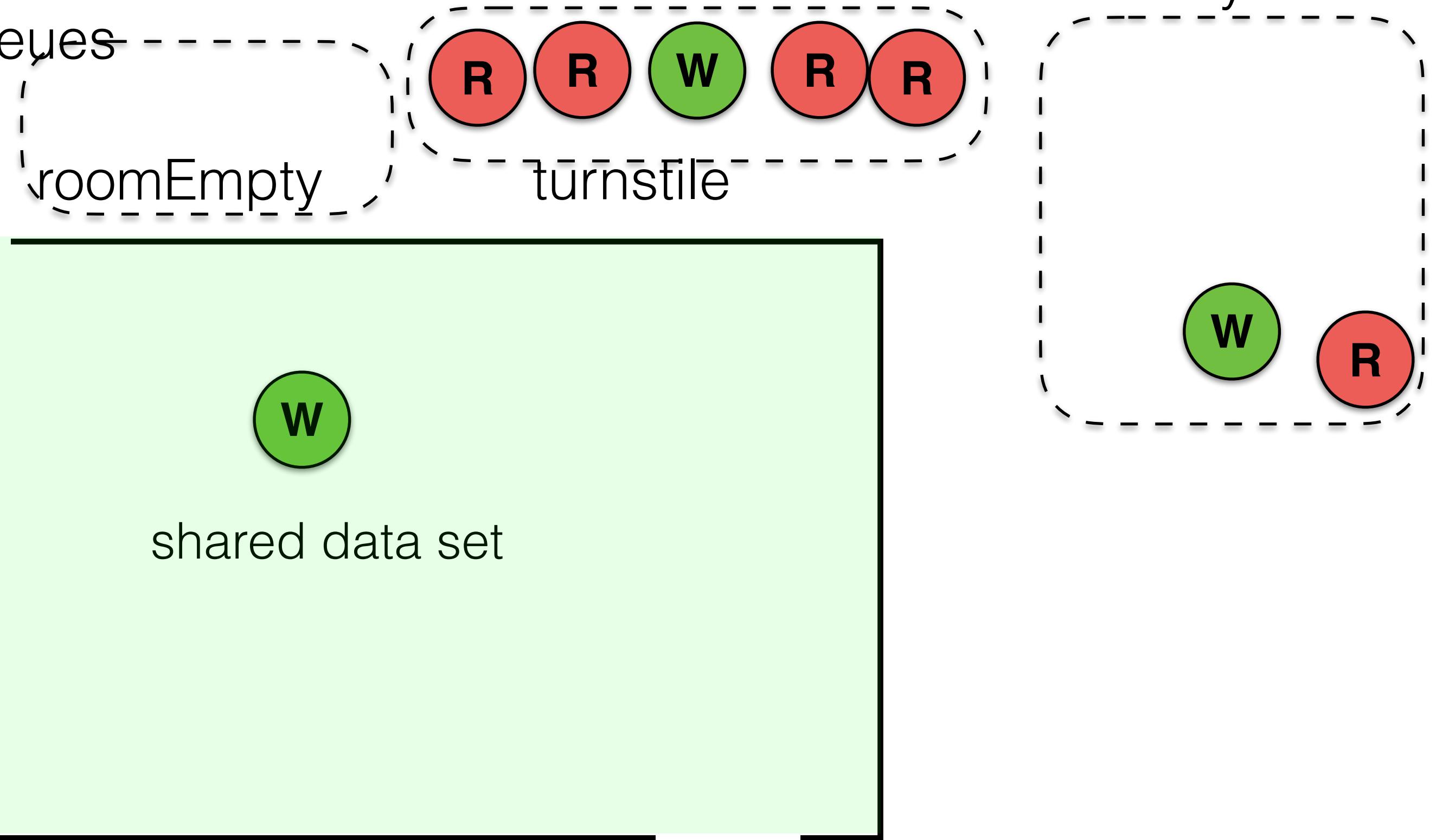
```

readSwitch.lock(roomEmpty)
  # critical section for readers
readSwitch.unlock(roomEmpty)

```

Scenario 2: The scheduler picks a reader.

waiting queues



writer thread

```

1 turnstile.wait()
2   roomEmpty.wait()
3   # critical section for writers
4 turnstile.signal()
5
6 roomEmpty.signal()

```

reader thread

```

1 turnstile.wait()
2 turnstile.signal()
3
4 readSwitch.lock(roomEmpty)
5   # critical section for readers
6 readSwitch.unlock(roomEmpty)

```

Readers-Writers

Depending on the application, it might be a good idea to give more priority to writers. For example, if writers are making time-critical updates to a data structure, it is best to minimize the number of readers that see the old data before the writer has a chance to proceed.

Readers-Writers

```
readSwitch = Lightswitch()  
roomEmpty = Semaphore(1)  
turnstile = Semaphore(1)
```

turnstile is a turnstile for readers and a mutex for writers. Readers will need to queue on the turnstile if a writer gets stuck inside it.

Readers-Writers

writer thread

```
1 turnstile.wait()  
2     roomEmpty.wait()  
3     # critical section for writers  
4 turnstile.signal()  
5  
6 roomEmpty.signal()
```

reader thread

```
1 turnstile.wait()  
2 turnstile.signal()  
3  
4 readSwitch.lock(roomEmpty)  
5     # critical section for readers  
6 readSwitch.unlock(roomEmpty)
```

Thread Synchronization (Part 4)

CSE 4001

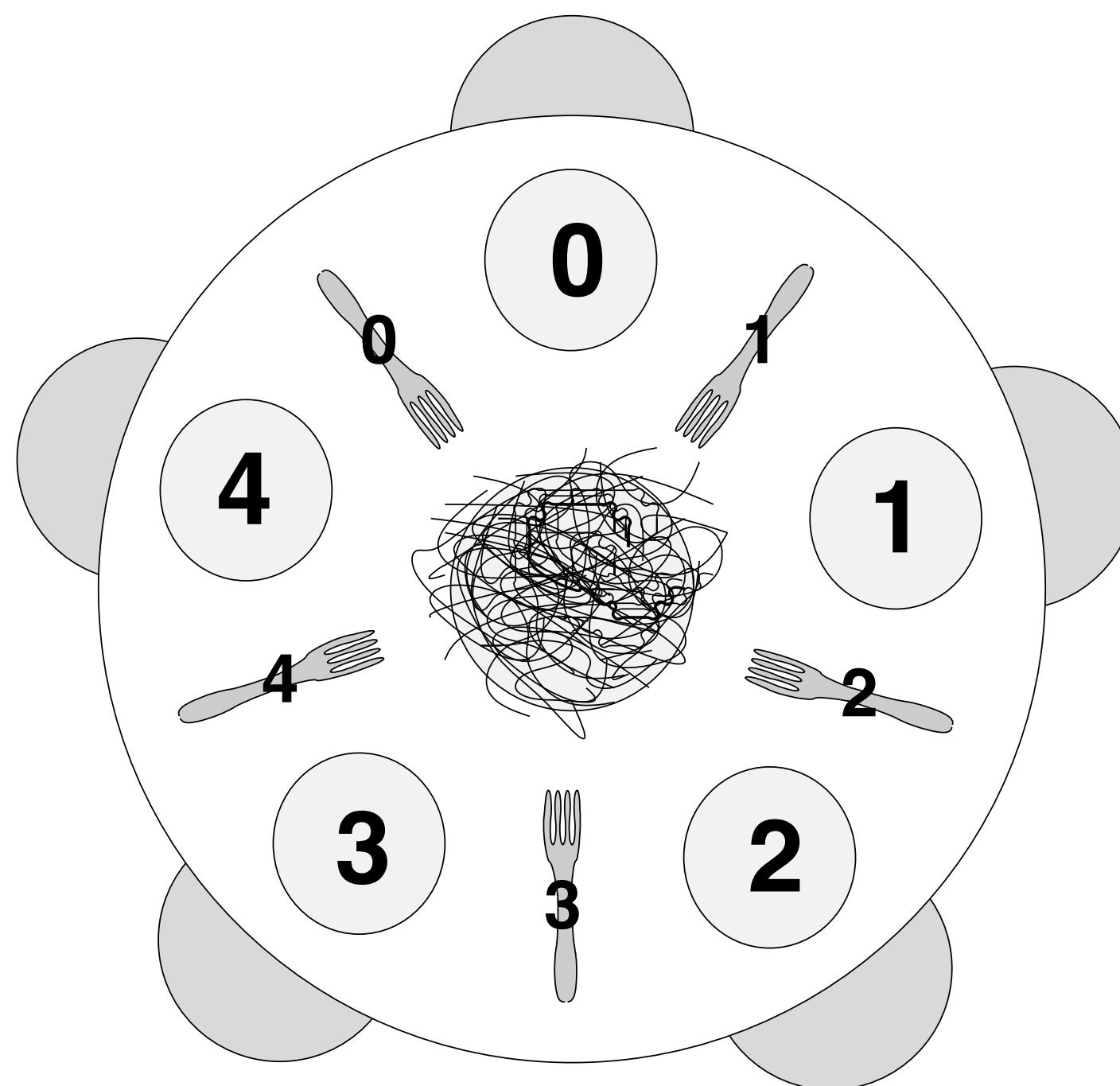
Contents

- The readers-writers problem
- **The dinning-philosophers problem**

Dining philosophers

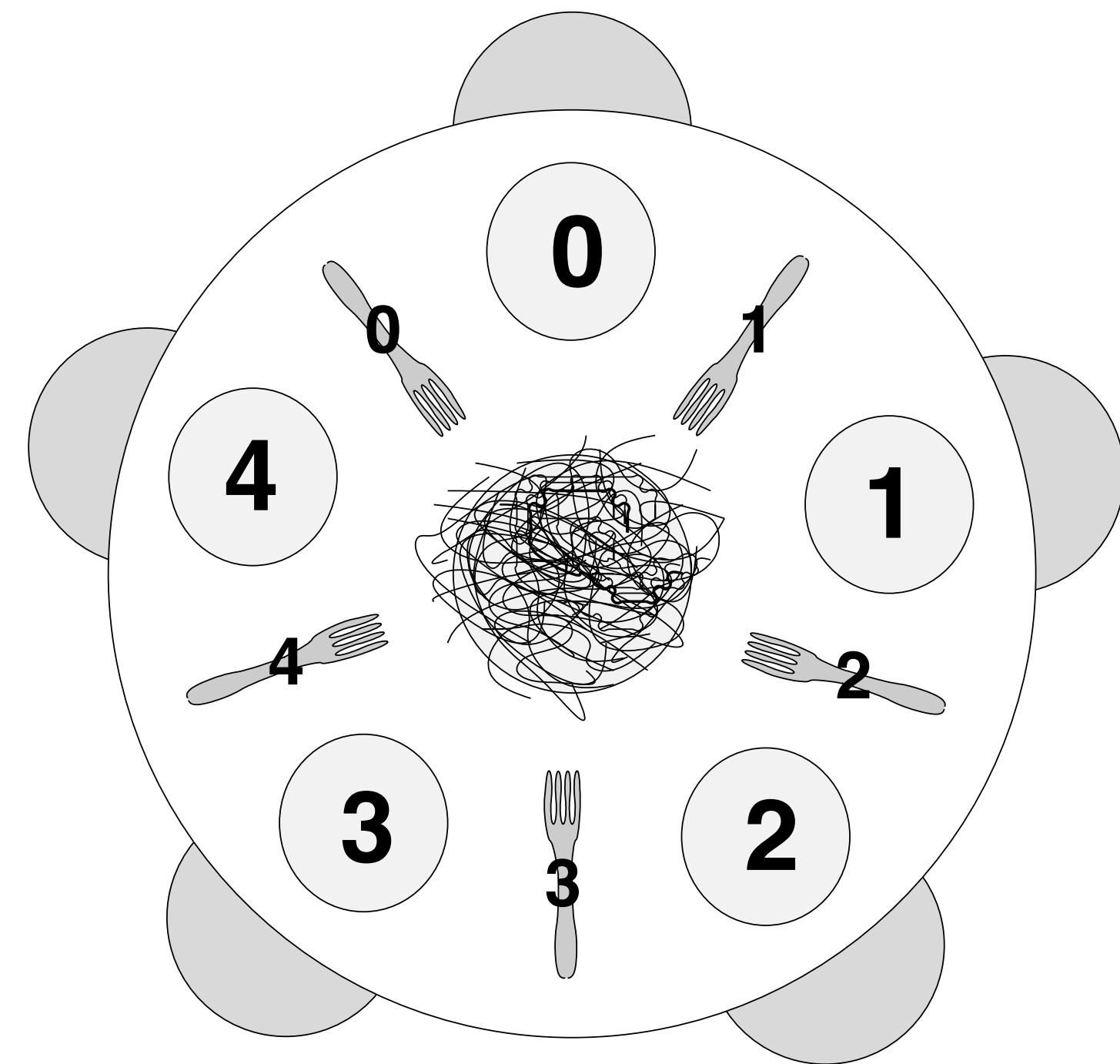
The Dining Philosophers Problem was proposed by Dijkstra in 1965. The standard version features are a table with five plates, five forks (or chopsticks) and a big bowl of spaghetti. Five philosophers, who represent interacting threads, come to the table and execute the following loop:

```
while True:  
    think()  
    get_forks()  
    eat()  
    put_forks()
```



Dining philosophers

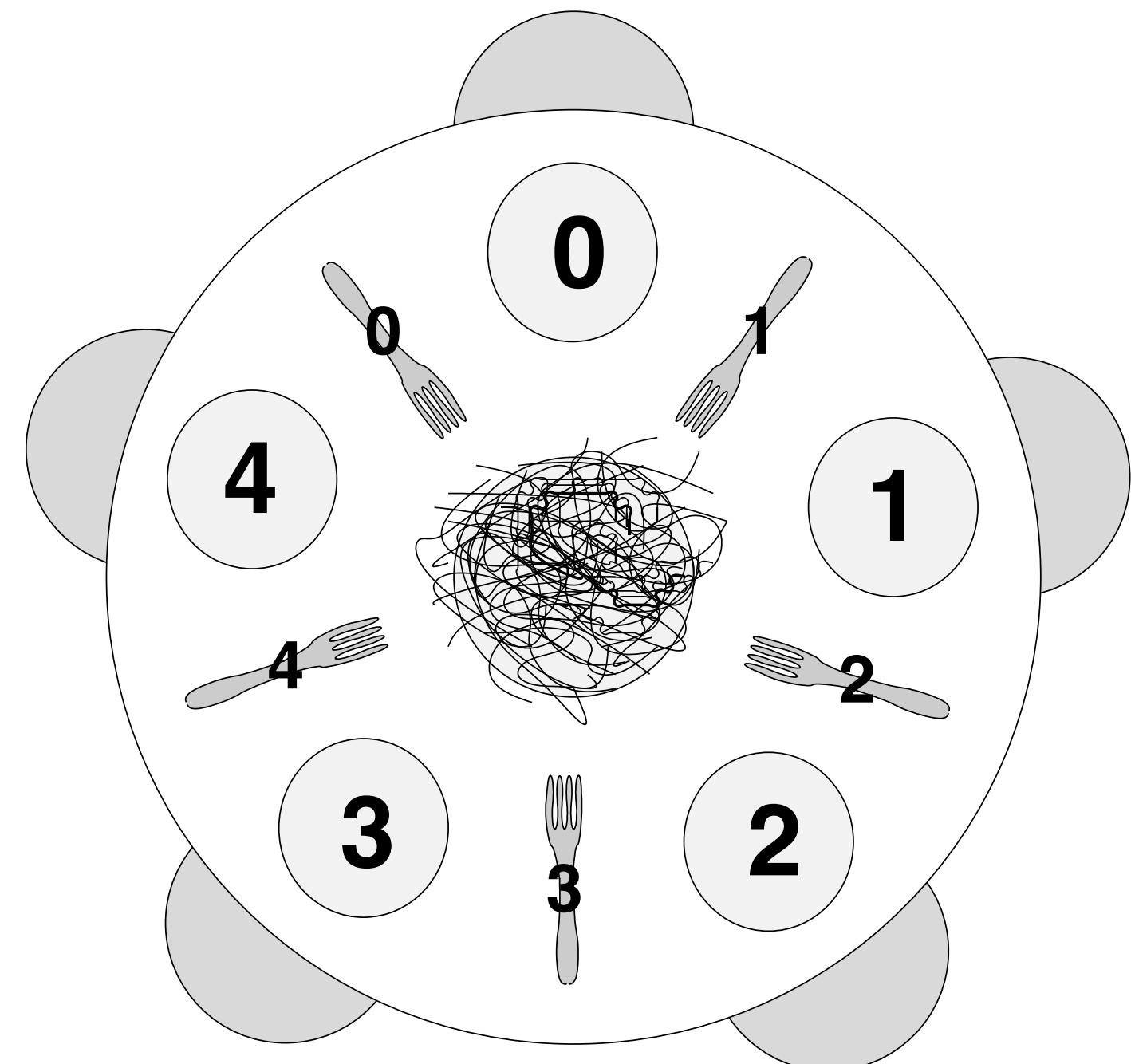
- The forks represent resources that the threads have to hold exclusively in order to make progress.
- The philosophers need *two* forks to eat, so a hungry philosopher might have to wait for a neighbor to put down a fork.



Dining philosophers

Assuming that the philosophers know how to think and eat, our job is to write a version of `get_forks` and `put_forks` that satisfies the following constraints:

- Only one philosopher can hold a fork at a time.
- It must be impossible for a deadlock to occur.
- It must be impossible for a philosopher to starve waiting for a fork.
- It must be possible for more than one philosopher to eat at the same time.



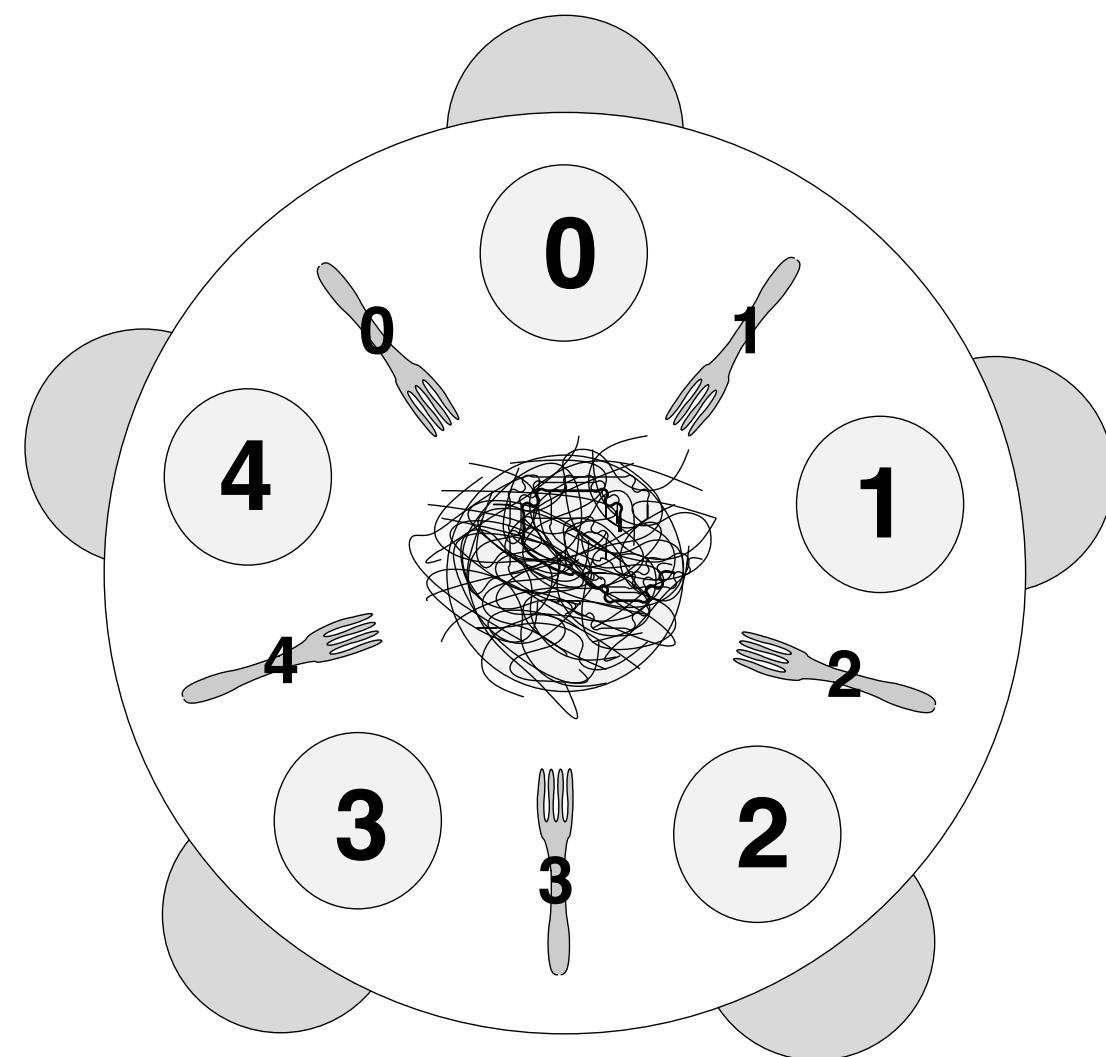
Dining philosophers

Let us define the functions `left` and `right` to refer to the forks' position.

```
def left(i): return i
def right(i): return (i + 1) % 5
```

Use a list of Semaphores, one for each fork. Initially, all the forks are available.

```
forks = [Semaphore(1) for i in range(5)]
```

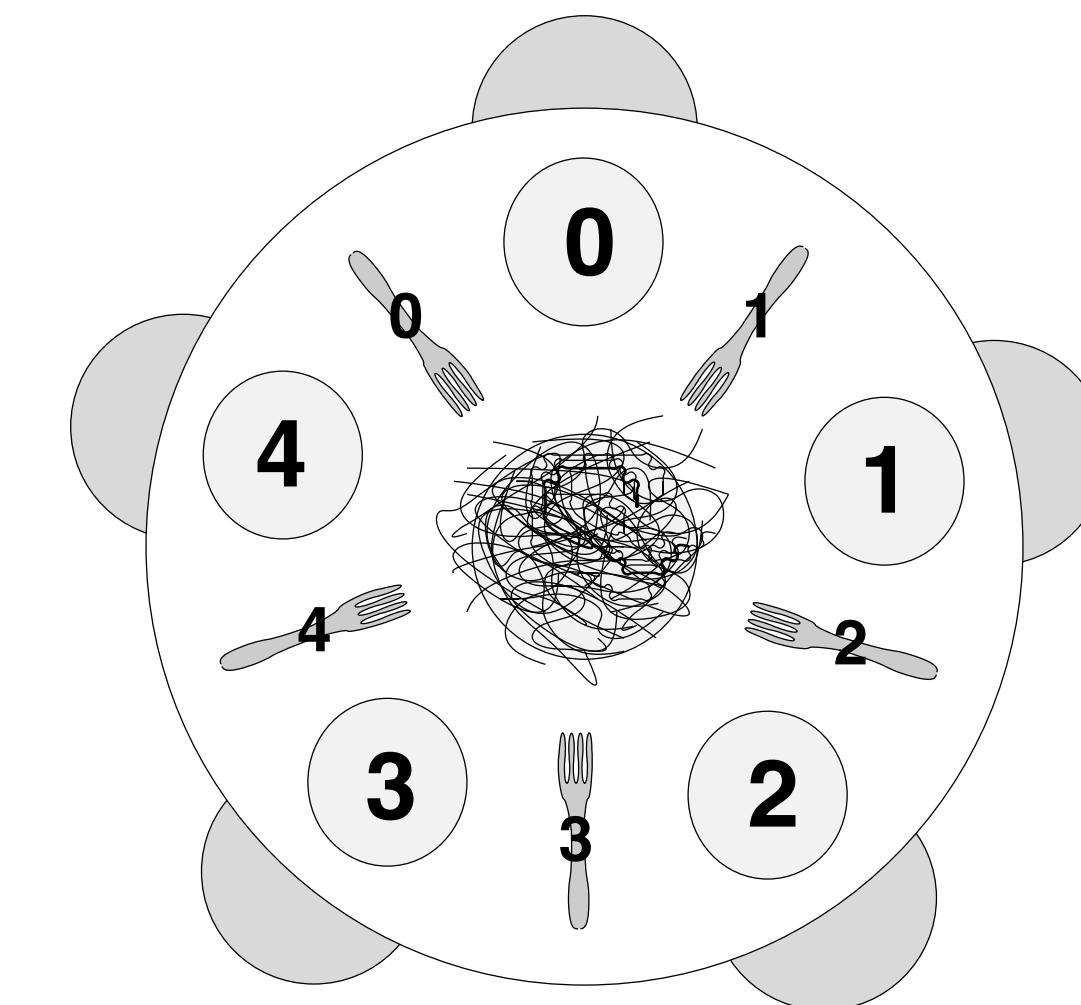


Dining philosophers

First attempt of a solution:

```
def get_forks(i):
    fork[right(i)].wait()
    fork[left(i)].wait()

def put_forks(i):
    fork[right(i)].signal()
    fork[left(i)].signal()
```



Which constraints are satisfied by this solution?

- ① Only one philosopher can hold a fork at a time.
- ② It must be impossible for a deadlock to occur.
- ③ It must be impossible for a philosopher to starve waiting for a fork.
- ④ It must be possible for more than one philosopher to eat at the same time.

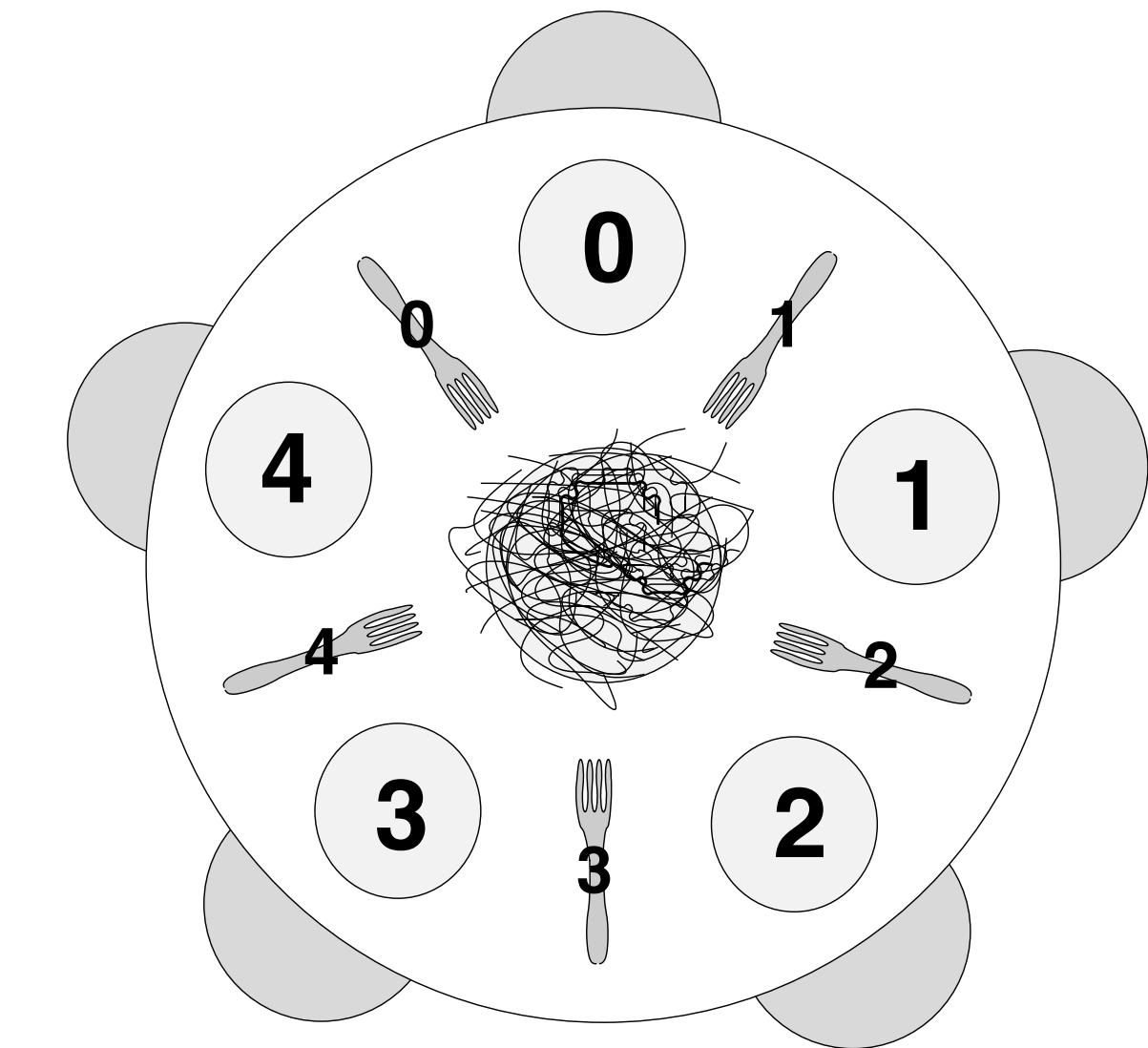
Dining philosophers

- **Limit the number of philosophers at the table at a time:** If only **four** philosophers are allowed at the table at a time, deadlock is impossible. There is always a fork on the table.
- We can control the number of philosophers at the table with a Multiplex named footman that is initialized to 4.

Dining philosophers

```
def get_forks(i):
    footman.wait()
    fork[right(i)].wait()
    fork[left(i)].wait()

def put_forks(i):
    fork[right(i)].signal()
    fork[left(i)].signal()
    footman.signal()
```



Which constraints are satisfied by this solution?

- 1 Only one philosopher can hold a fork at a time.
- 2 It must be impossible for a deadlock to occur.
- 3 It must be impossible for a philosopher to starve waiting for a fork.
- 4 It must be possible for more than one philosopher to eat at the same time.

Dining philosophers

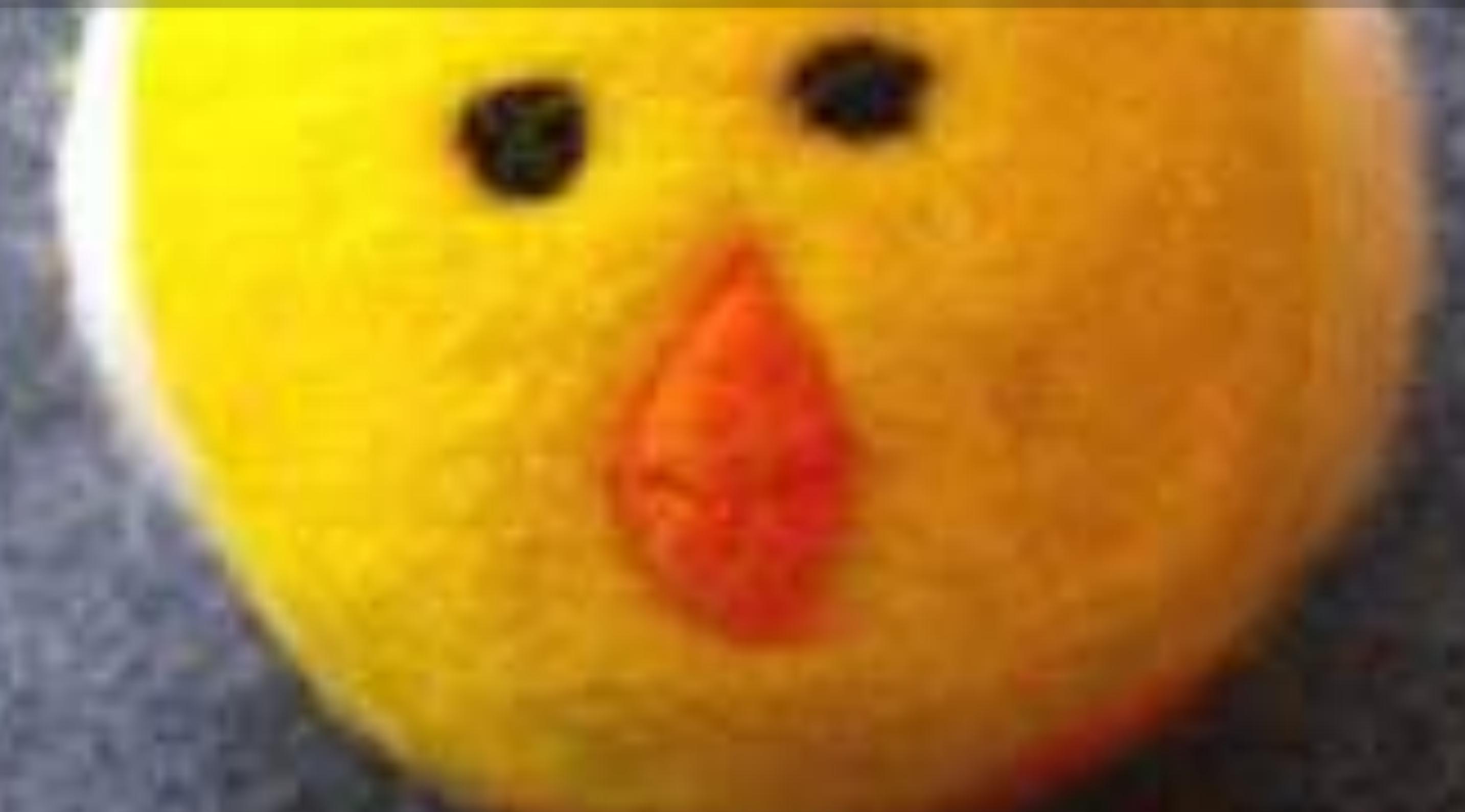
- Another way to avoid deadlock is to change the order in which the philosophers pick up forks. In the original non-solution, the philosophers are “righties”; that is, they pick up the right fork first. But what happens if Philosopher 0 is a leftie?

scheduling

THE CRUX: HOW TO DEVELOP SCHEDULING POLICY

How should we develop a basic framework for thinking about scheduling policies? What are the key assumptions? What metrics are important? What basic approaches have been used in the earliest of computer systems?

“Well, I found a solution to your problem with the chickens. But, the solution only works for spherical chickens in the vacuum and with a uniform mass distribution...”



Initial (simplifying) assumptions



1. Each job runs for the same amount of time.
2. All jobs arrive at the same time.
3. Once started, each job runs to completion.
4. All jobs only use the CPU (i.e., they perform no I/O)
5. The run-time of each job is known.

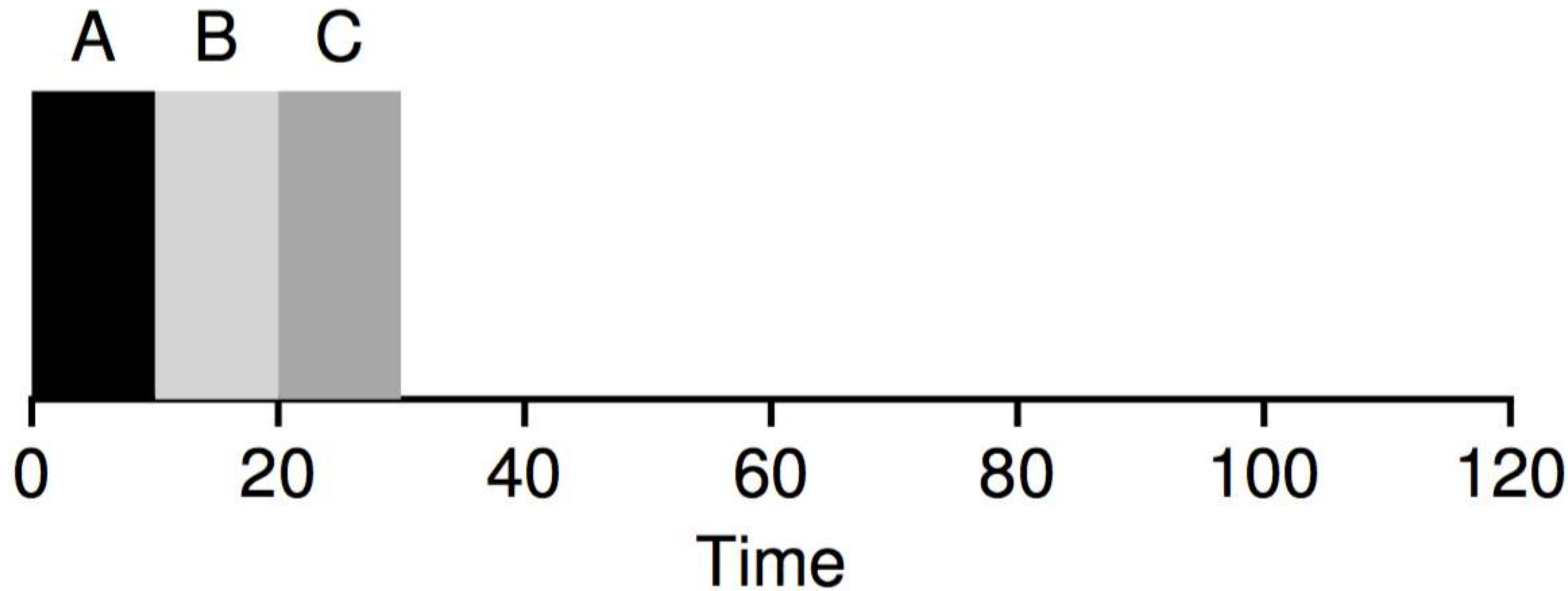


Figure 7.1: FIFO Simple Example

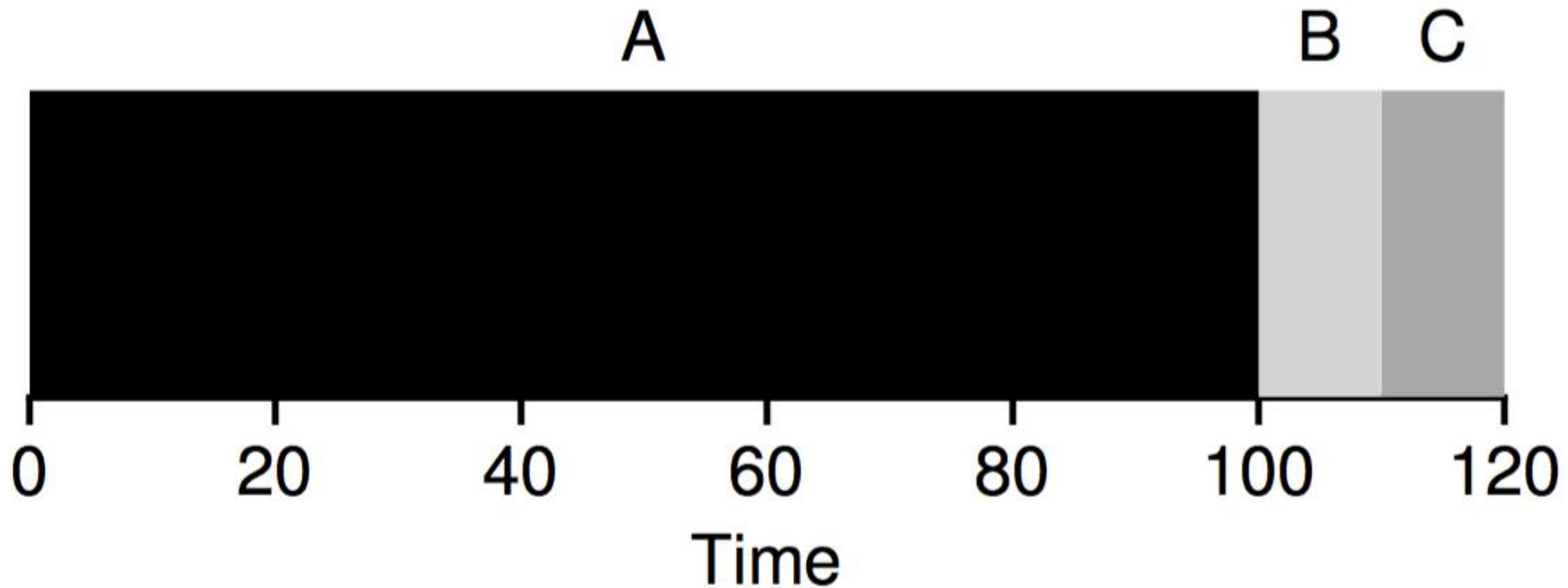


Figure 7.2: Why FIFO Is Not That Great

Shortest Job First

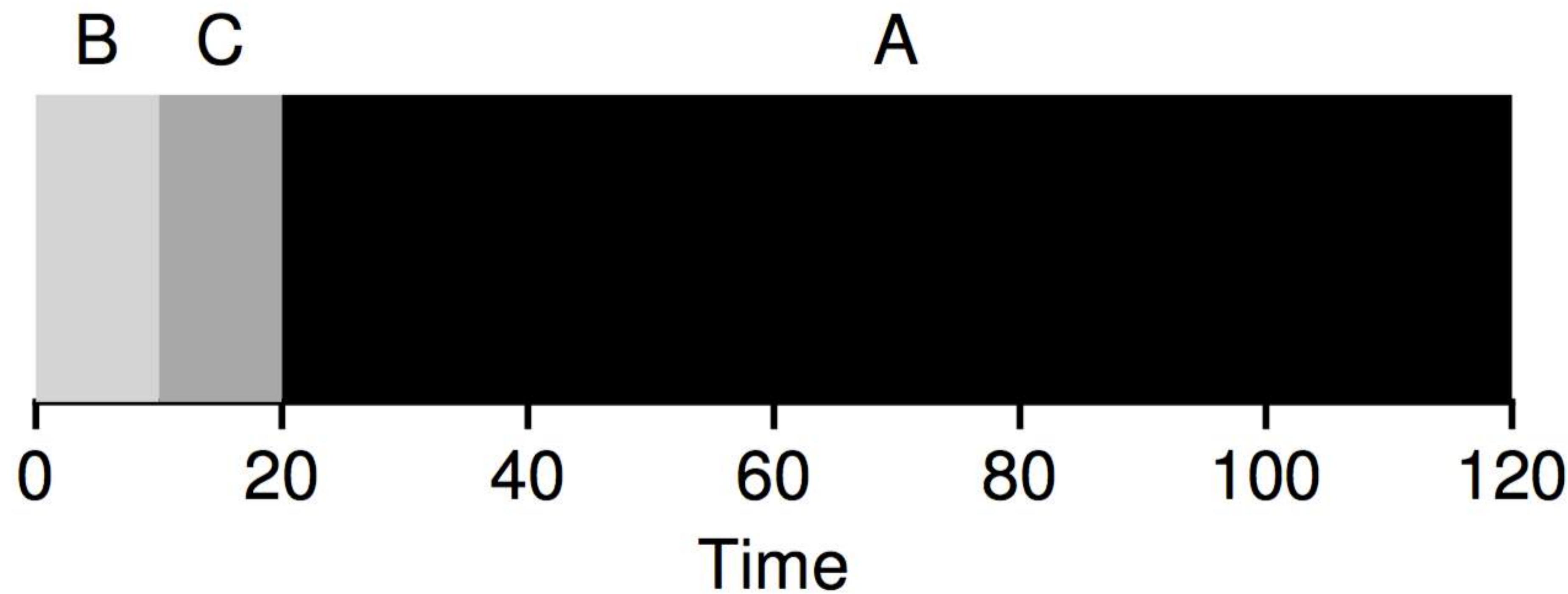


Figure 7.3: SJF Simple Example

Initial (simplifying) assumptions



1. Each job runs for the same amount of time.
- ~~2. All jobs arrive at the same time.~~
3. Once started, each job runs to completion.
4. All jobs only use the CPU (i.e., they perform no I/O)
5. The run-time of each job is known.

Shortest Job First: different arrival times

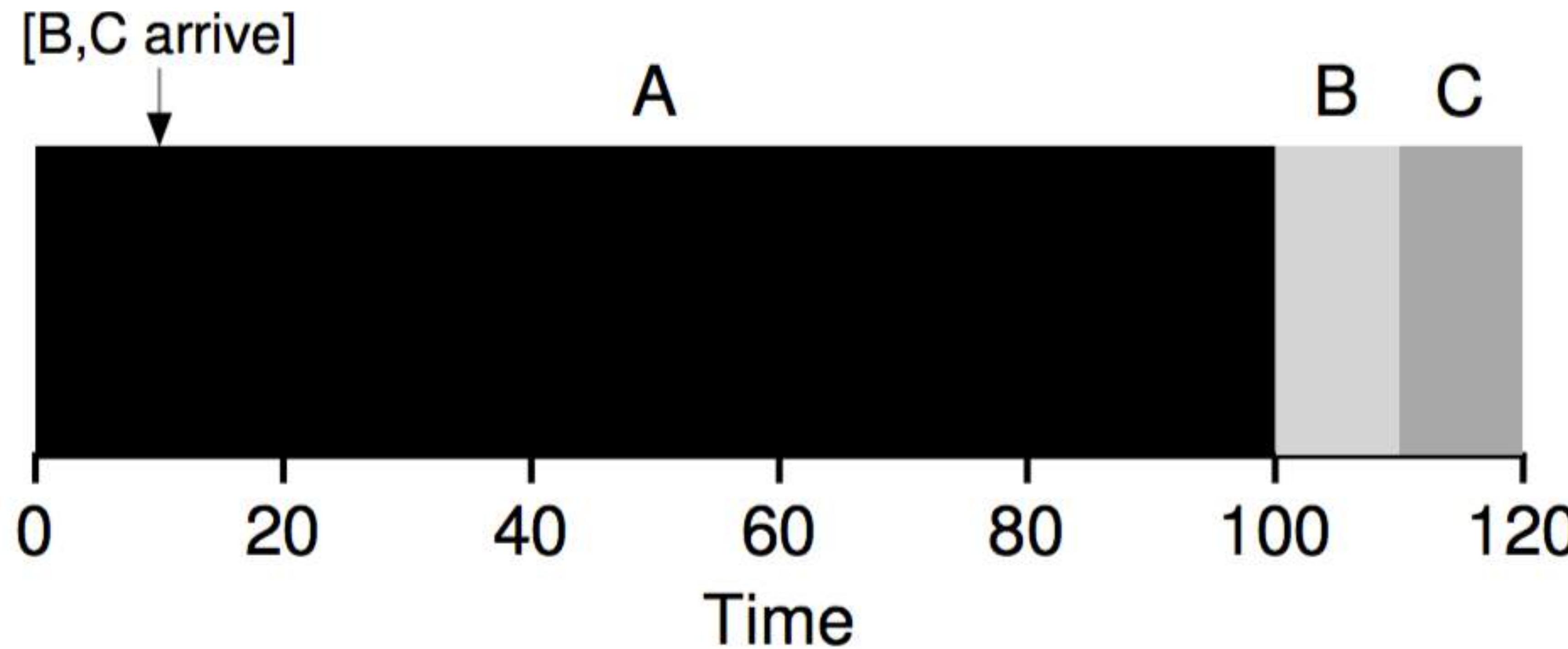


Figure 7.4: SJF With Late Arrivals From B and C

Initial (simplifying) assumptions



1. Each job runs for the same amount of time.
- ~~2. All jobs arrive at the same time.~~
- ~~3. Once started, each job runs to completion.~~
4. All jobs only use the CPU (i.e., they perform no I/O)
5. The run-time of each job is known.

Shortest Time to Completion First

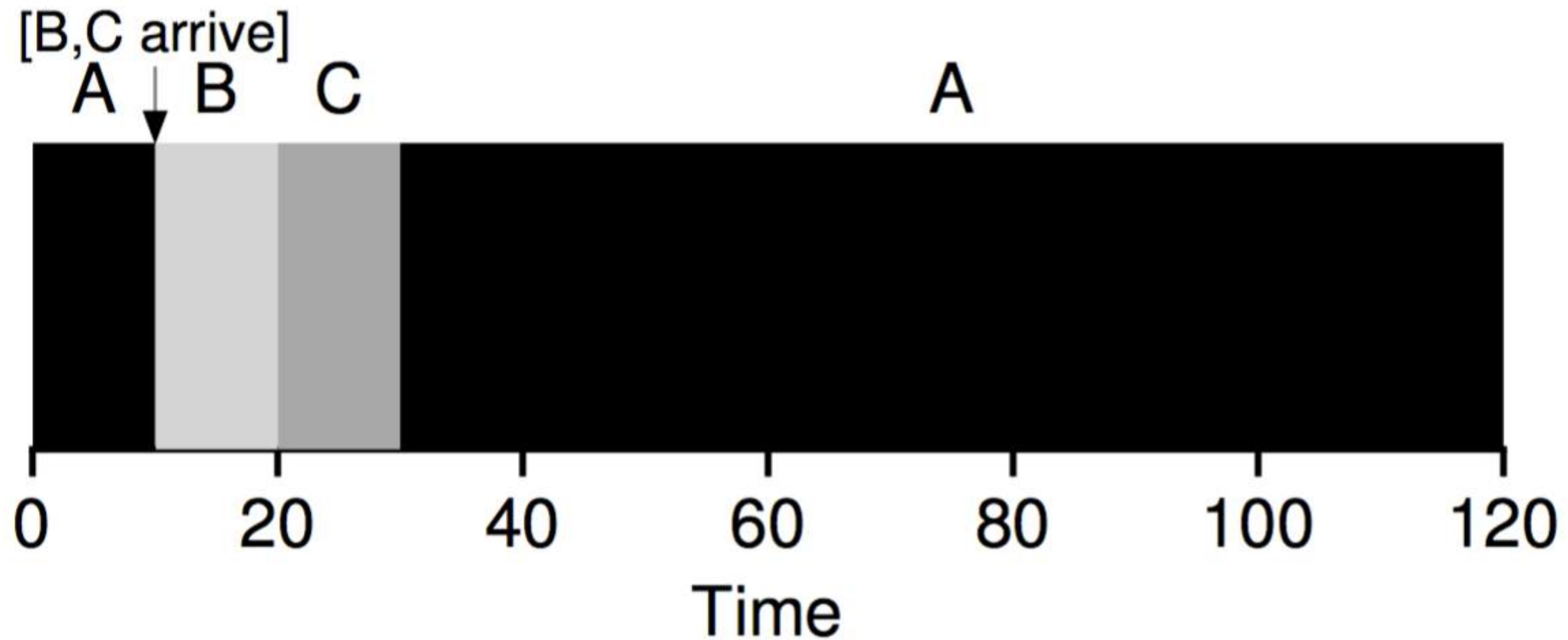
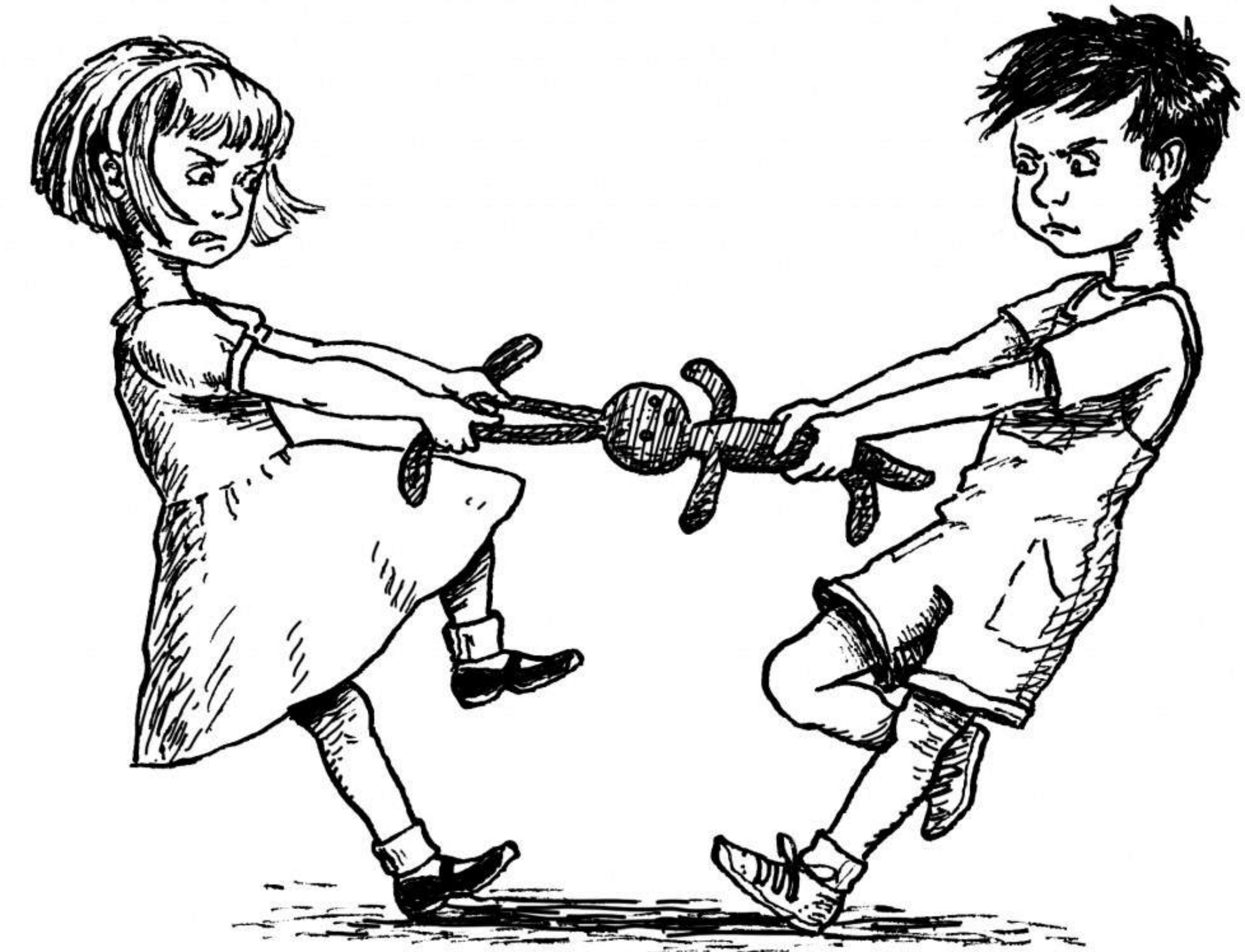


Figure 7.5: STCF Simple Example

Time sharing and interactive systems



Metric for Interactive systems:

$$T_{response} = T_{firstrun} - T_{arrival}$$

Interactive systems

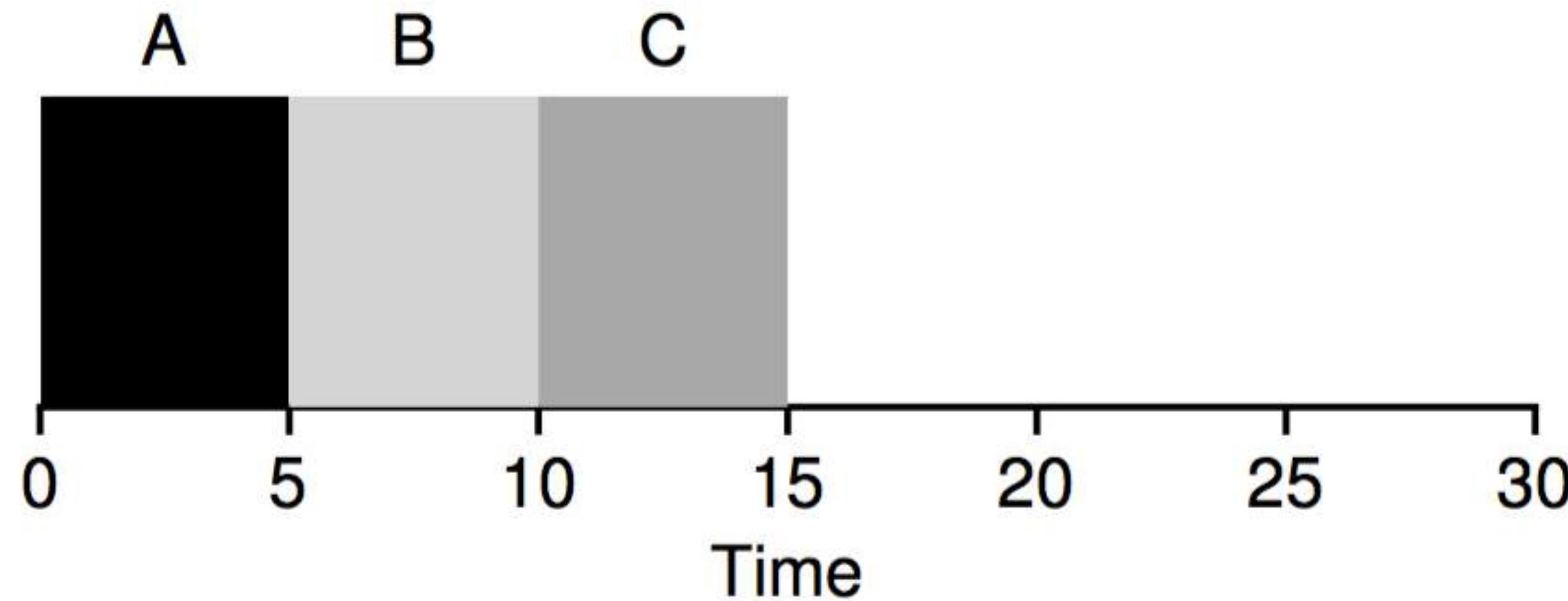


Figure 7.6: SJF Again (Bad for Response Time)

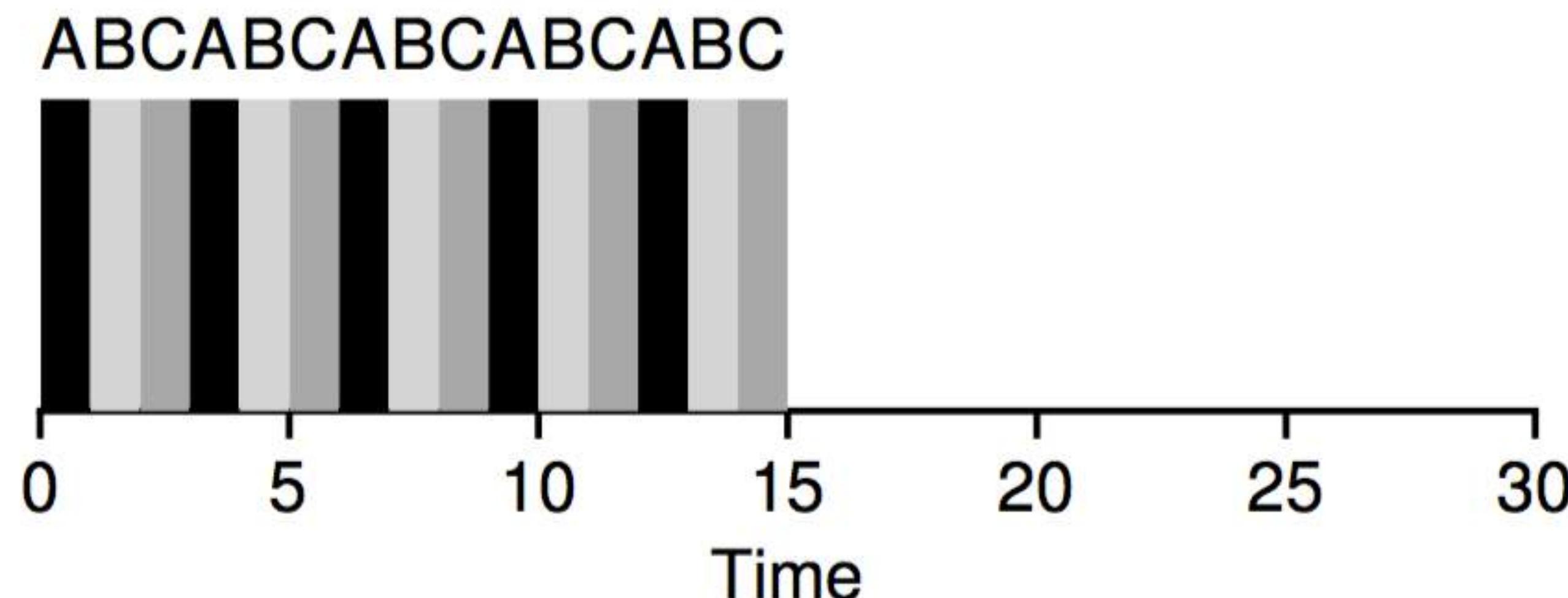


Figure 7.7: Round Robin (Good for Response Time)

Incorporating I/O

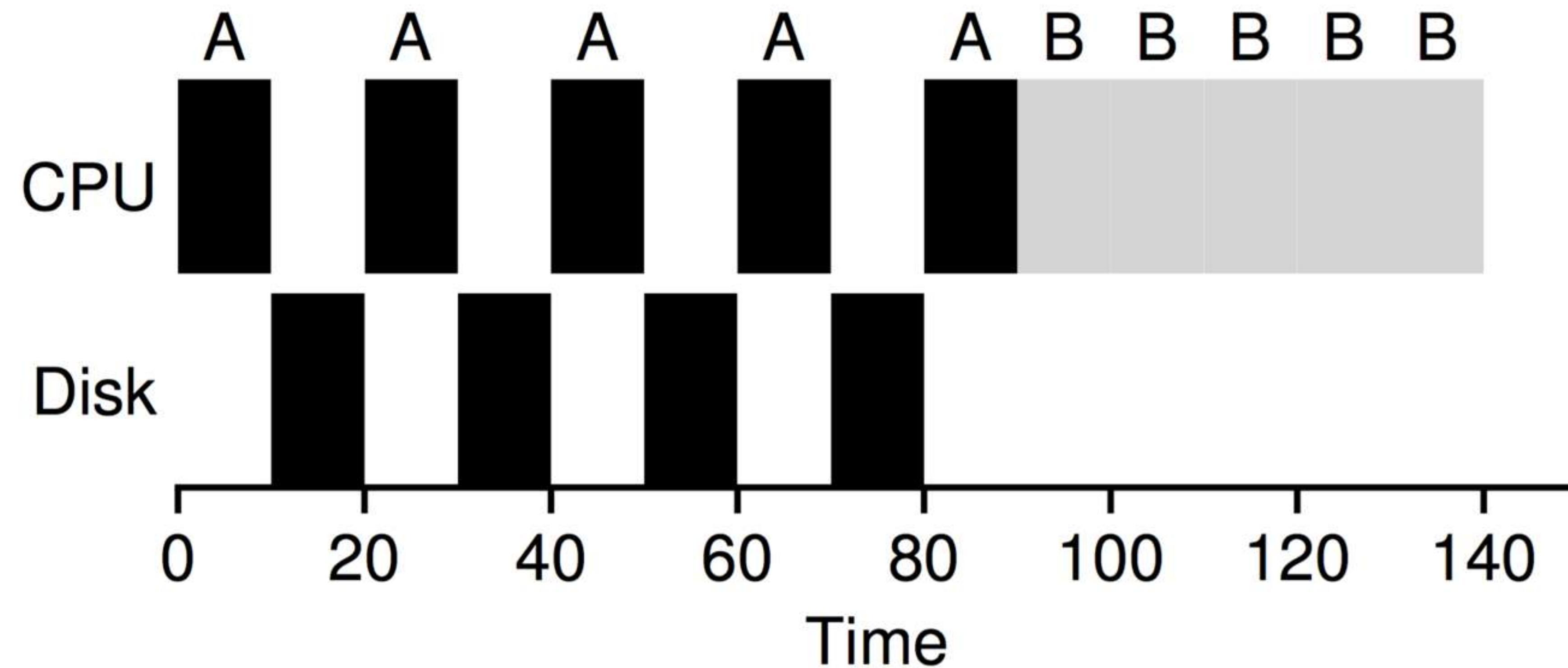


Figure 7.8: Poor Use of Resources

Overlap

TIP: OVERLAP ENABLES HIGHER UTILIZATION

When possible, **overlap** operations to maximize the utilization of systems. Overlap is useful in many different domains, including when performing disk I/O or sending messages to remote machines; in either case, starting the operation and then switching to other work is a good idea, and improves the overall utilization and efficiency of the system.

Initial (simplifying) assumptions



1. Each job runs for the same amount of time.
- ~~2. All jobs arrive at the same time.~~
- ~~3. Once started, each job runs to completion.~~
- ~~4. All jobs only use the CPU (i.e., they perform no I/O)~~
5. The run-time of each job is known.

Overlap

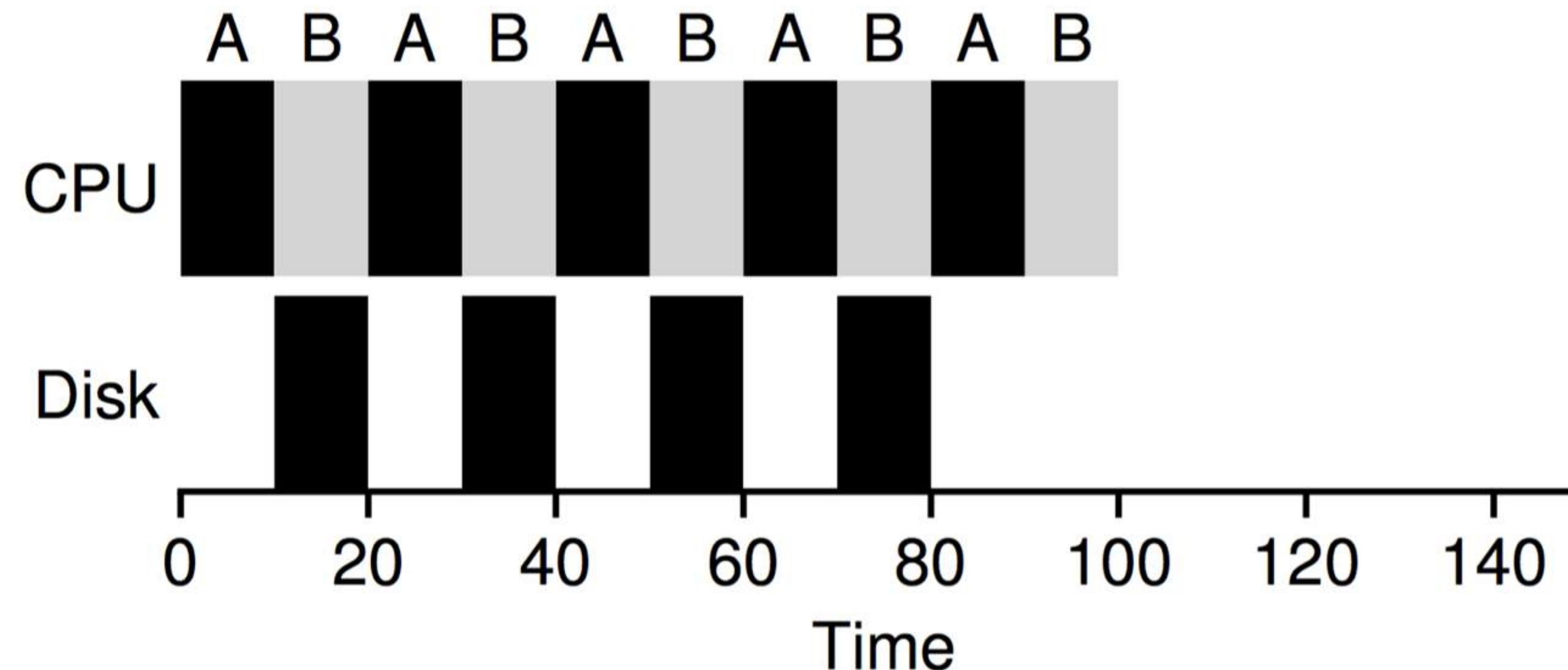
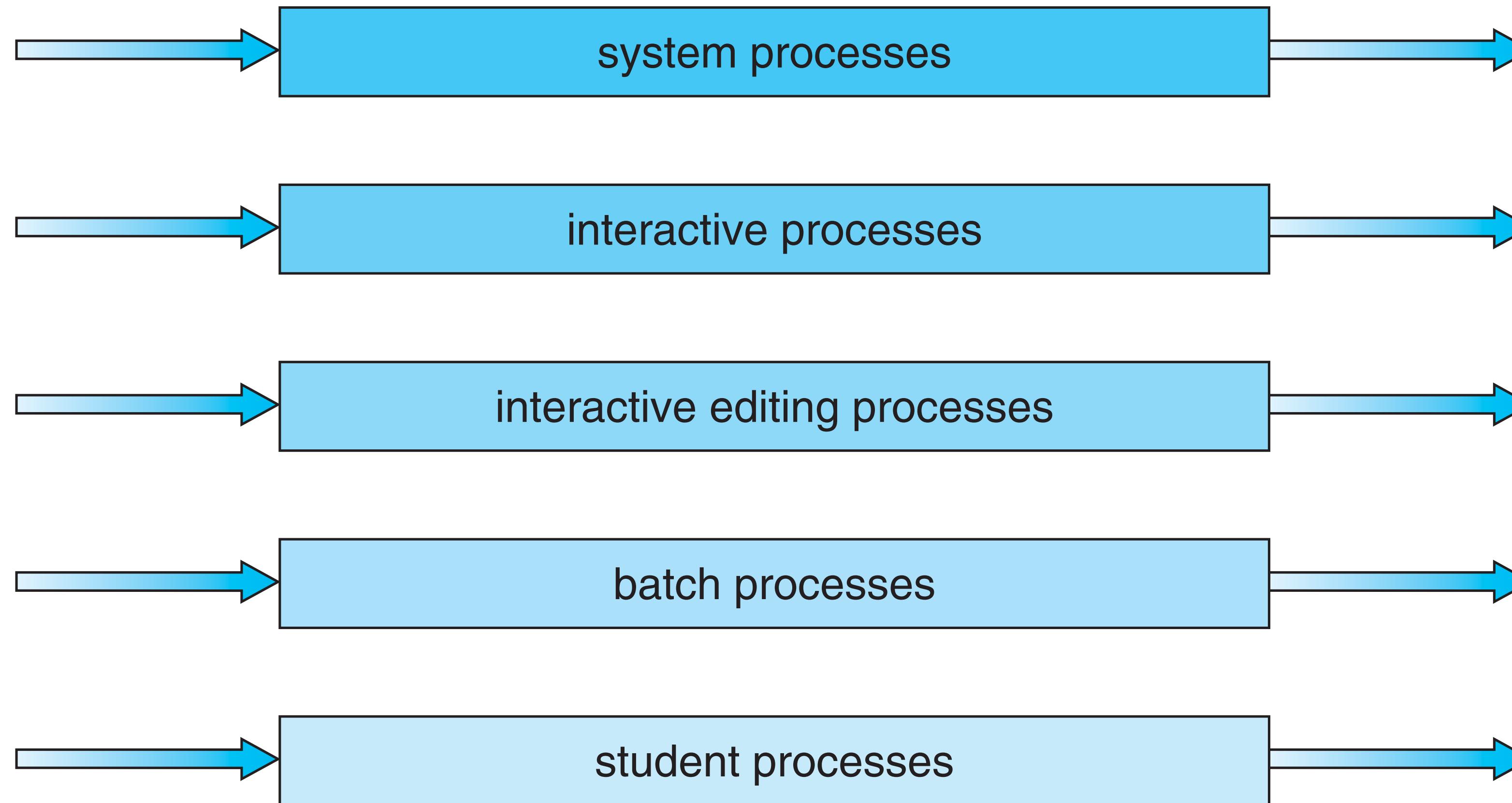


Figure 7.9: Overlap Allows Better Use of Resources

Multi-level queue scheduling

highest priority



lowest priority

Initial (simplifying) assumptions

- 1. Each job runs for the same amount of time.
- 2. All jobs arrive at the same time.
- 3. Once started, each job runs to completion.
- 4. All jobs only use the CPU (i.e., they perform no I/O)
- 5. The run-time of each job is known.

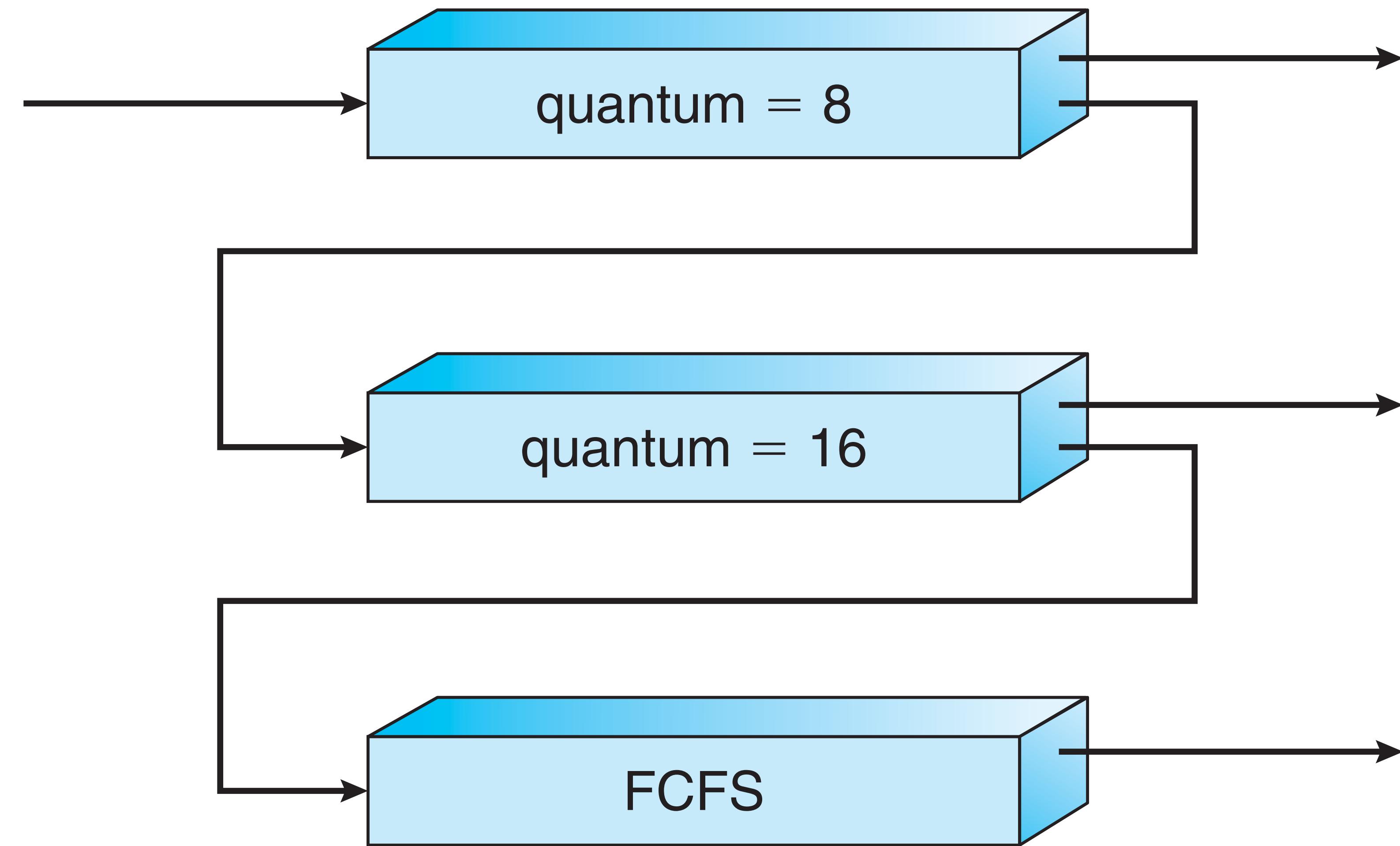


The OS can't see into future...



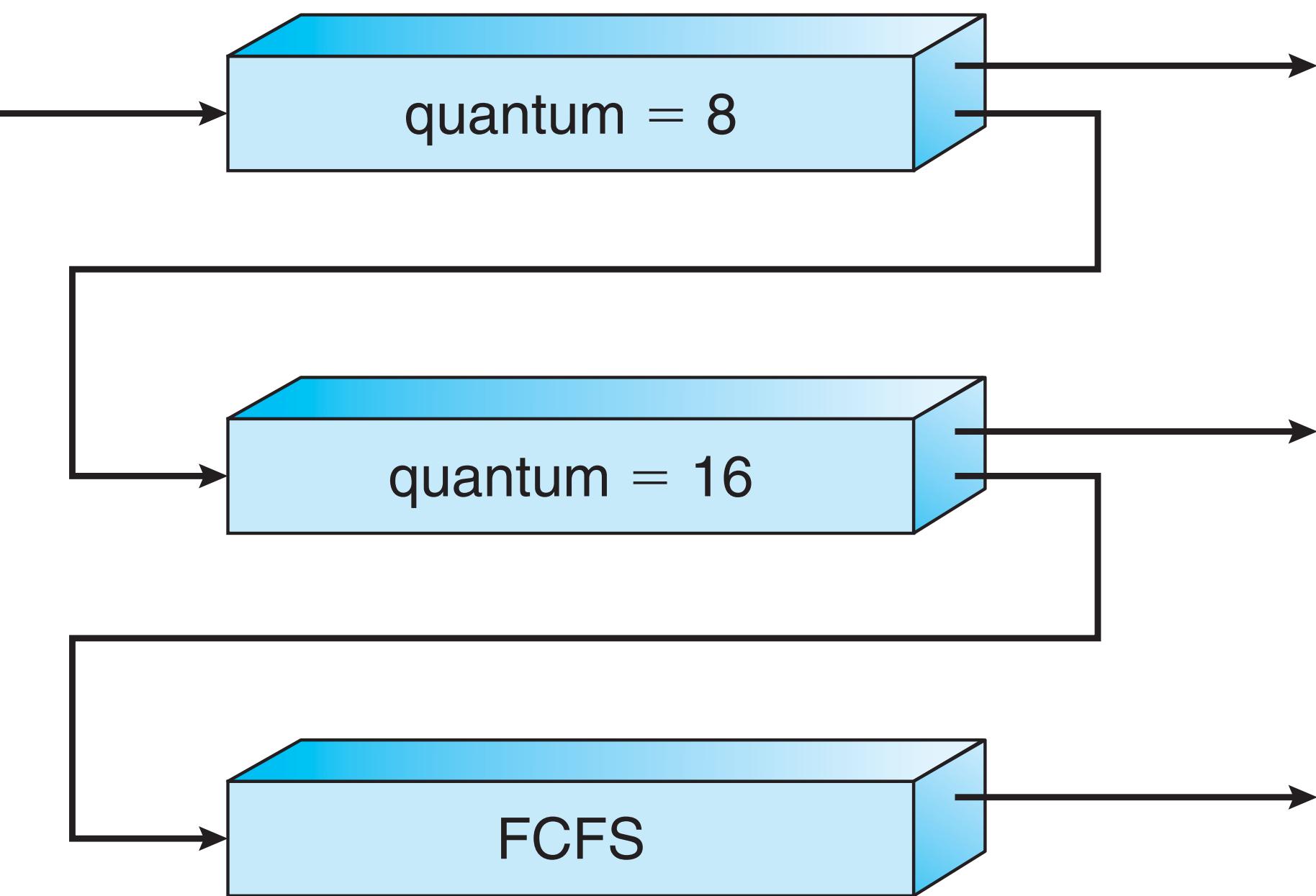
Multi-level feedback queue

Multi-level feedback queue scheduling



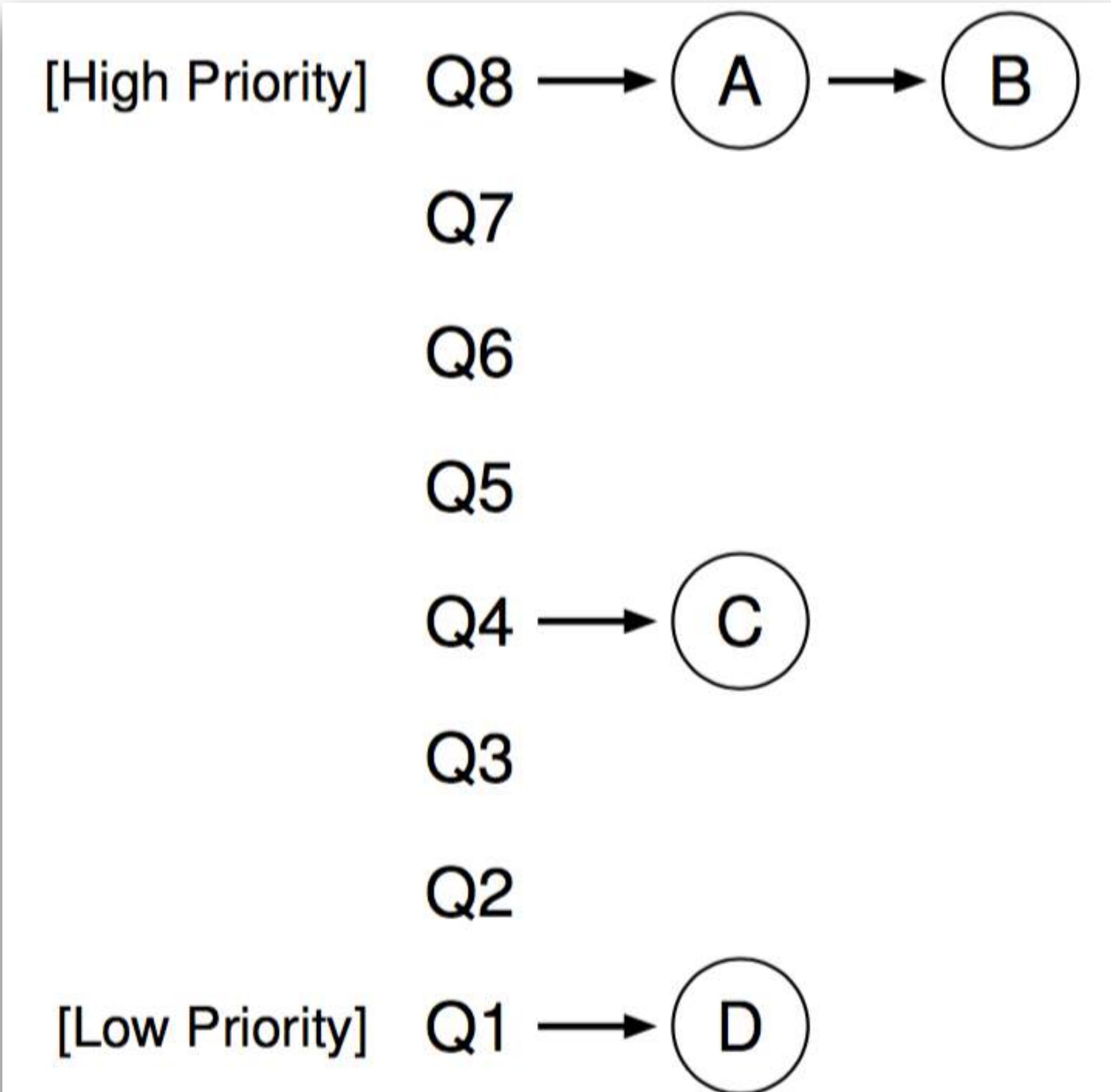
Multi-level feedback queue scheduling

- **Rule 1:** If $\text{Priority}(A) > \text{Priority}(B)$, A runs (B doesn't).
- **Rule 2:** If $\text{Priority}(A) = \text{Priority}(B)$, A & B run in RR.
- **Rule 3:** All jobs enter the system with the highest priority (the topmost queue).
- **Rule 4:** Once a job uses up its time allotment at a given level (regardless of how many times it has given up the CPU), its priority is reduced (i.e., it moves down one queue).
- **Rule 5:** After some time period S, move all the jobs in the system to the topmost queue.



Multi-level feedback queue scheduling

Example



Linux scheduling

Prior to Version 2.5:

- Linux used a variation of the traditional UNIX scheduling algorithm.
- Did not have good support for multiple processors.
- Had poor performance for systems with many runnable processes.

Linux scheduling

Version 2.5:

- Presented a new scheduling algorithm: $O(1)$
- $O(1)$ ran in constant time regardless the number of runnable processes.
- Provided support for SMP systems including load balancing and processor affinity.
- It worked great for SMP systems. But, it wasn't very good for interactive systems (e.g., Desktop systems) because of slow response times.

Linux scheduling

Version 2.6:

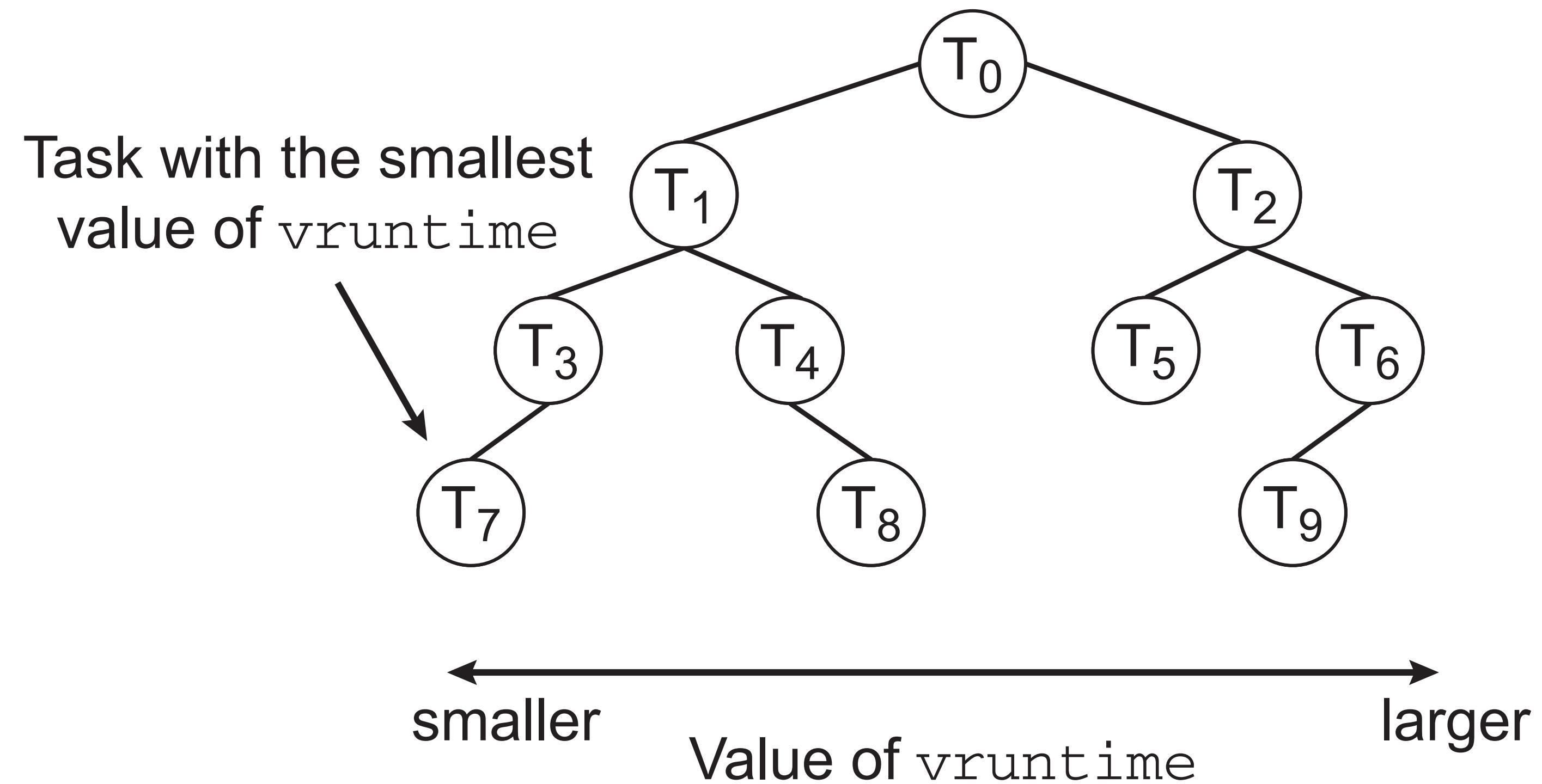
- Scheduler was revised again: *Completely Fair Scheduler* (CFS).
- CFS became the default linux scheduler.

Completely Fair Scheduler (CFS)

- Scheduling based on **scheduling classes**.
 - Each class has a priority.
 - Different classes allow for different scheduling algorithms depending on the system needs.
 - ▶ Example: Scheduling criteria for server systems can be different from criteria for mobile devices.

Completely Fair Scheduler (CFS): red-black tree

- A task is added to the tree when it becomes runnable.
 - A task is removed from the tree when it is not runnable.
 - Tasks that are given less processing time are on the left. Tasks that are given more time are on the right.



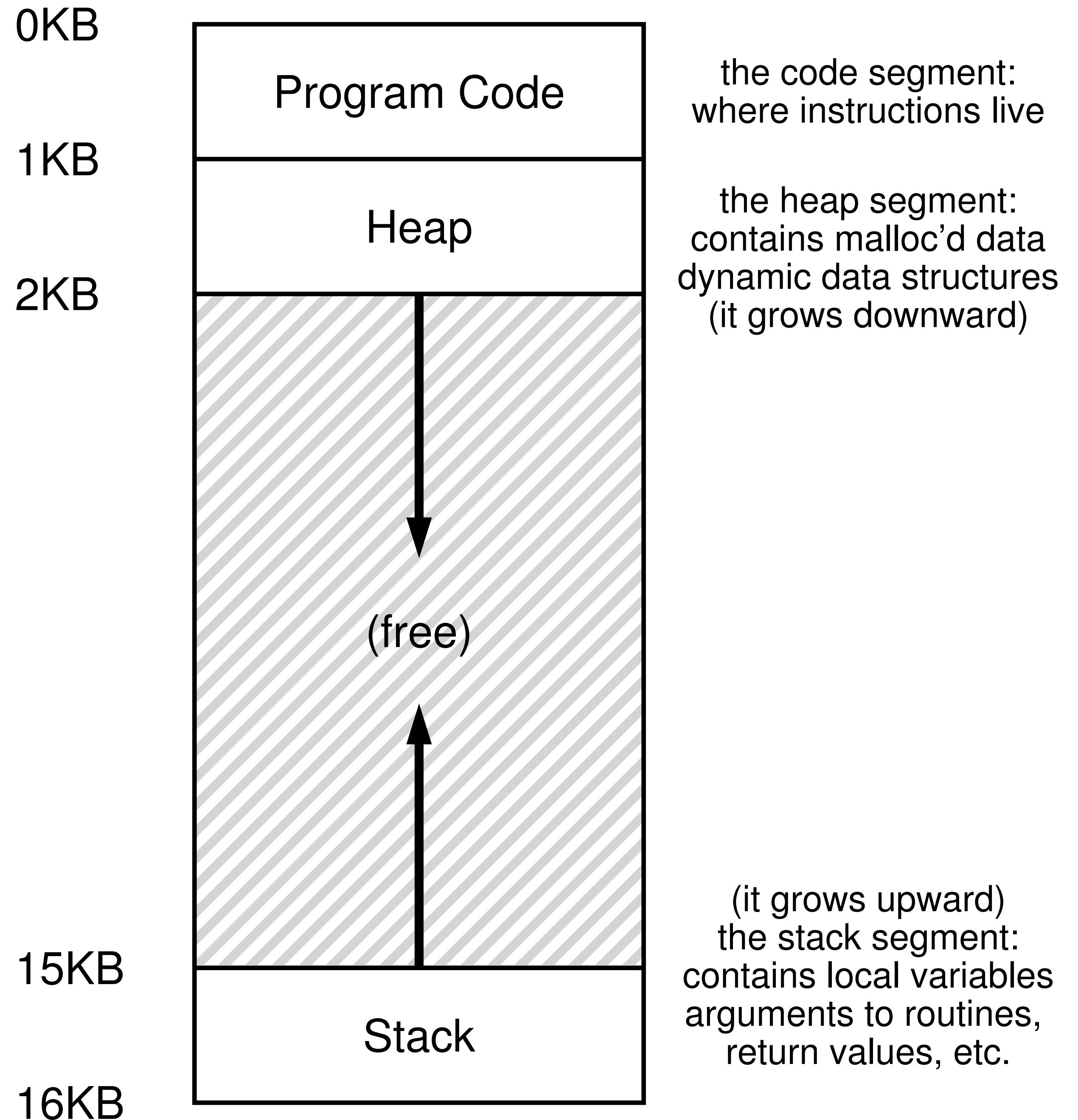
Paging

CSE 4001

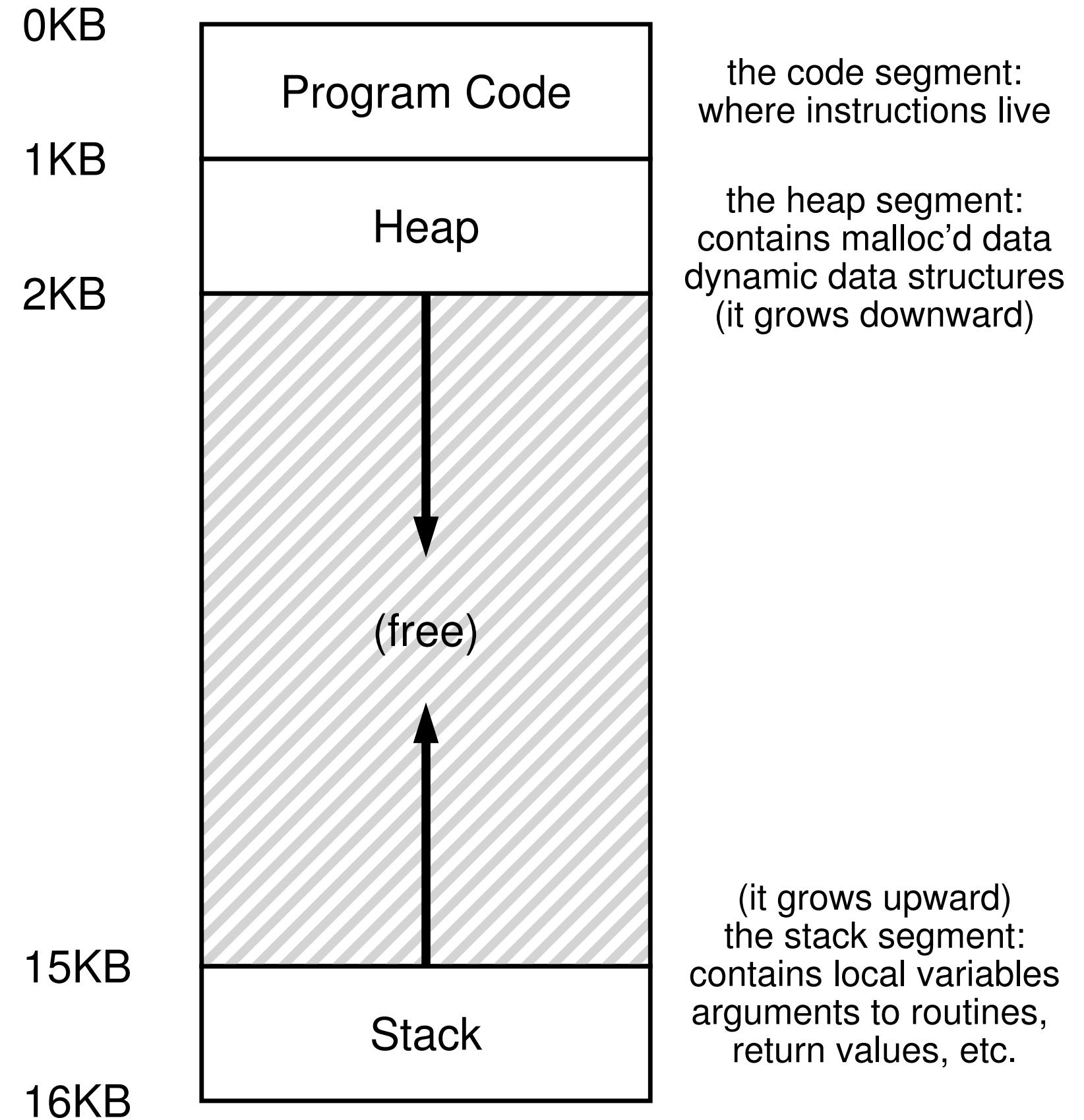
Content

- Virtual address space
- Basic paging mechanism
- Limitations
- Protection
- Shared pages

Example address space



Types of memory



Stack: Short-lived memory. Allocations and deallocations are managed *implicitly* (e.g., by the compiler), not by the programmer.

Heap: Long-lived memory. Allocations and deallocations are *explicitly* handled by the programmer.

Examples

```
void func() {  
    int x;  
    . . .  
}
```

Examples

```
void func() {  
    int *x = (int *) malloc(sizeof(int));  
    . . .  
}
```

Every address you see is virtual

Here's a little program that prints out the locations of the main() routine (where code lives), the value of a heap-allocated value returned from malloc(), and the location of an integer on the stack:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(int argc, char *argv[]) {
4     printf("location of code : %p\n", (void *) main);
5     printf("location of heap : %p\n", (void *) malloc(1));
6     int x = 3;
7     printf("location of stack : %p\n", (void *) &x);
8     return x;
9 }
```

When run on a 64-bit Mac OS X machine, we get the following output:

```
location of code : 0x1095afe50
location of heap : 0x1096008c0
location of stack : 0x7fff691aea64
```

Paging

Basic problem with allocating contiguous blocks of memory for processes

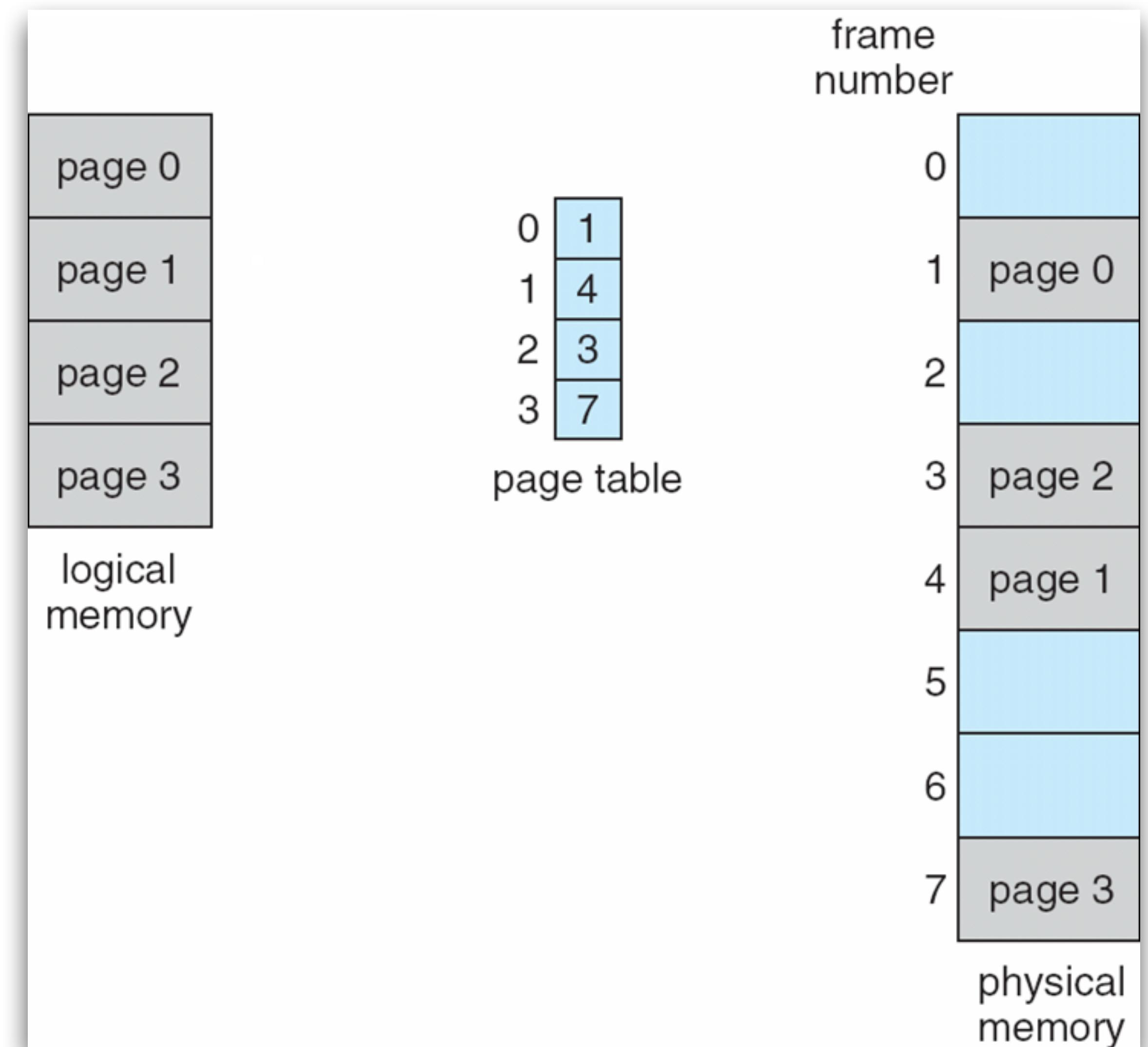
- Determining the size of memory blocks is difficult because different processes have different memory requirements.

Paging: physical address space is allowed to be non-contiguous

Paging

Basic paging method

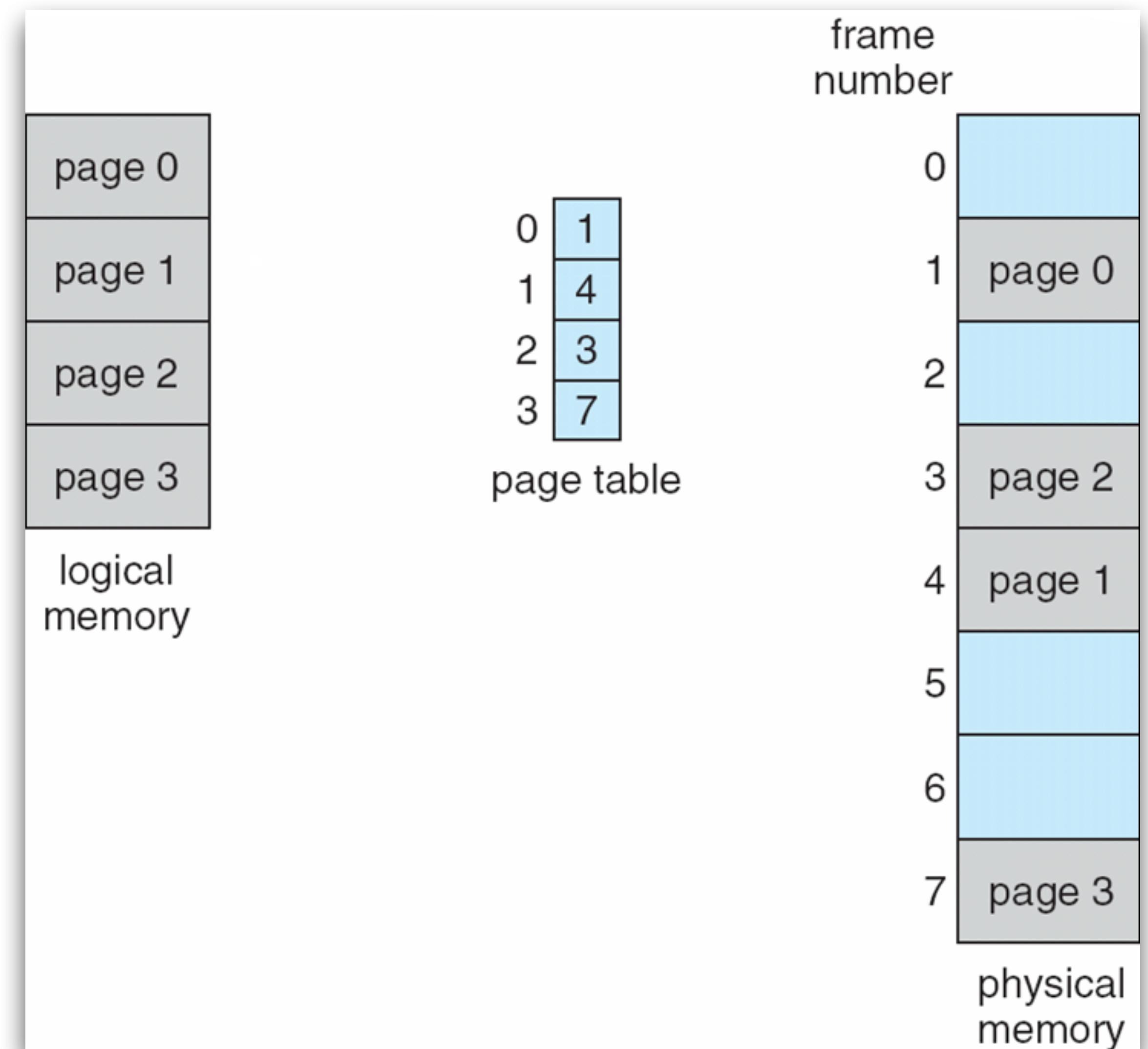
- Divide physical memory into fixed-sized blocks called **frames** (size is power of 2, between 512 bytes and 16 MB).
- Divide logical memory into blocks of same size called **pages**.



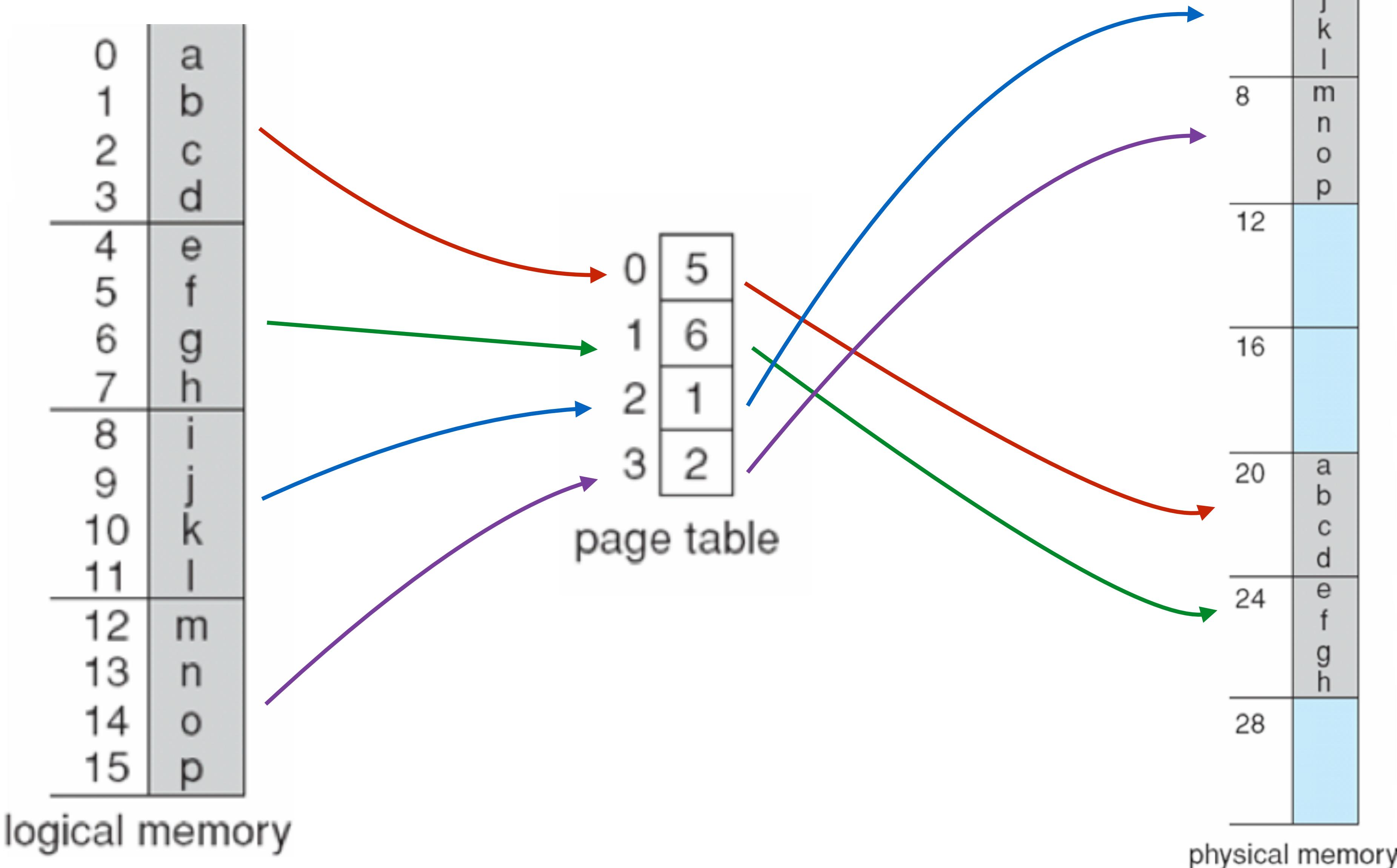
Paging

Basic paging method

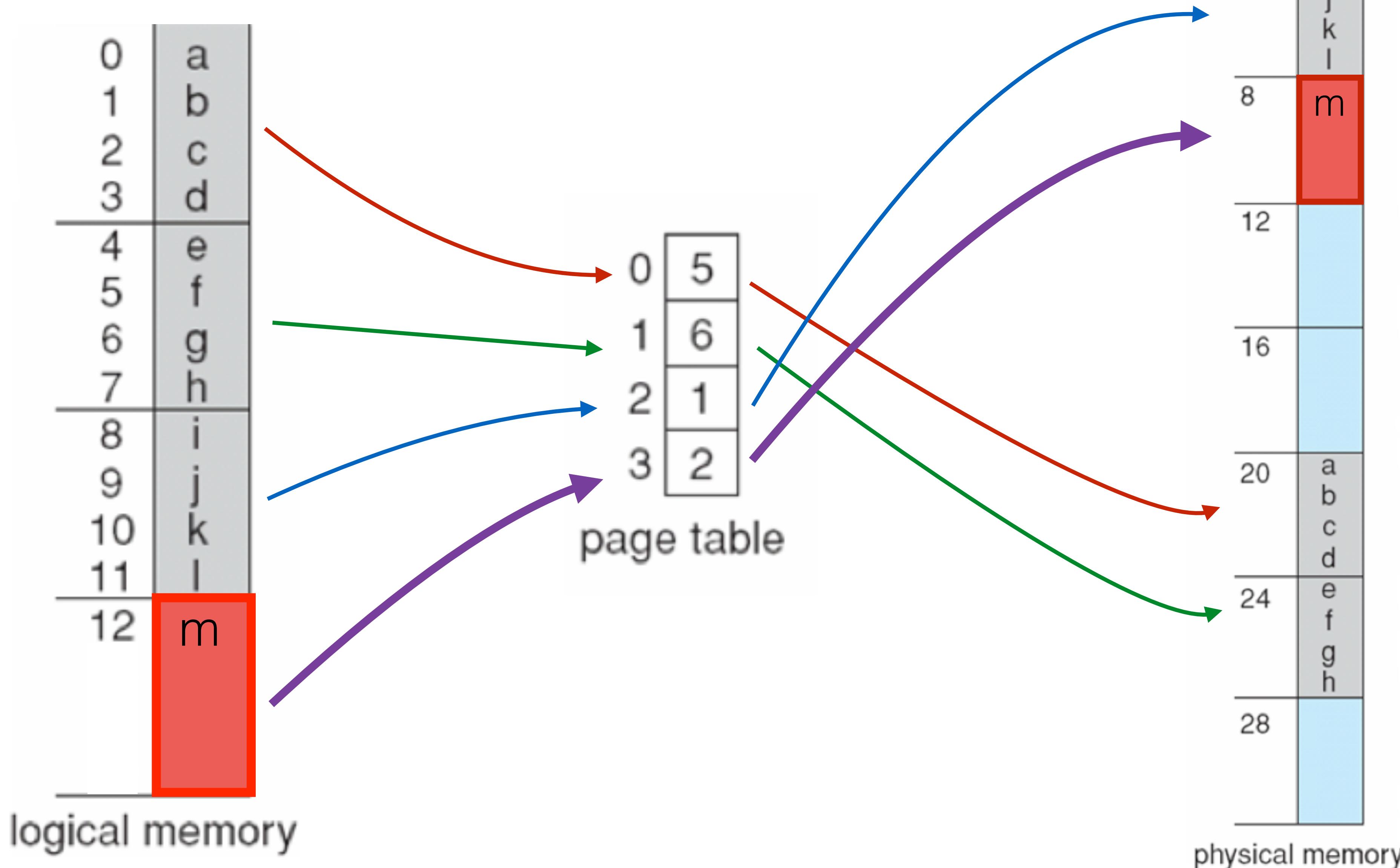
- Any page can be assigned to any free page frame
- External fragmentation is eliminated
- Internal fragmentation is at most a part of one page per process



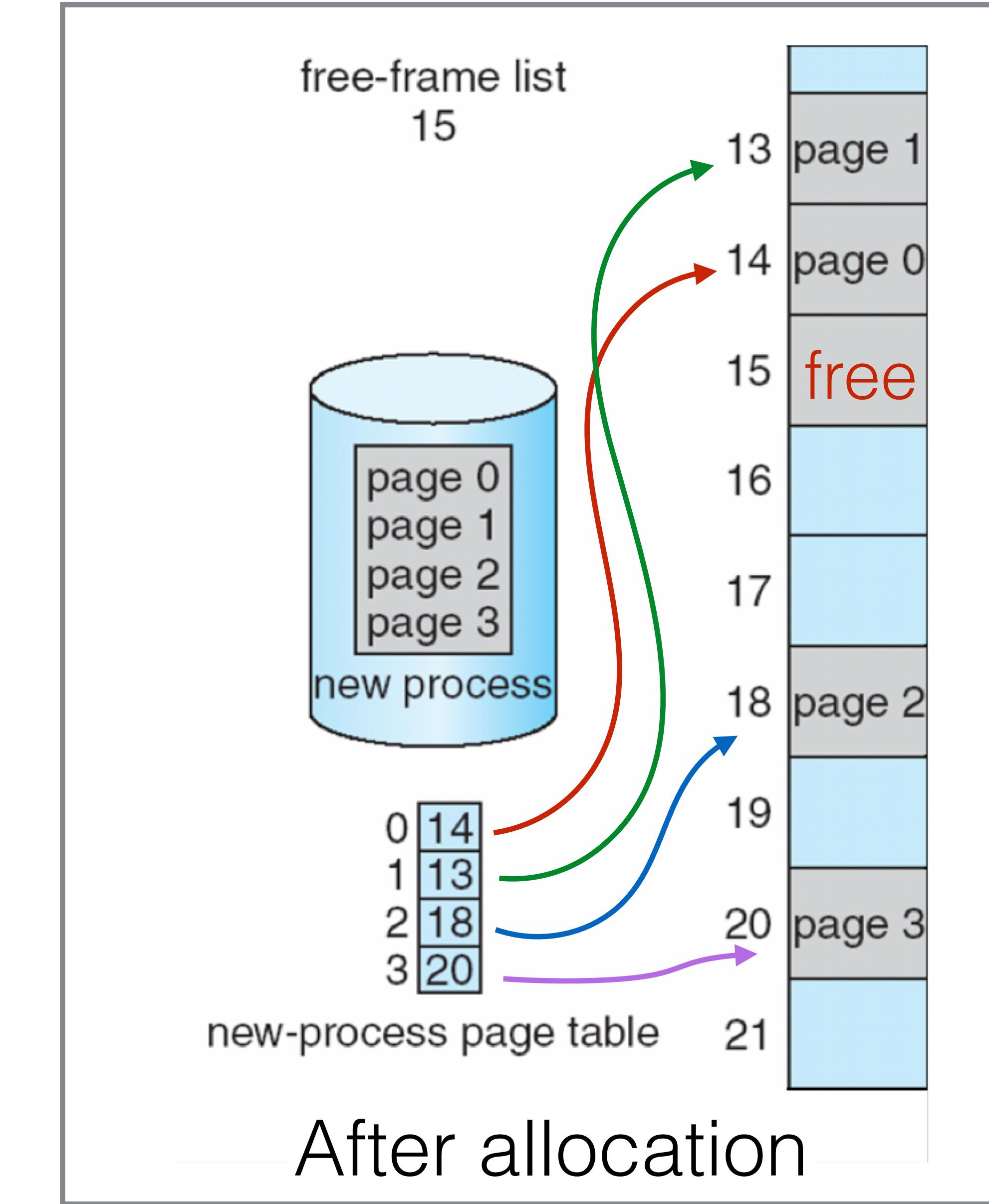
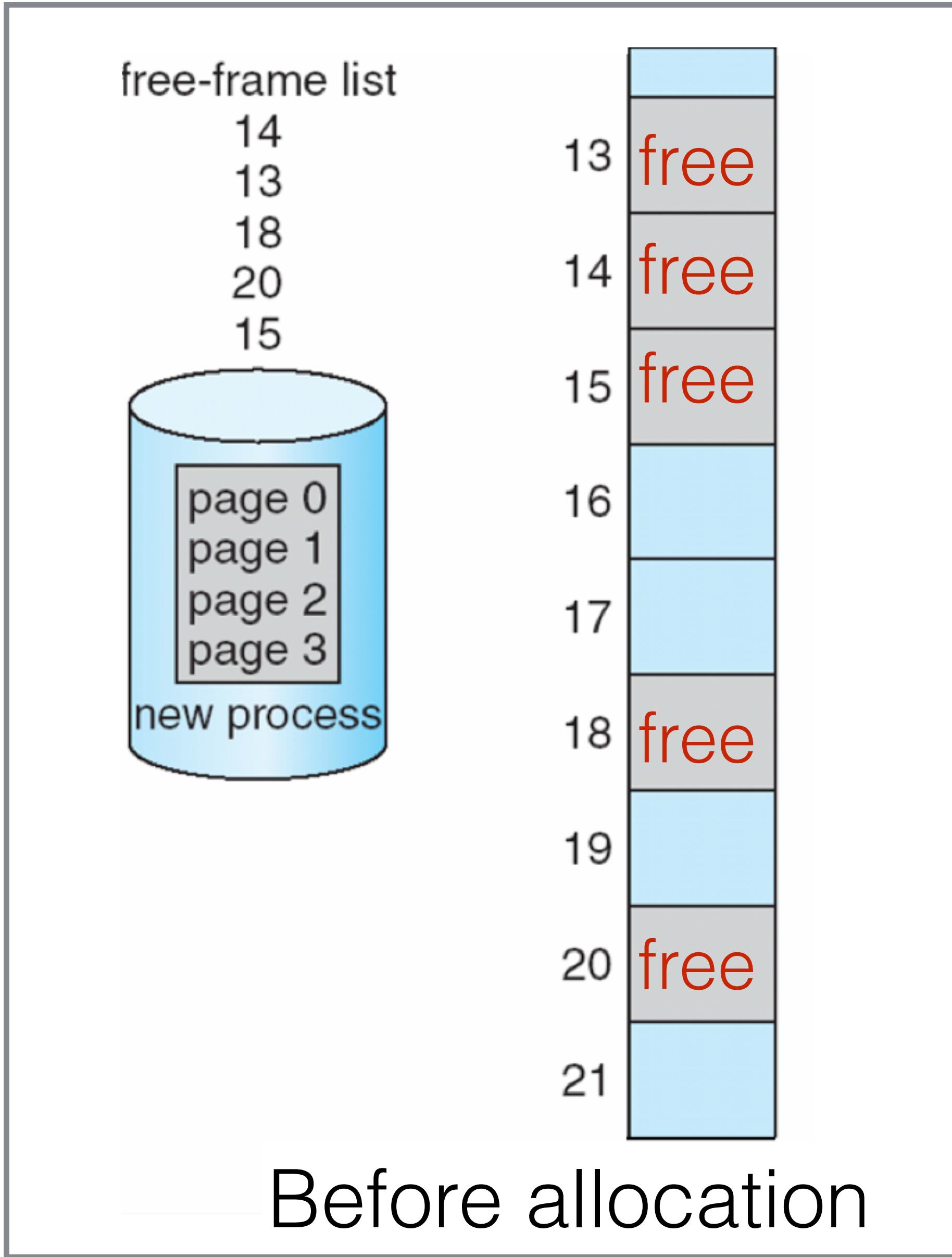
Paging example: 32-byte memory and 4-byte pages



Paging example: 32-byte memory and 4-byte pages (with internal fragmentation)



New process is executed: free frames before and after allocation



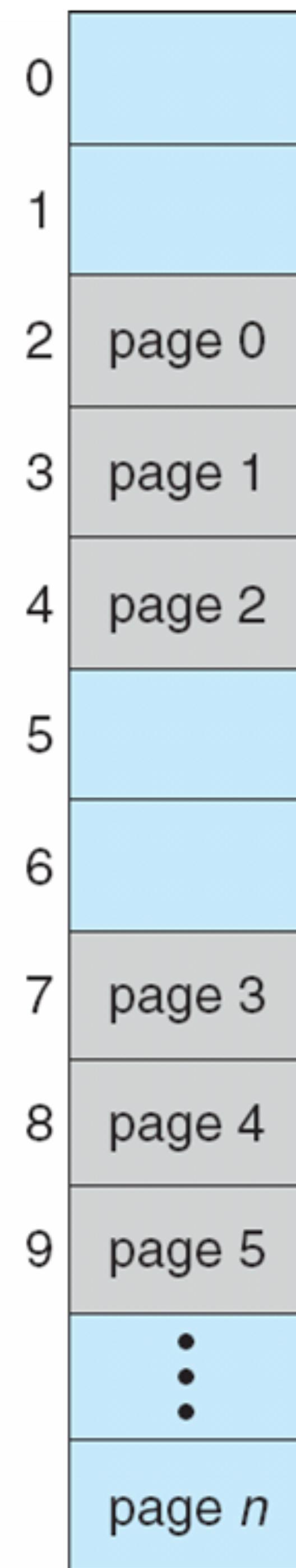
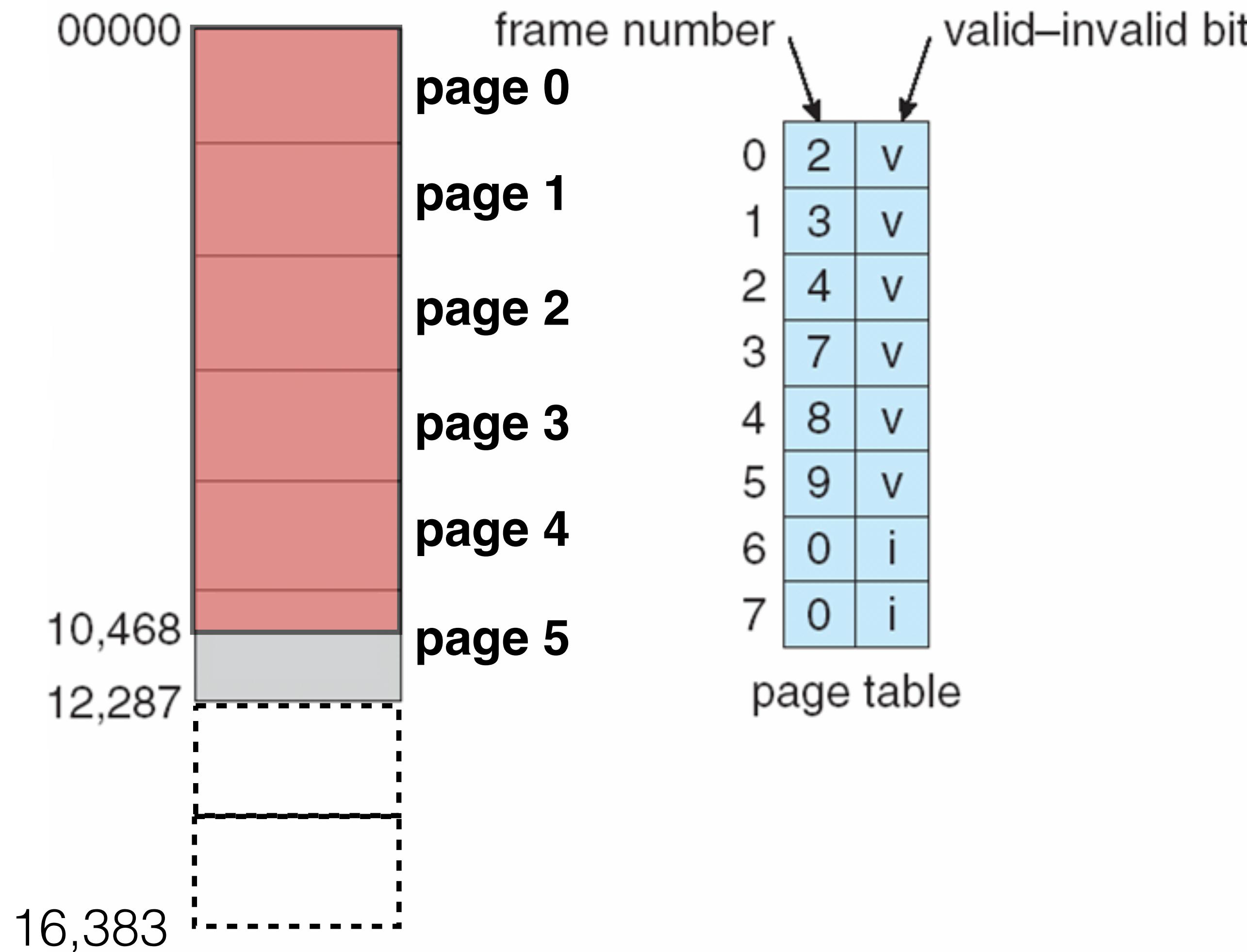
Paging Limitations - Space

- Page table might need a lot of space
- Registers can be used to store page tables but they are only feasible for small tables (e.g., 256 entries).
- Modern computers have page tables of 1 million entries.
- Such large page tables are kept in main memory and a page-table base register (PTBR) points to the table.

Protection

- Memory protection: each frame has a **protection bit**.
- **Valid-invalid** bit for each entry in the page table:
 - “valid” indicates that the associated page is in the process’ logical address space, and is thus a legal page.
 - “invalid” indicates that the page is not in the process’ logical address space.

Protection



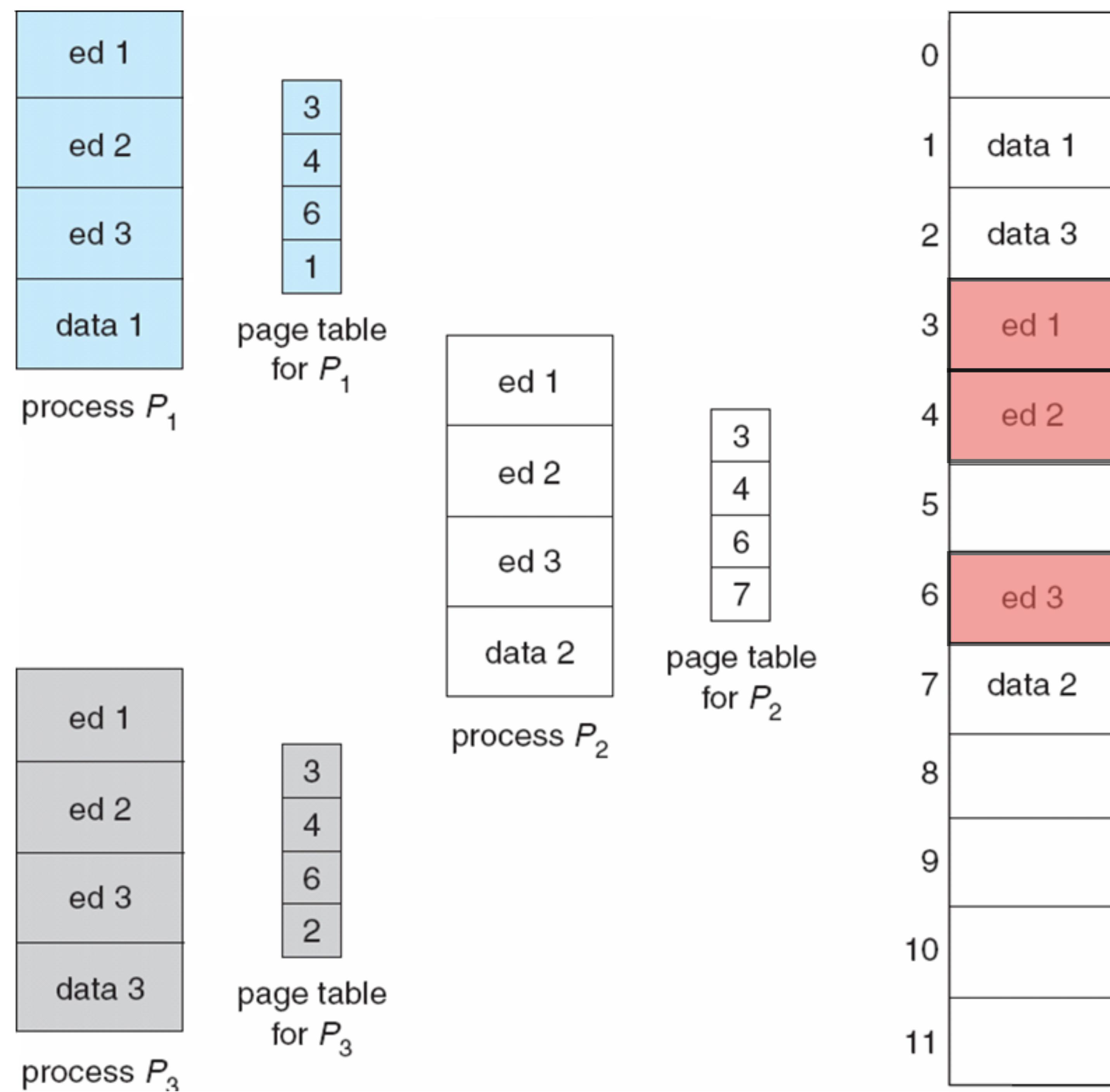
A few more useful aspects of paging

- Shared pages
- Copy-on-write
- Memory-mapped files

Shared Pages

- Paging allows for the possibility of sharing common code.
- Sharing pages is useful in time-sharing environments (e.g., 40 users, each executing a text editor).
- OS can implement shared-memory (IPC) using shared pages.

Example of shared Pages



Copy-on-Write (COW), e.g. on `fork()`

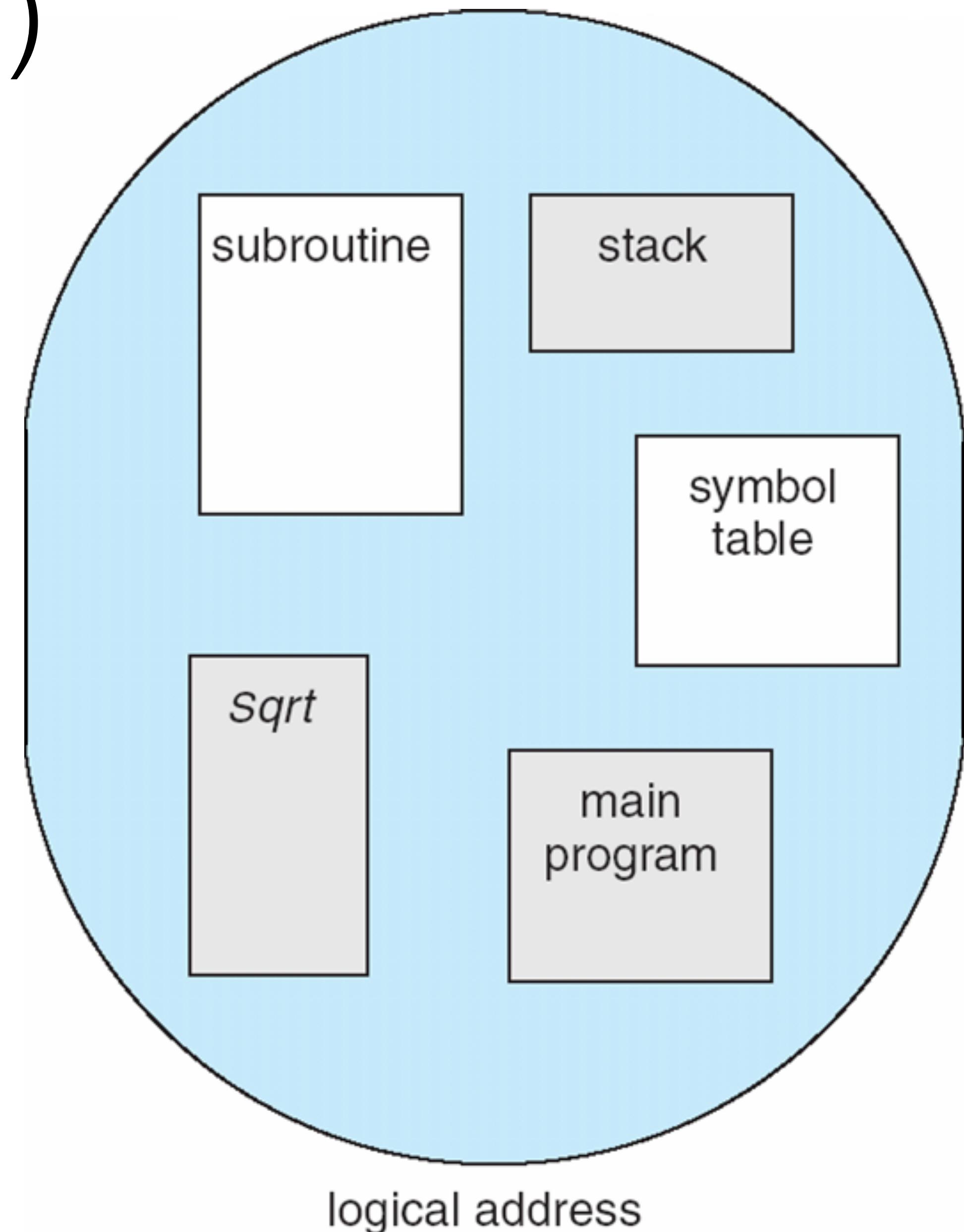
- copy-on-write (COW), e.g., on `fork()`
 - Instead of copying all pages, create shared mappings of parent pages in child address space
 - A. Make shared mappings read-only in child space
 - B. When child does a write, a protection fault occurs, OS takes over and can then copy the page and resume child.

Segmentation

- Memory-management scheme that supports the user's view of memory.
- View memory as a collection of variable-sized segments, with no necessary ordering among segments.

Segmentation (a program)

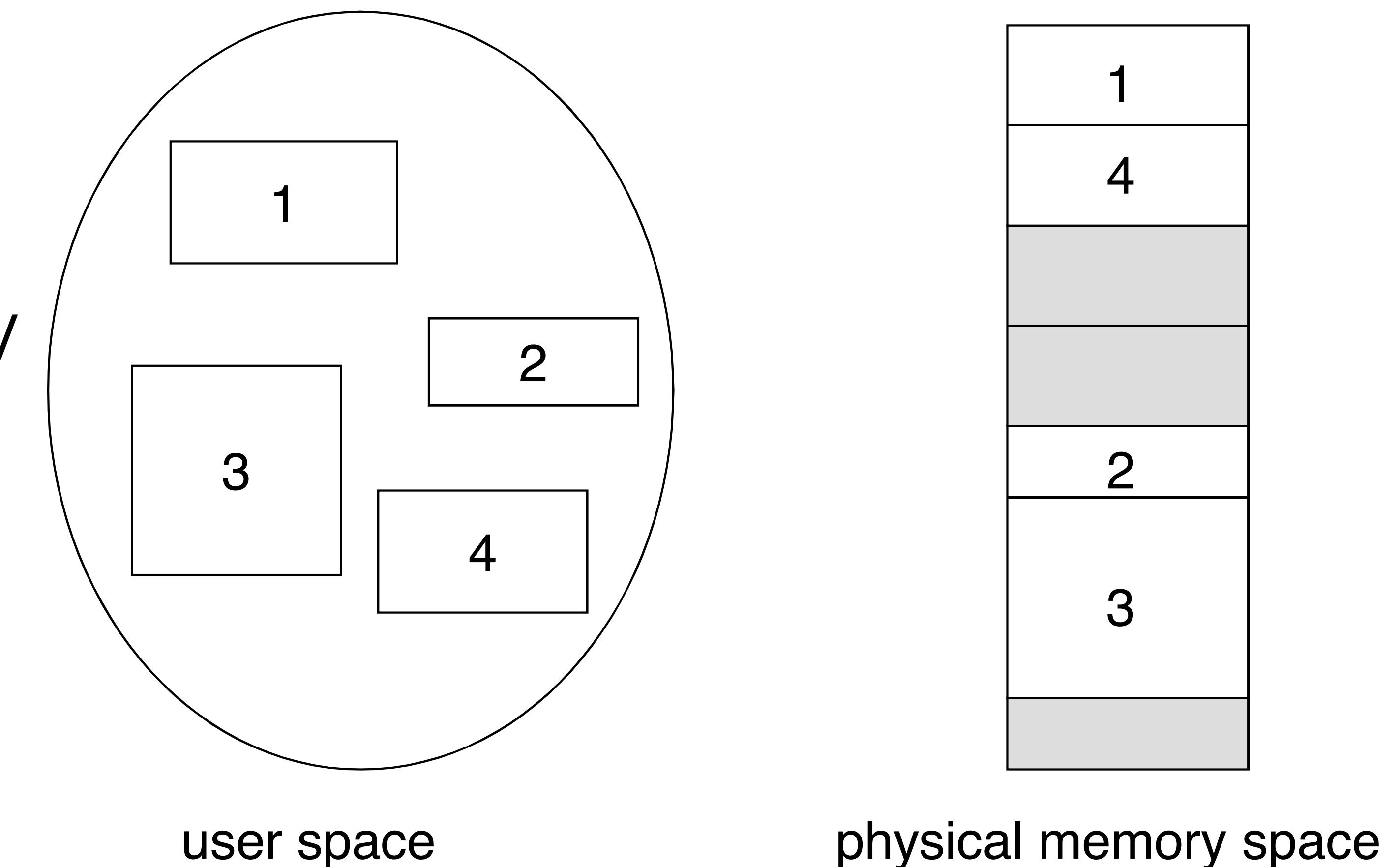
- We think of a program as a main program, a stack, a math library, etc.
- Each module is referred to by name
- In this view of a program, we might not care whether the stack is stored before or after the `sqrt()` function.



Logical view of segmentation

- For simplicity of implementation, each segment is addressed by a **segment number** and an **offset**:

<segment-number, offset>



Segmentation Hardware

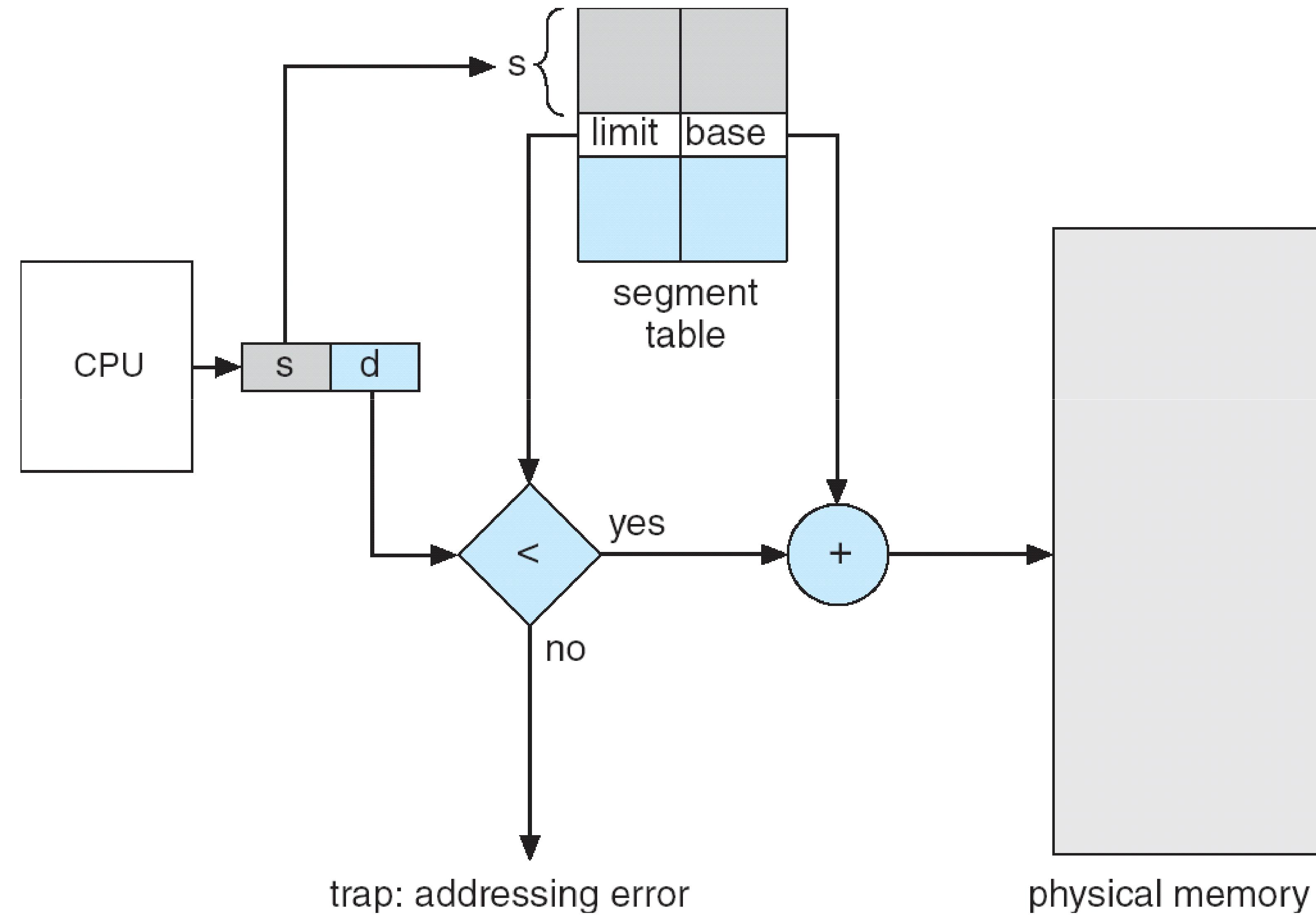
Segment tables:

Base: starting address of the segment in physical memory.

Limit: length of the segment.

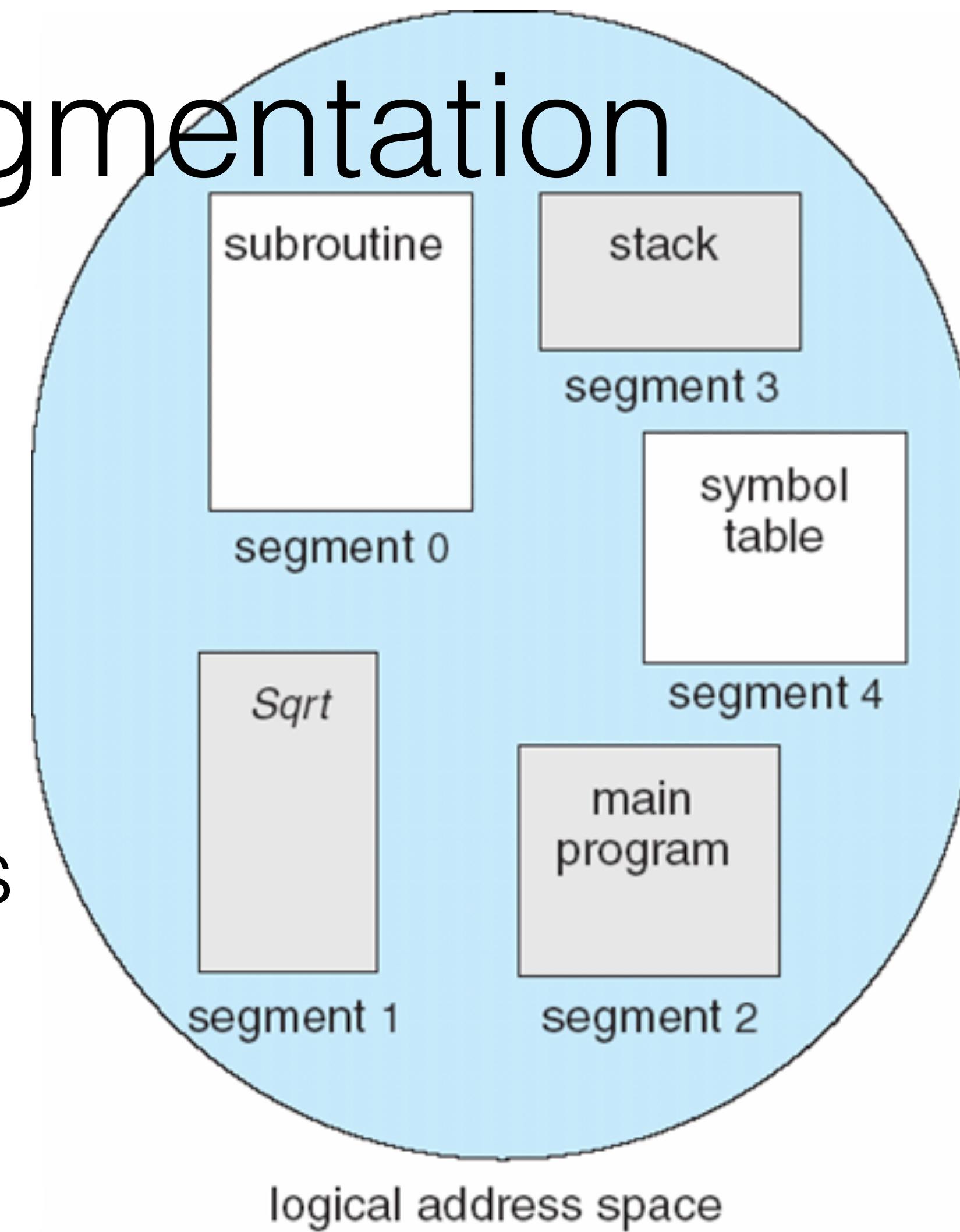
Additional metadata includes protection bits.

<segment-number, offset>



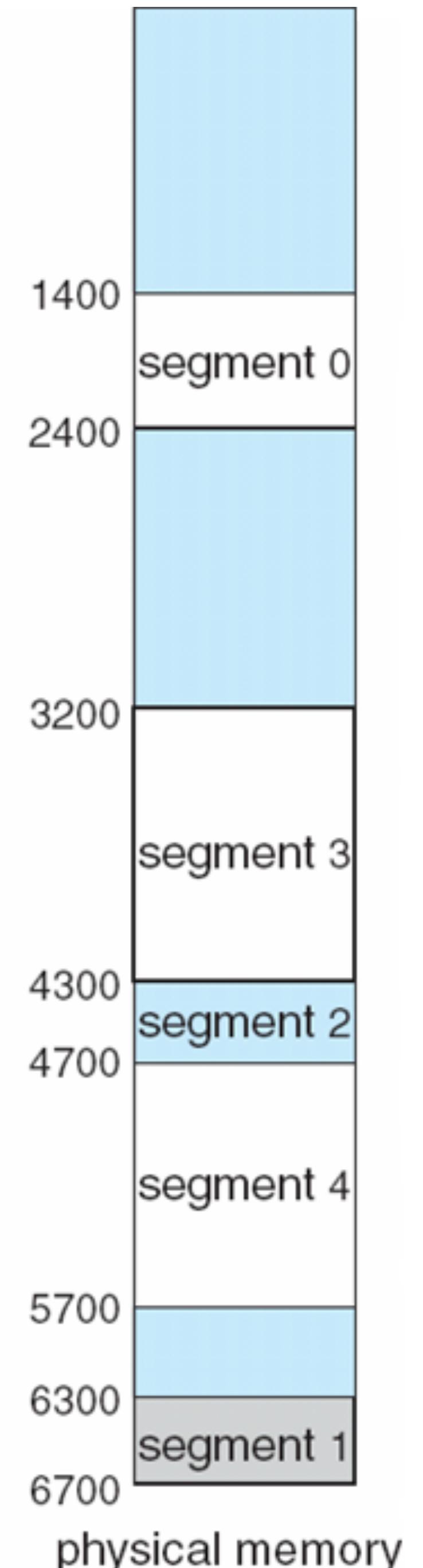
Example of segmentation

- Logical memory divided into 5 segments.
- Segment 2 is 400 bytes long and begins at location 4,300.
- **Question:** What happens if there is a reference to byte 1,222 of segment 0?



| | limit | base |
|---|-------|------|
| 0 | 1000 | 1400 |
| 1 | 400 | 6300 |
| 2 | 400 | 4300 |
| 3 | 1100 | 3200 |
| 4 | 1000 | 4700 |

segment table



Some questions

How do paging and segmentation compare with respect to the following issues?

- External fragmentation
- Internal fragmentation
- Ability to share code across processes

Some questions

Assuming a 1-KB page size, what are the page numbers and offset for the following address:

- A. 2375
- B. 256

Some questions

| Segment | Base | Length |
|---------|------|--------|
| 0 | 219 | 600 |
| 1 | 2300 | 14 |
| 2 | 90 | 100 |
| 3 | 1327 | 580 |
| 4 | 1952 | 96 |

What are the physical addresses for the following logical addresses?

- a. 0,430
- b. 2,500

Virtual Memory

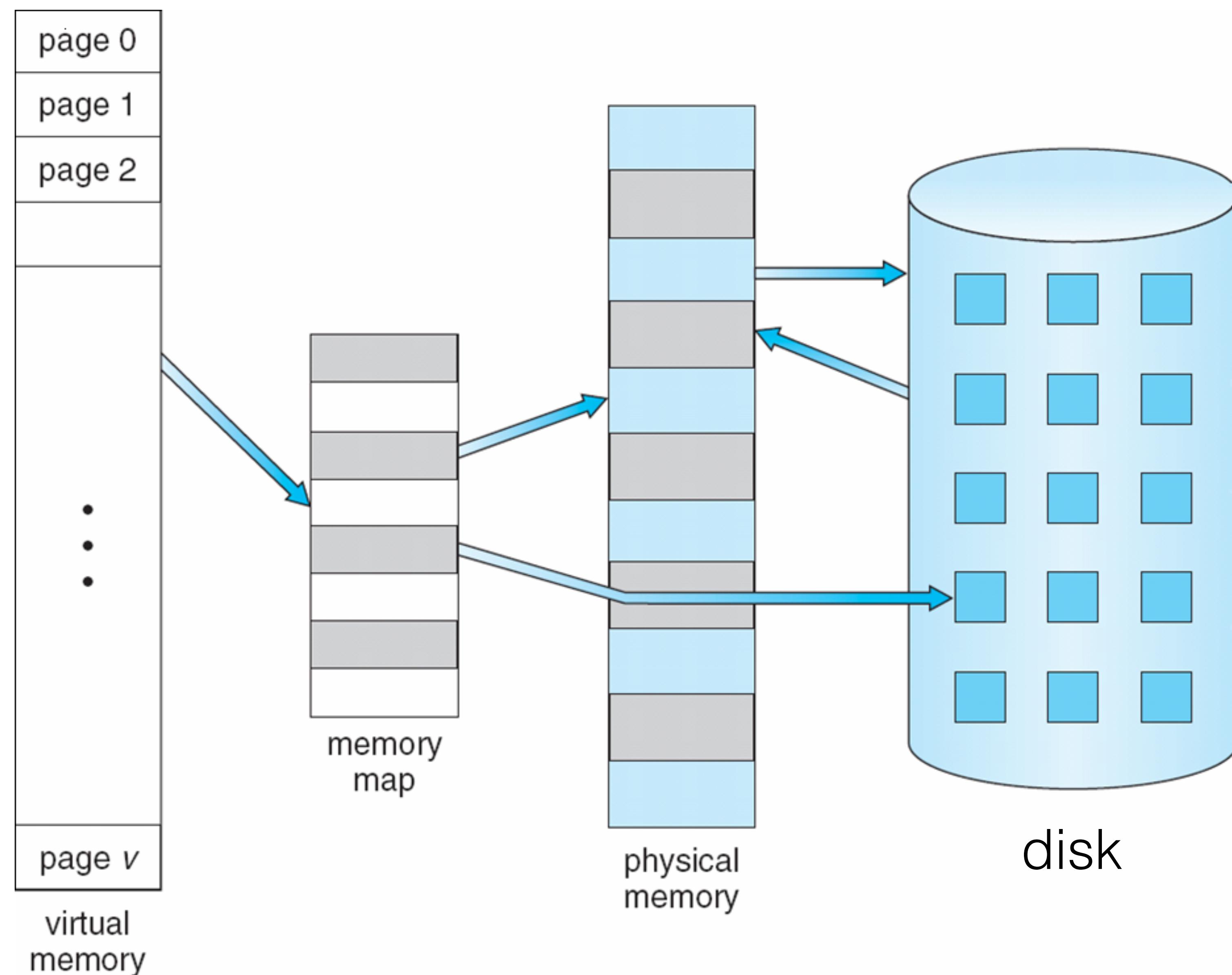
CSE 4001

Content

- Demand paging

Virtual Memory

- Separation of user logical memory from physical memory.
- Programs can be partially in memory for execution
- Logical address space can be much larger than physical address space



Implementation

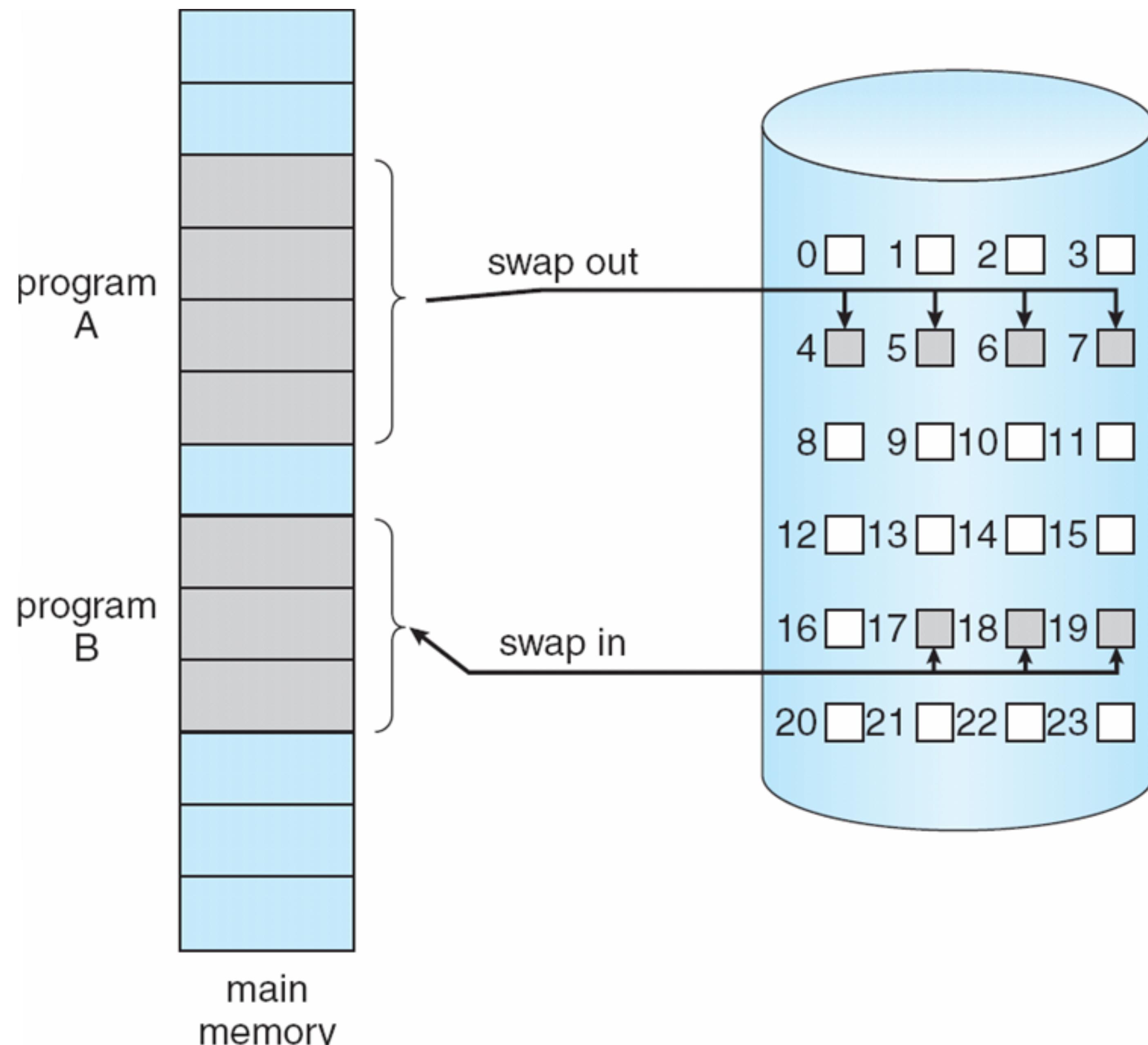
Virtual memory can be implemented via:

- Demand paging
- Demand segmentation

Demand paging

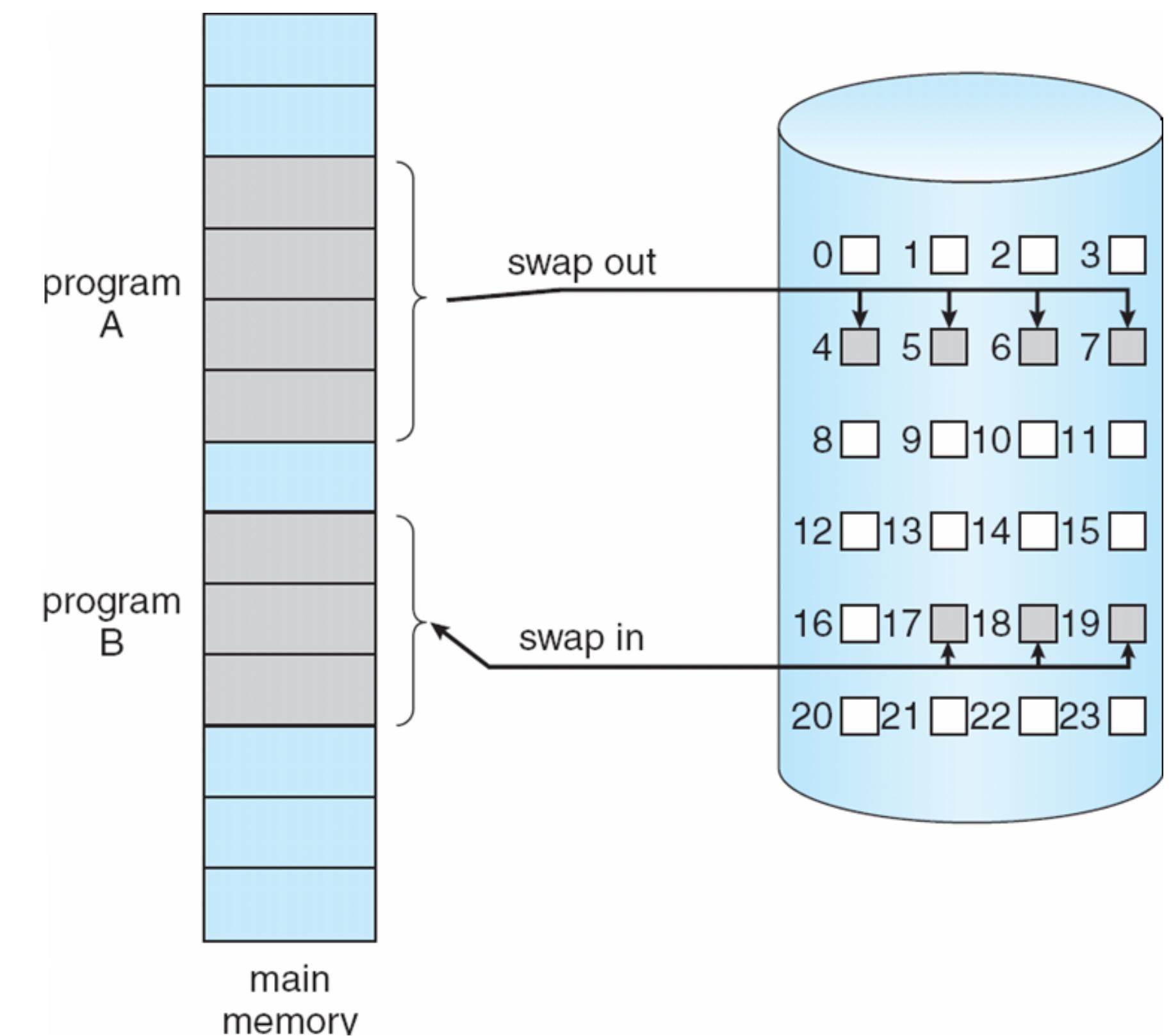
Bring a page into memory only when it is needed:

- Less I/O needed
- Less memory needs
- Faster response
- More users



Demand paging

- Demand paging is similar to a paging system with swapping, where processes reside in secondary memory (e.g., disk).
- **Lazy swapper**: only bring pages when they are needed.
- In the context of demand paging, we use the term **pager** instead of **swapper**.

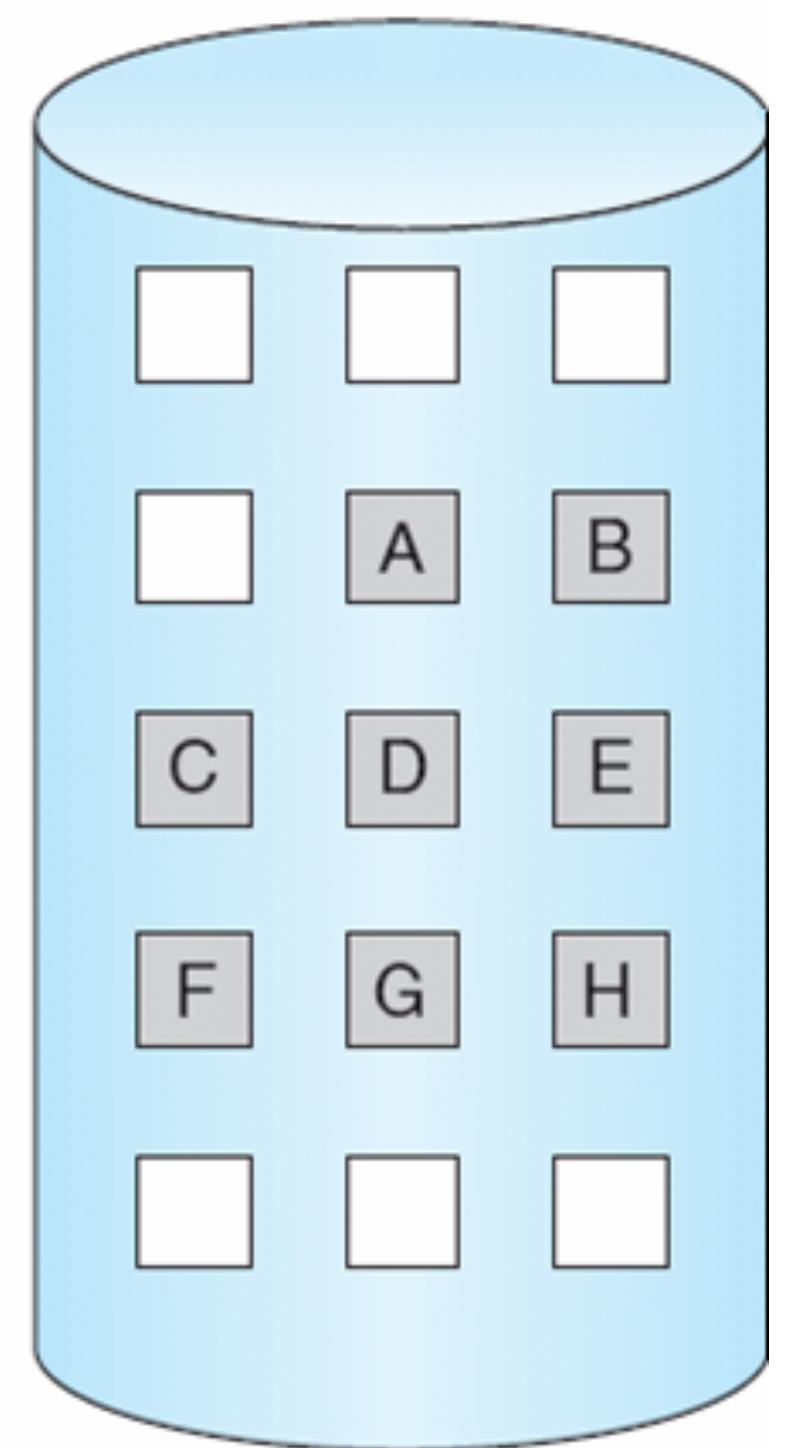
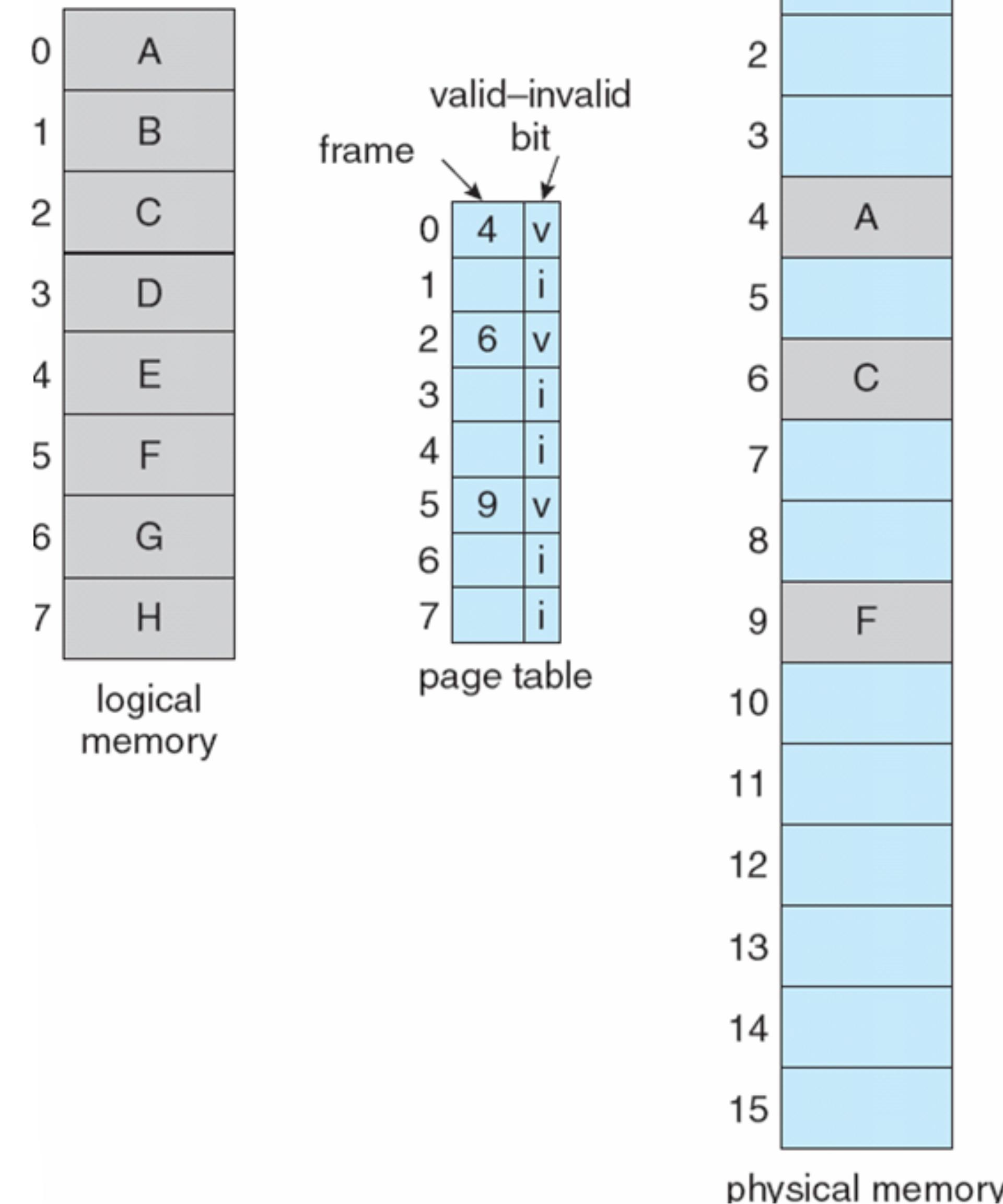


Valid-Invalid Bit

- **Hardware support** is needed to distinguish between the pages that are in memory and the ones that are on the disk.
- We can re-use the support provided by the **valid-invalid bit** in the page table.
 - Bit == **valid** then page is in memory (and is valid).
 - Bit == **invalid** then page is either not a valid one for that process or is valid but is currently in disk (pager needs to bring it to main memory).

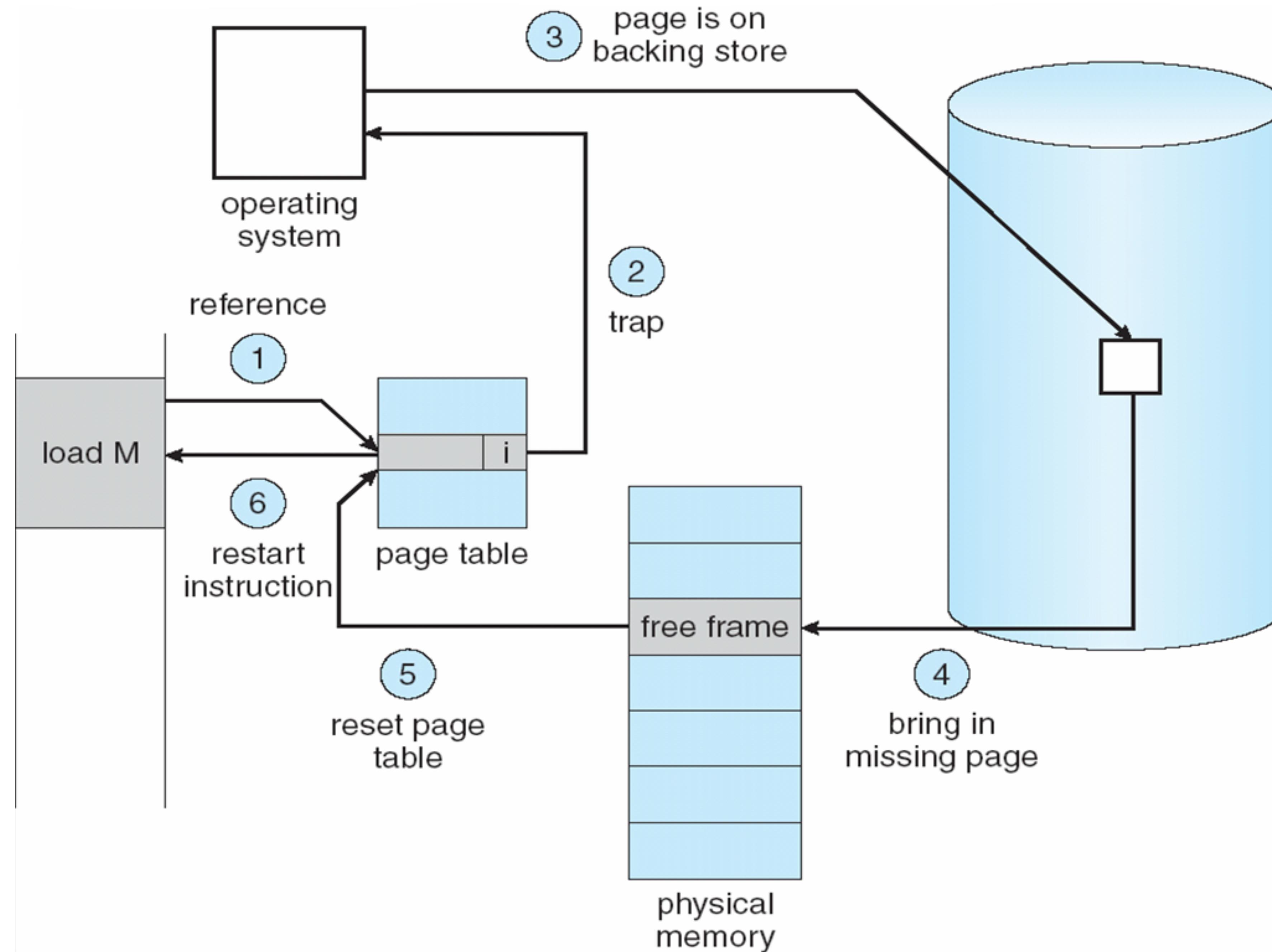
Valid-Invalid Bit

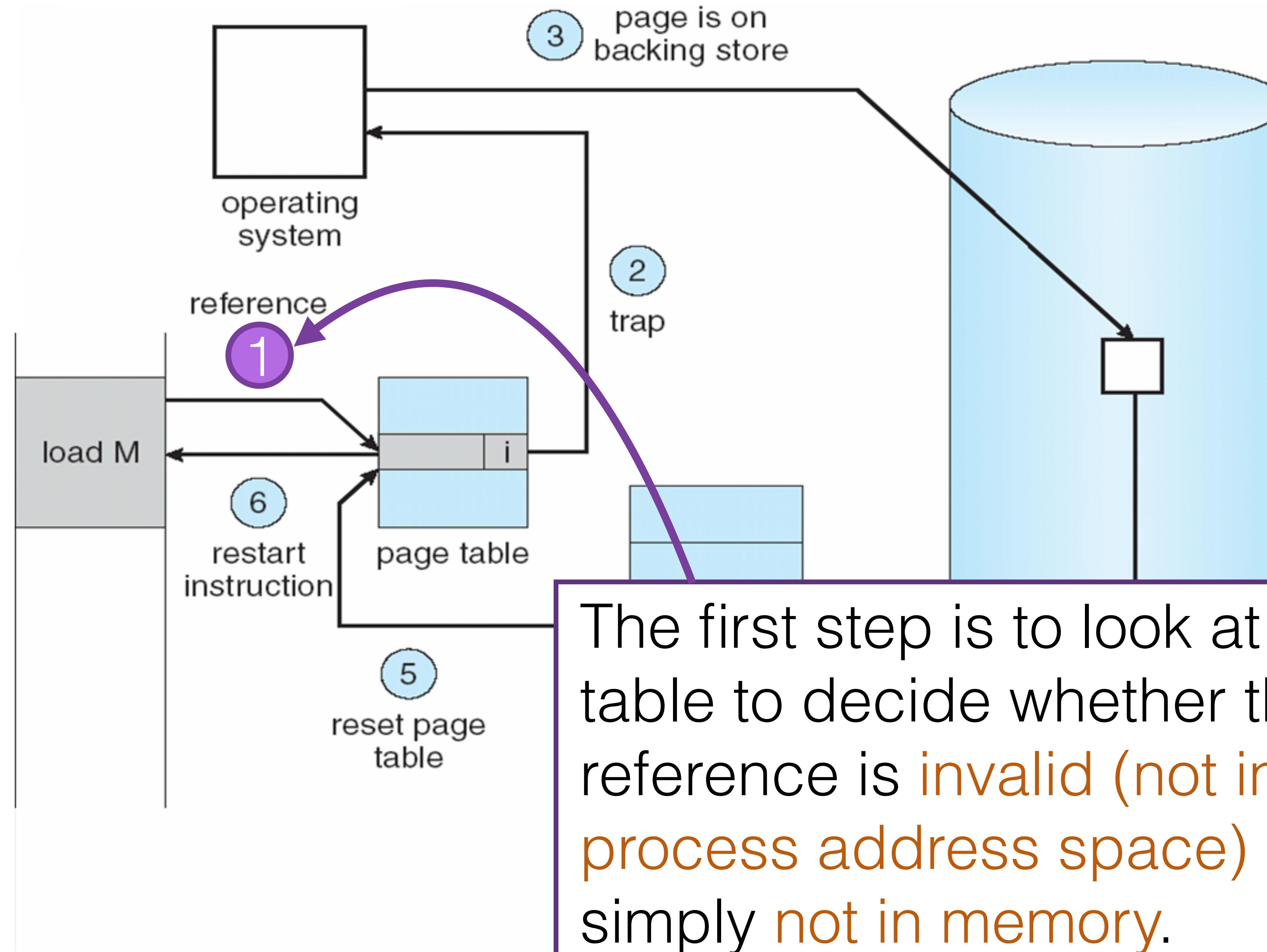
- Marking a page invalid has no effect if the process never attempts to access that page.
- Pages that are in memory are called **memory resident**.



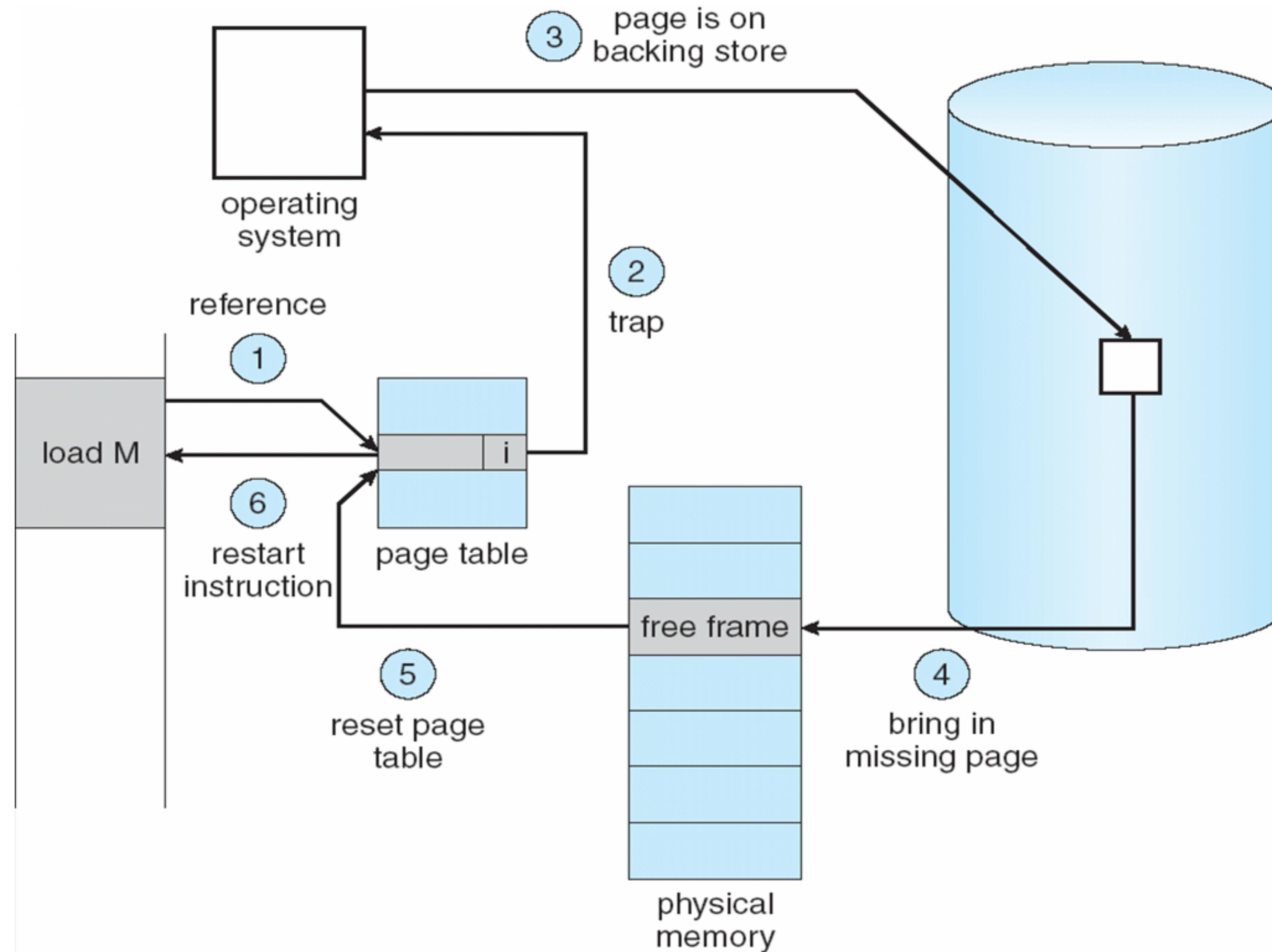
Page Faults

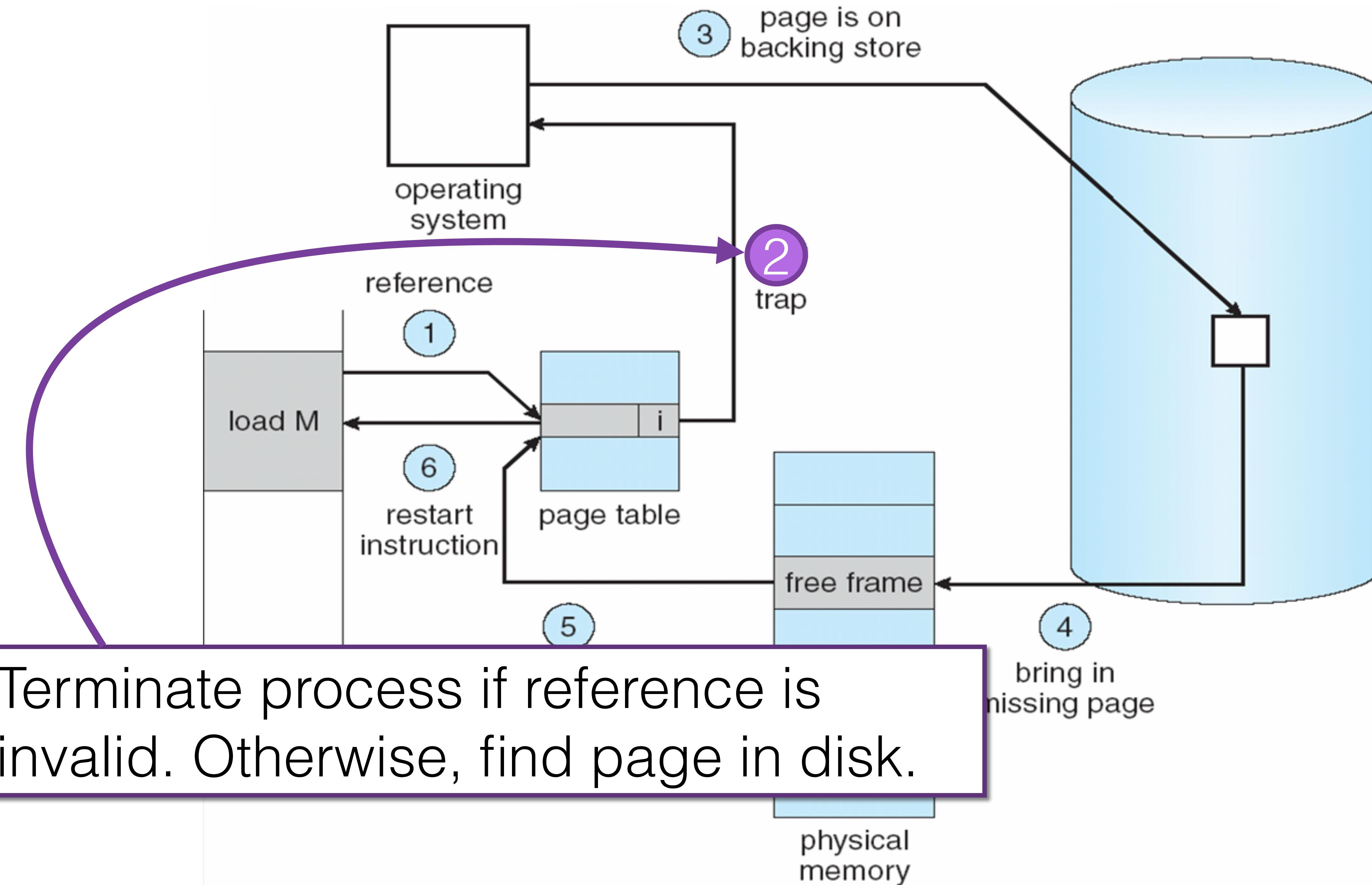
- What happens when a process tries to access non-resident pages?
 - Page Fault: A trap that results because the OS's failed to bring the desired page into memory.

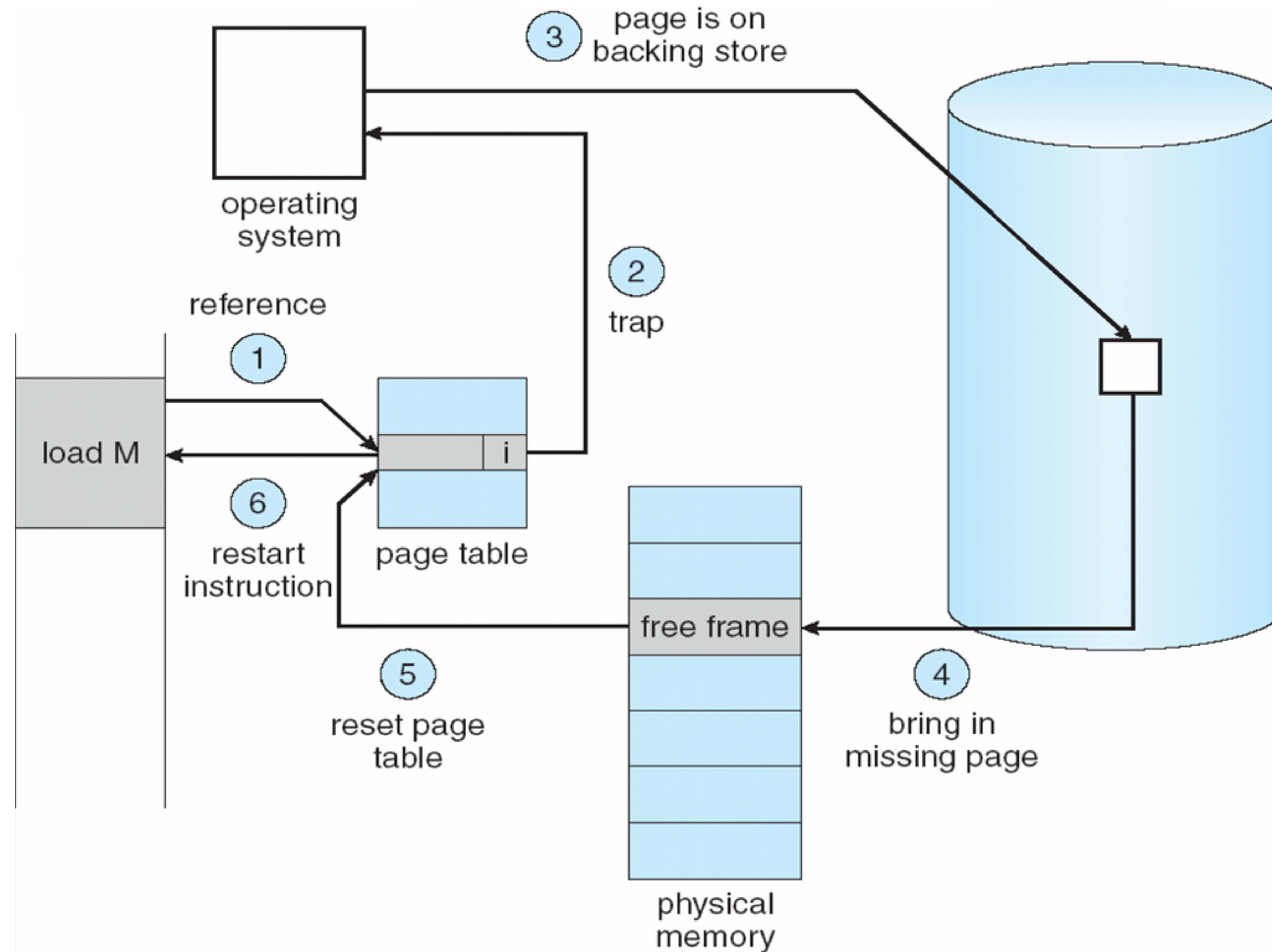


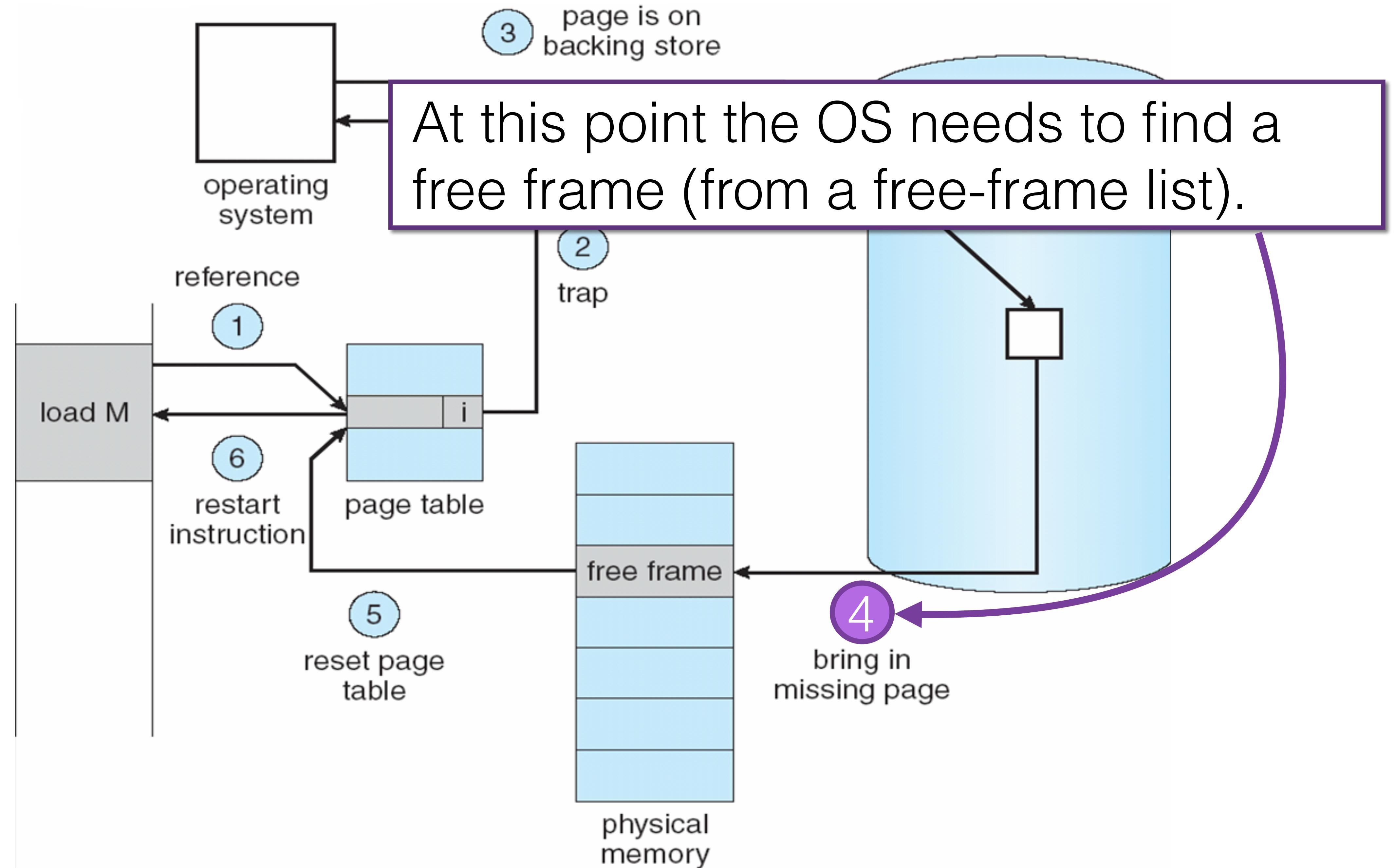


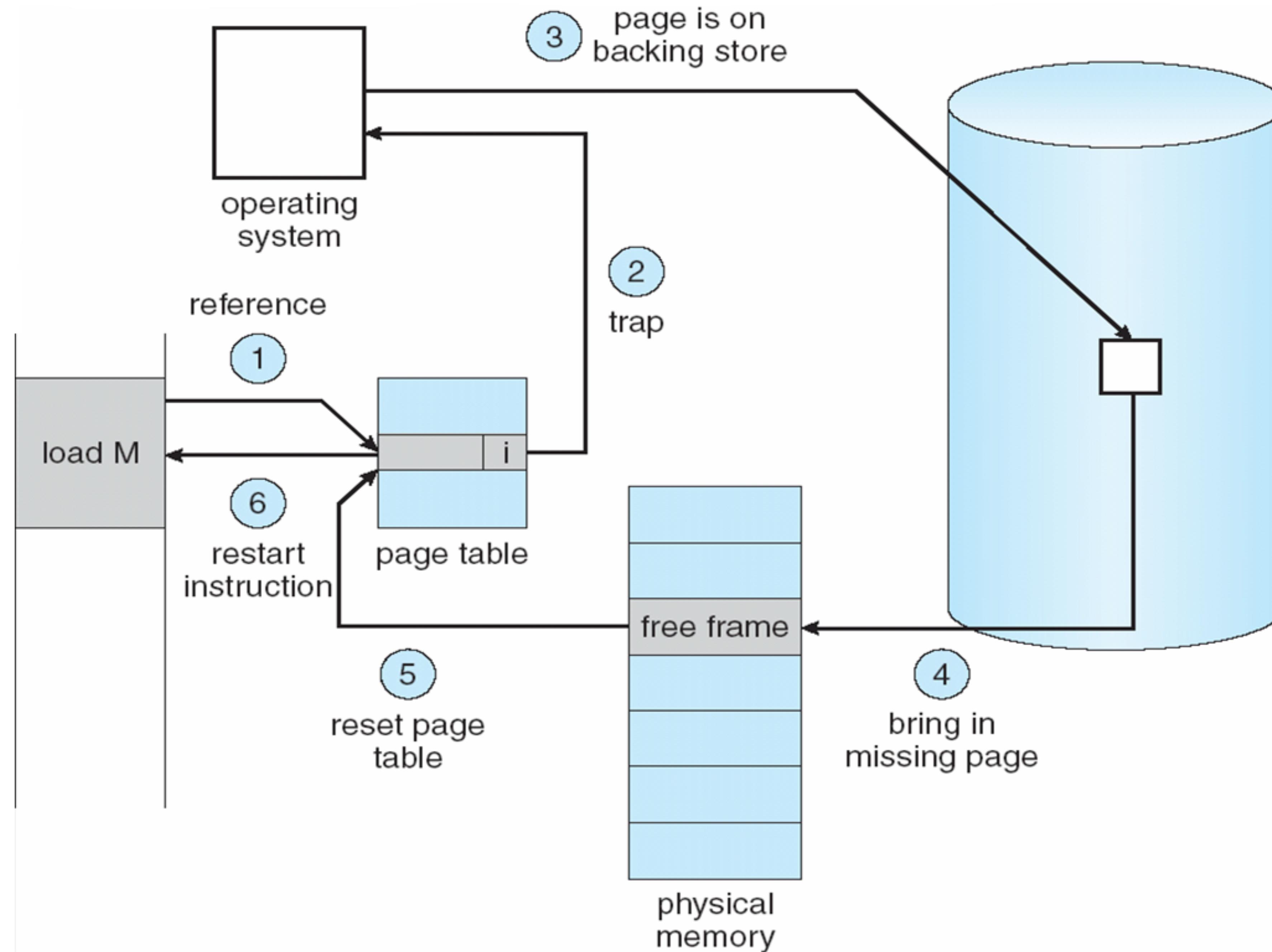
The first step is to look at another table to decide whether the actual reference is invalid (not in the process address space) or is simply not in memory.





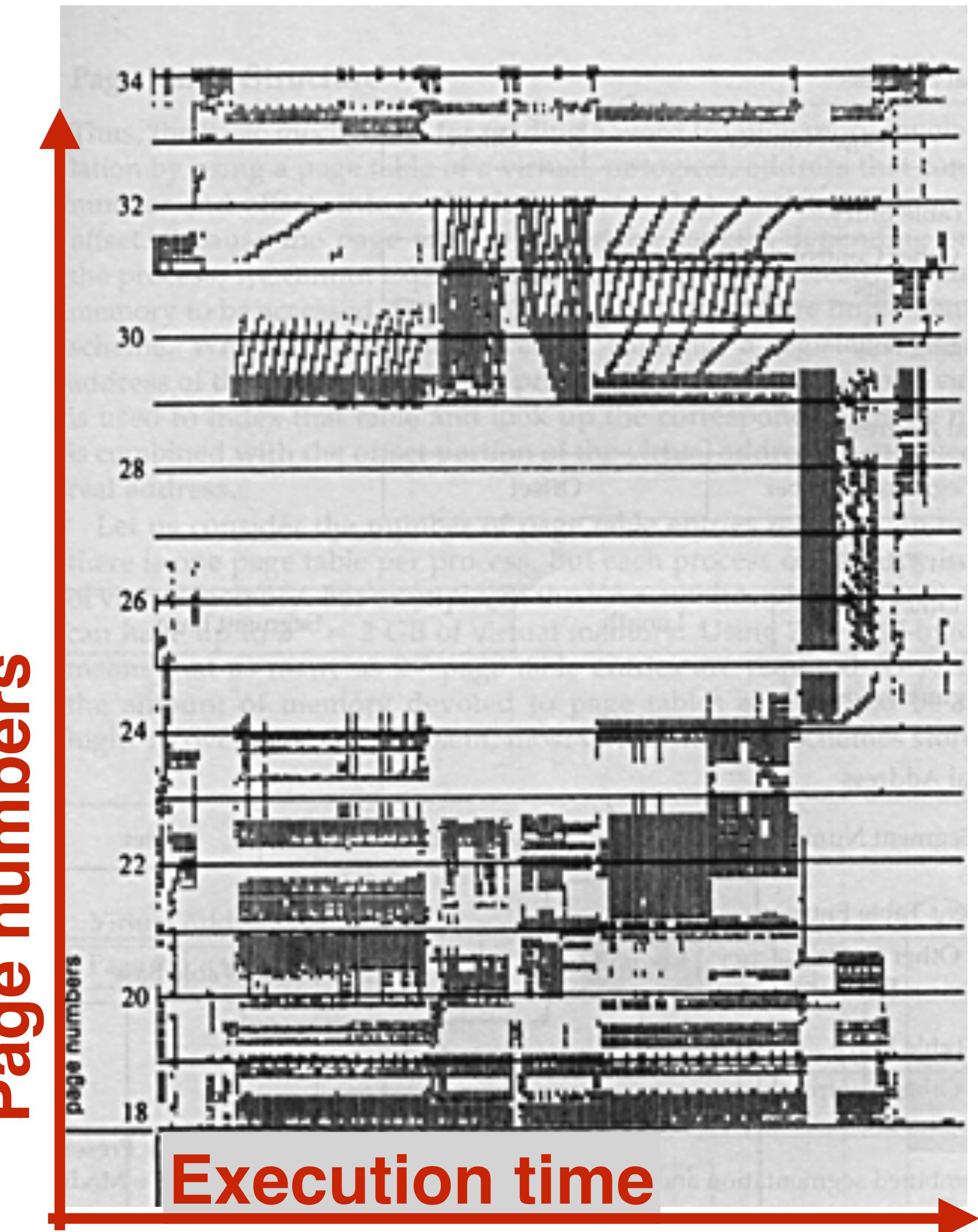






Locality in memory-reference pattern

- Theoretically, some programs could access several new pages with a single instruction.
- In this case, system performance could be seriously degraded.
- Luckily, this behavior is unlikely.



Writing code with demand-paging in mind...

■ Program structure

- Int[128,128] data;
- Each row is stored in one page

● Program 1

```
for (j = 0; j < 128; j++)
    for (i = 0; i < 128; i++)
        data[i, j] = 0;
```

128 x 128 = 16,384 page faults

● Program 2

```
for (i = 0; i < 128; i++)
    for (j = 0; j < 128; j++)
        data[i, j] = 0;
```

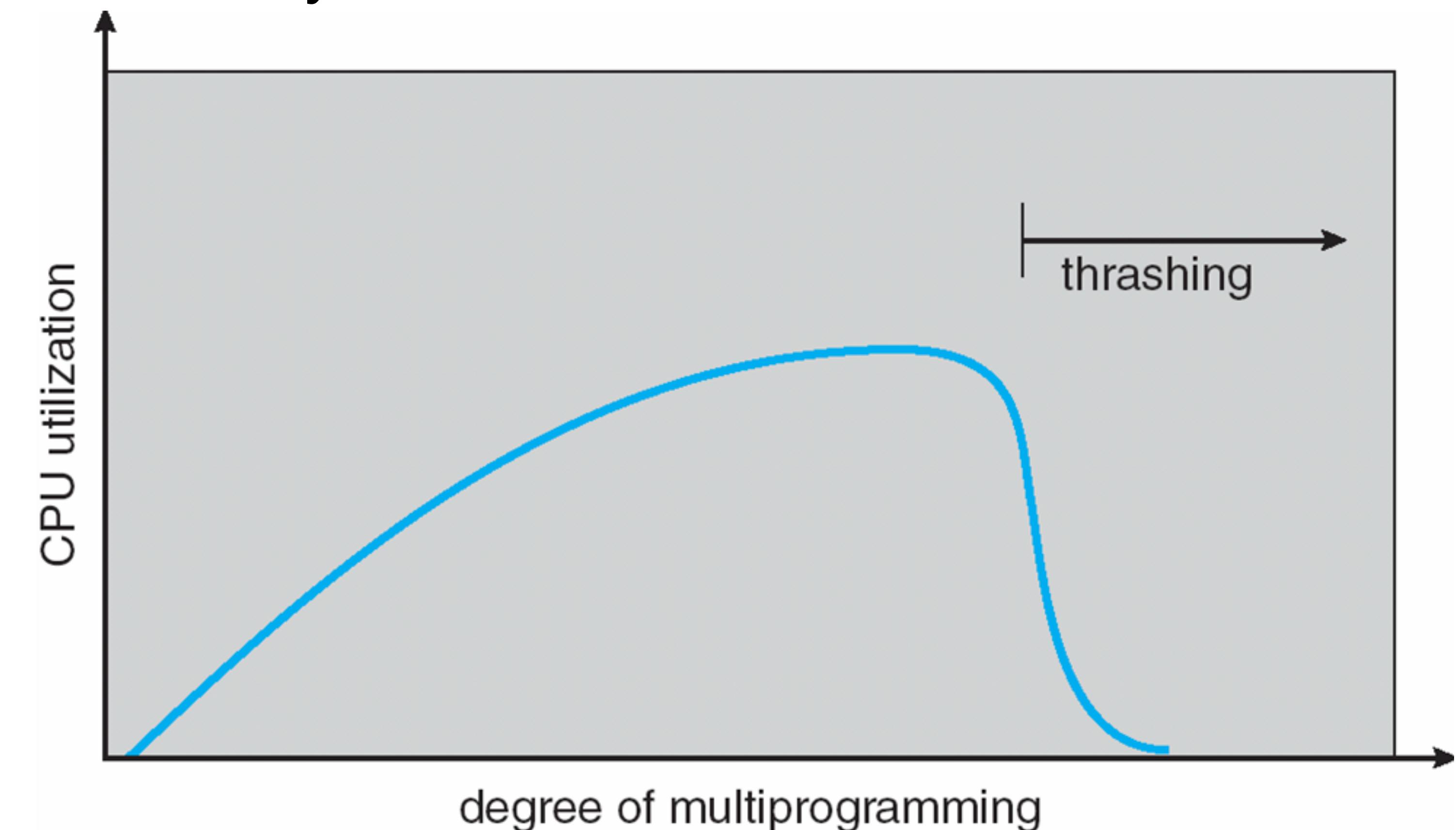
128 page faults

Thrashing

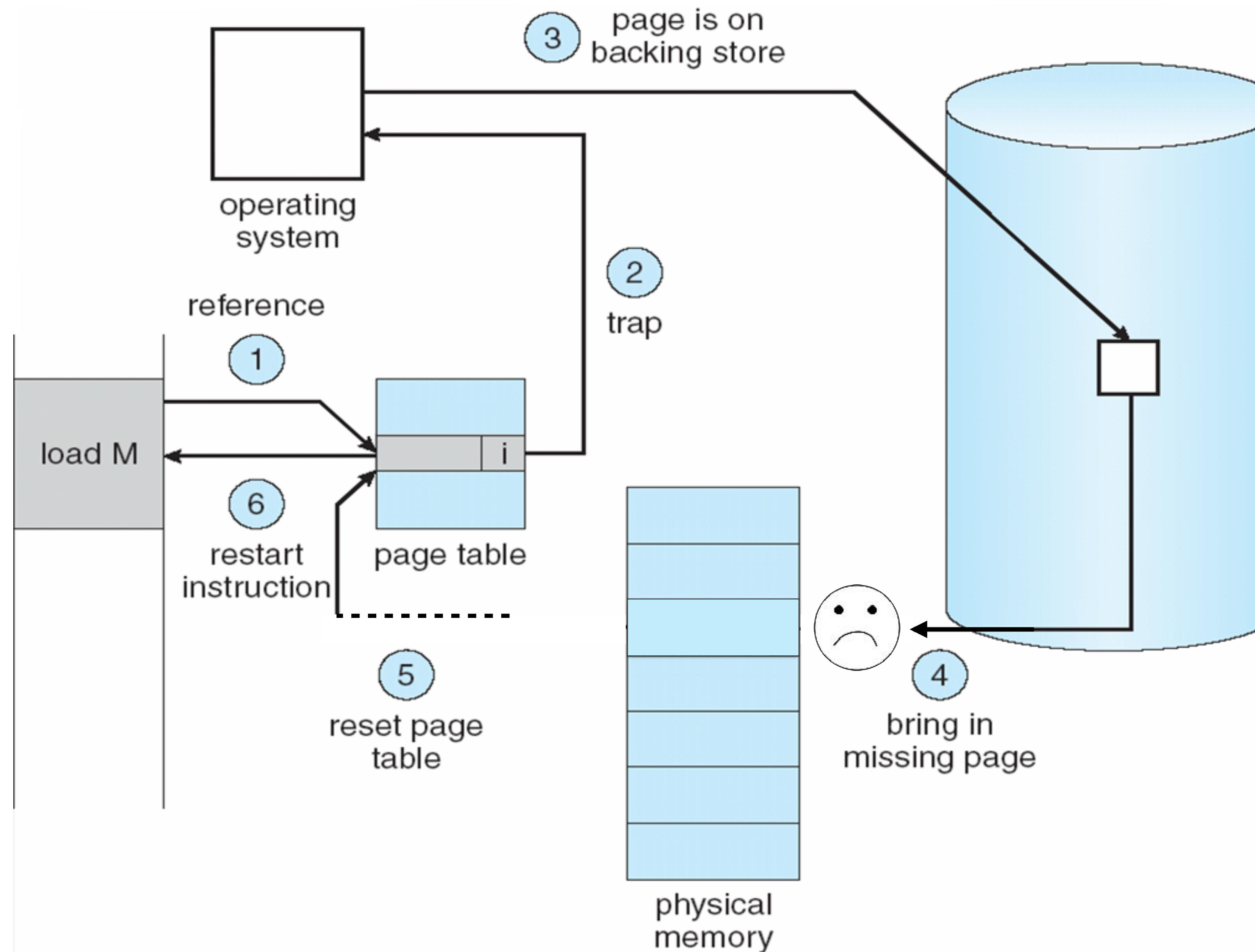
- The process does not have “enough” pages, the page-fault rate is very high and CPU becomes sub-utilized.
- The OS wants to maximize CPU utilization. As a result, it decides that it is a good idea to increase the degree of multiprogramming by adding new processes to the system.
- **Thrashing:** A process is spending more time paging than executing.

Thrashing

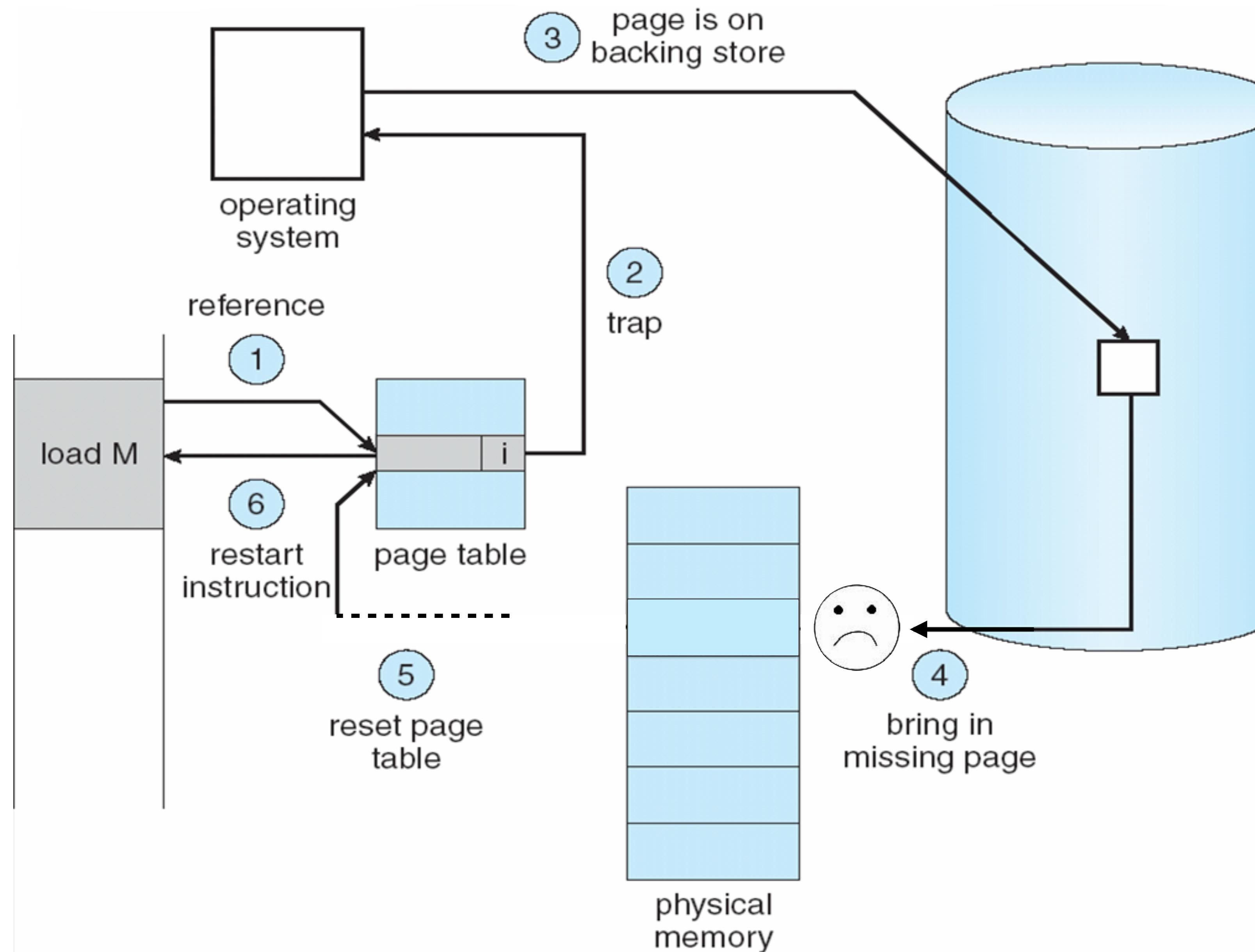
- The OS wants to maximize CPU utilization. As a result, it decides that it is a good idea to increase the degree of multiprogramming by adding new processes to the system.



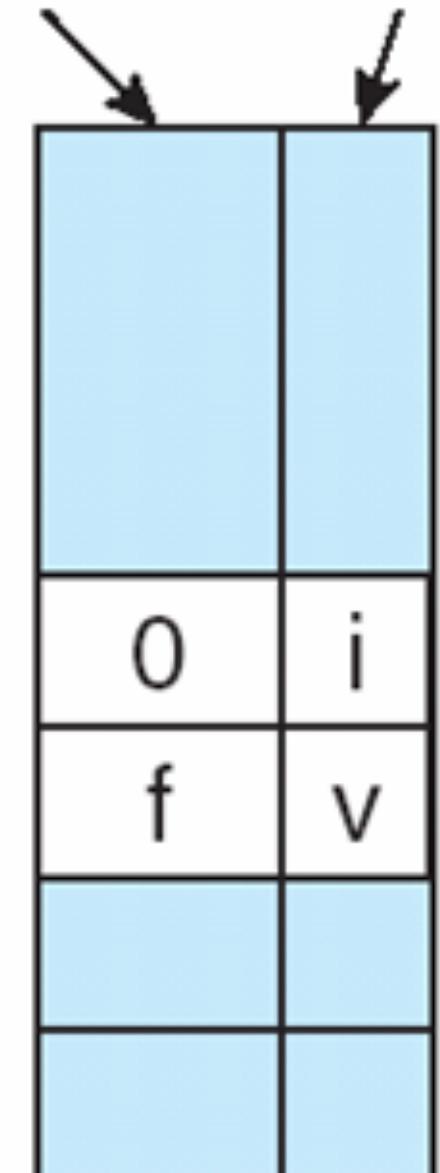
What happens if there is no free frame?



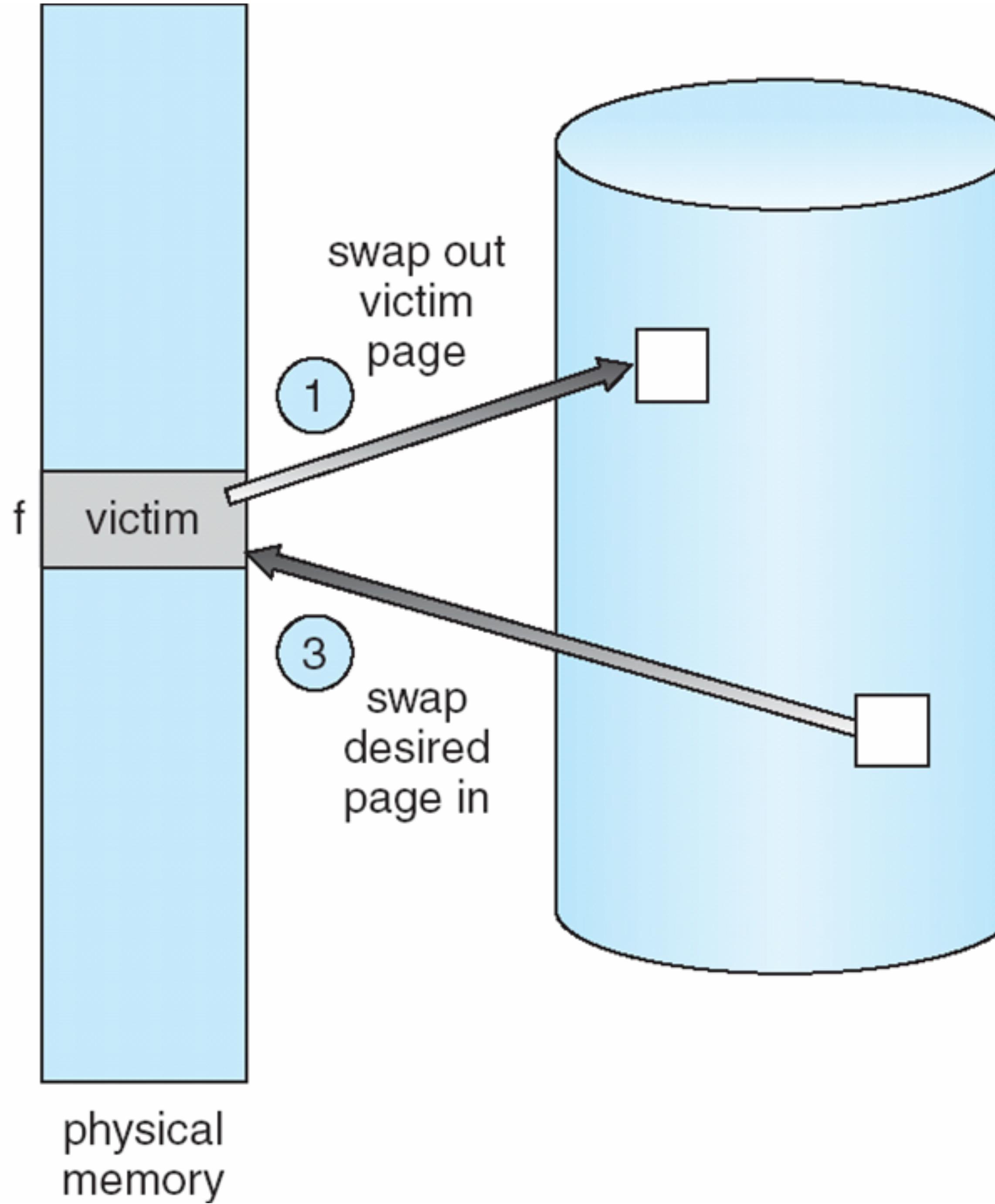
What happens if there is no free frame?



frame valid-invalid bit



- 2 change to invalid
- 4 reset page table for new page



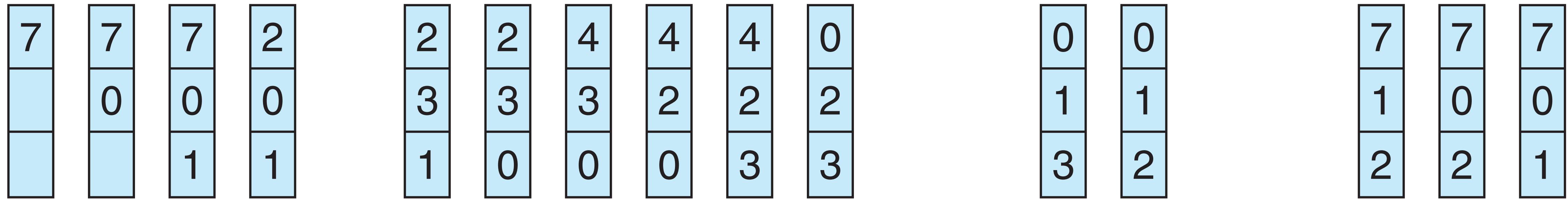
Page-Replacement Algorithms

- FIFO algorithm
- Optimal page-replacement algorithm
- Least-recently used (LRU) algorithm
- Second-chance algorithm (clock)

FIFO Algorithms

reference string

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

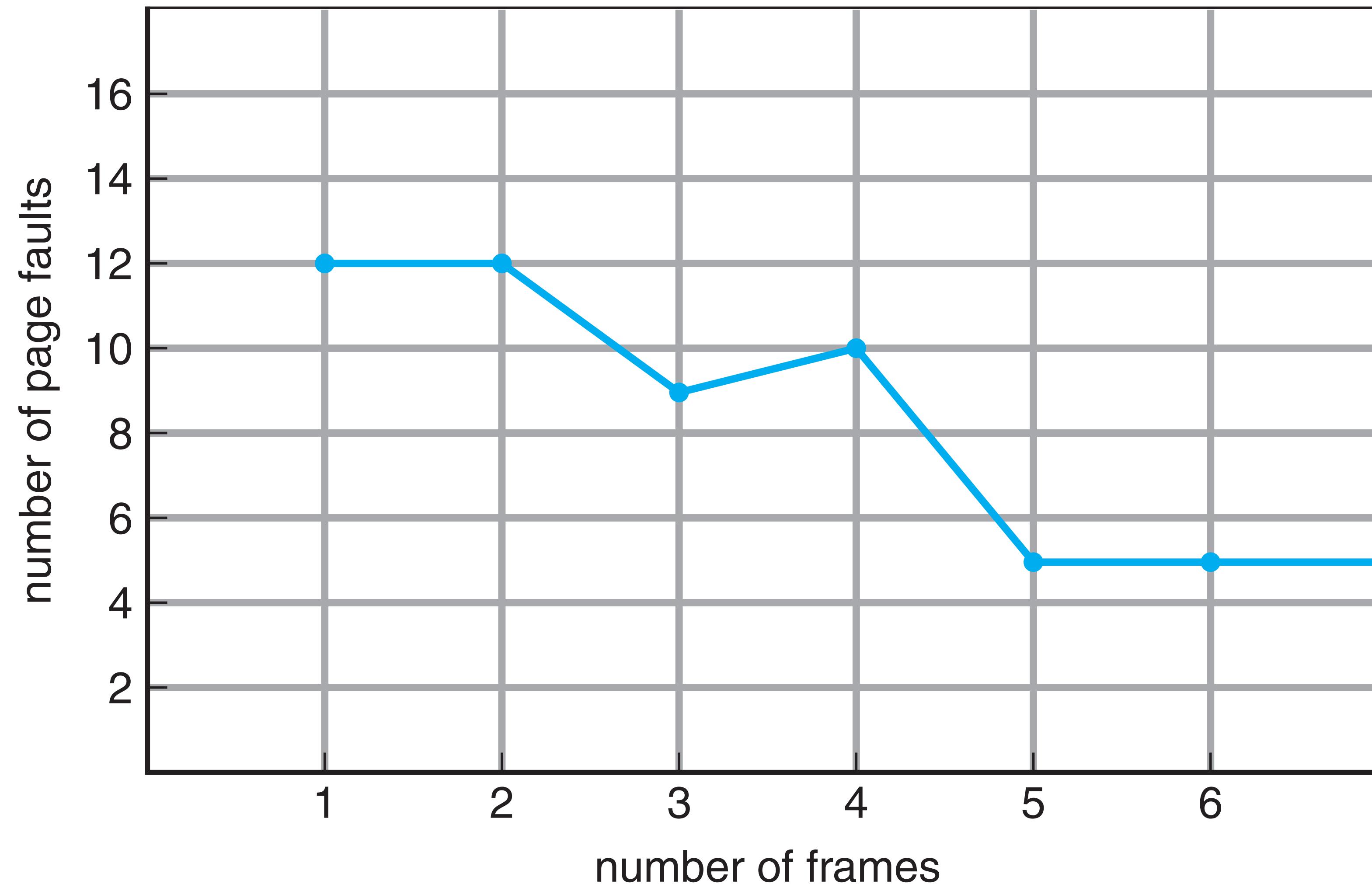


page frames

FIFO Algorithms

| Access | Hit/Miss? | Evict | Resulting Cache State | |
|--------|-----------|-------|-----------------------|---------|
| 0 | Miss | | First-in→ | 0 |
| 1 | Miss | | First-in→ | 0, 1 |
| 2 | Miss | | First-in→ | 0, 1, 2 |
| 0 | Hit | | First-in→ | 0, 1, 2 |
| 1 | Hit | | First-in→ | 0, 1, 2 |
| 3 | Miss | 0 | First-in→ | 1, 2, 3 |
| 0 | Miss | 1 | First-in→ | 2, 3, 0 |
| 3 | Hit | | First-in→ | 2, 3, 0 |
| 1 | Miss | 2 | First-in→ | 3, 0, 1 |
| 2 | Miss | 3 | First-in→ | 0, 1, 2 |
| 1 | Hit | | First-in→ | 0, 1, 2 |

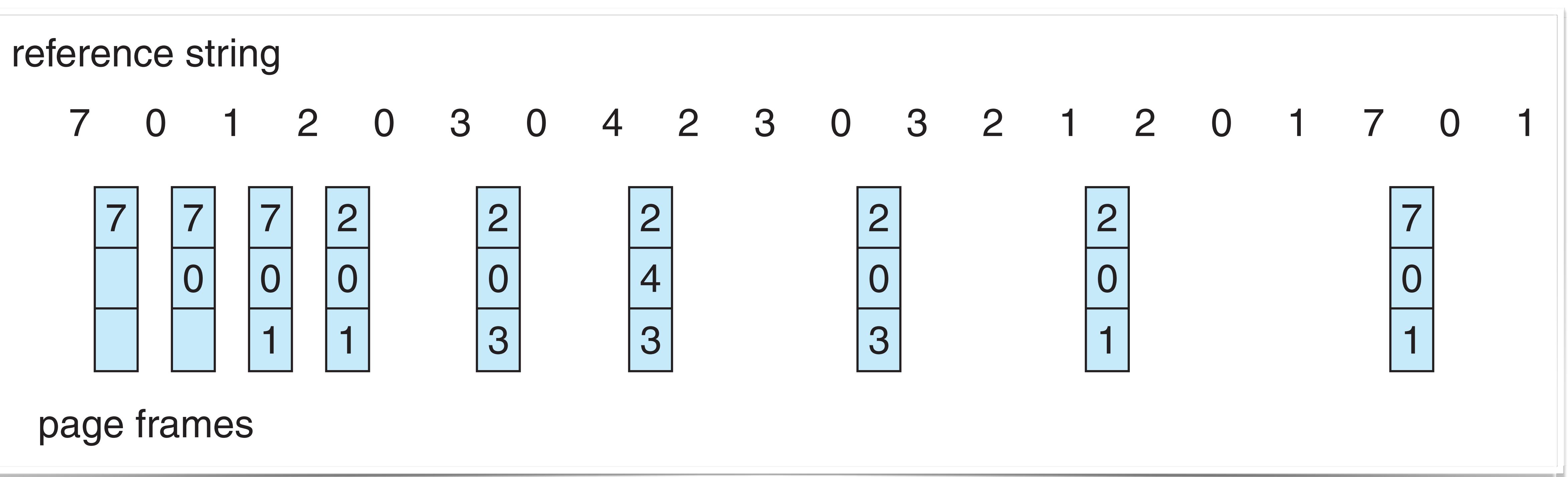
Belady's anomaly



Paper: L. A. Belady, R. A. Nelson, G. S. Shedler, An anomaly in space-time characteristics of certain programs running in paging machine, Comm. ACM , 12, 1 (1969) 349–353.

Optimal Algorithm

Policy: Replace the page that will not be used for the longest period of time.



Optimal Algorithm

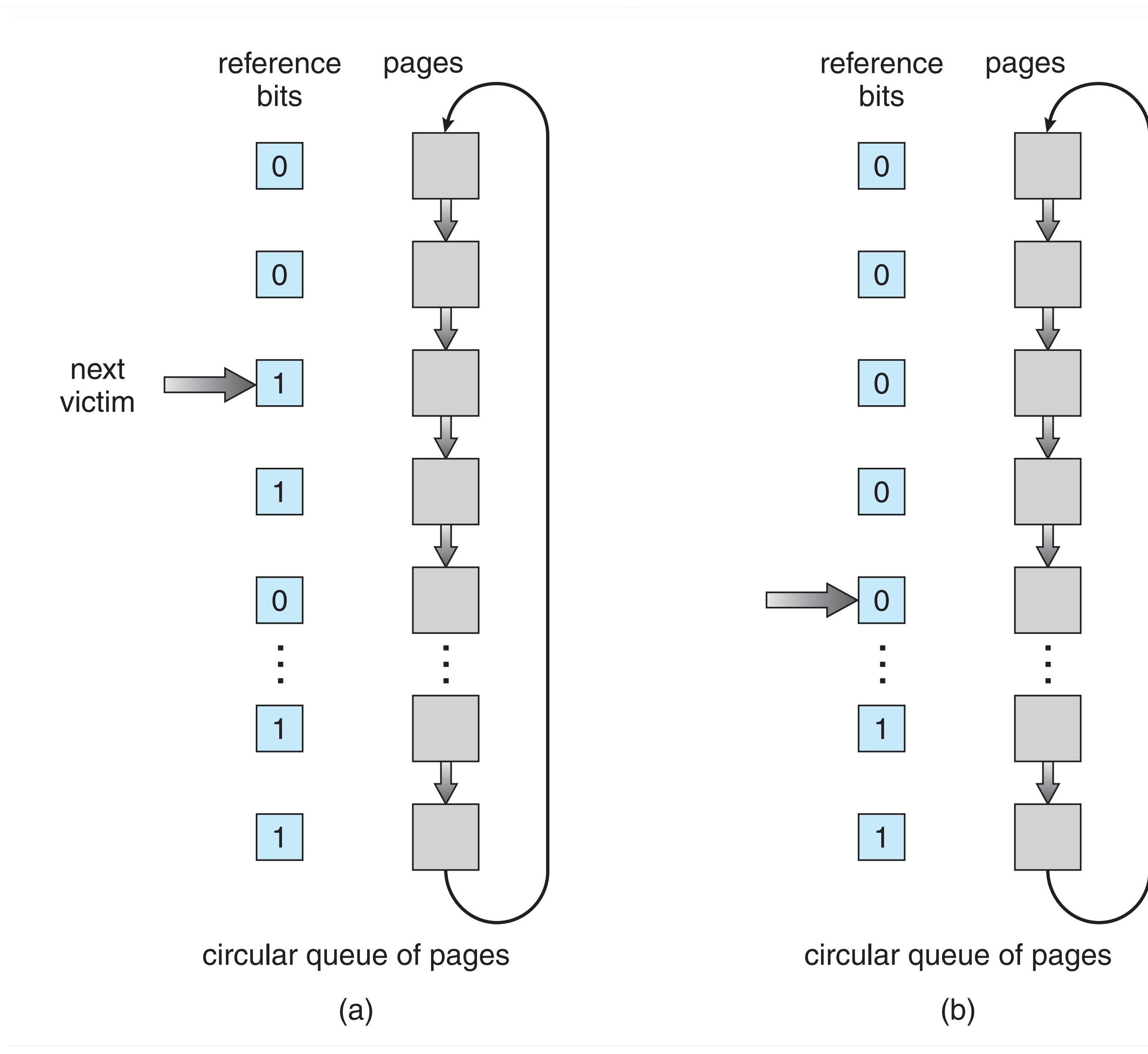
| Access | Hit/Miss? | Evict | Resulting Cache State |
|--------|-----------|-------|-----------------------|
| 0 | Miss | | 0 |
| 1 | Miss | | 0, 1 |
| 2 | Miss | | 0, 1, 2 |
| 0 | Hit | | 0, 1, 2 |
| 1 | Hit | | 0, 1, 2 |
| 3 | Miss | 2 | 0, 1, 3 |
| 0 | Hit | | 0, 1, 3 |
| 3 | Hit | | 0, 1, 3 |
| 1 | Hit | | 0, 1, 3 |
| 2 | Miss | 3 | 0, 1, 2 |
| 1 | Hit | | 0, 1, 2 |

LRU Algorithm

| Access | Hit/Miss? | Evict | Resulting Cache State |
|--------|-----------|-------|-----------------------|
| 0 | Miss | | LRU→ 0 |
| 1 | Miss | | LRU→ 0, 1 |
| 2 | Miss | | LRU→ 0, 1, 2 |
| 0 | Hit | | LRU→ 1, 2, 0 |
| 1 | Hit | | LRU→ 2, 0, 1 |
| 3 | Miss | 2 | LRU→ 0, 1, 3 |
| 0 | Hit | | LRU→ 1, 3, 0 |
| 3 | Hit | | LRU→ 1, 0, 3 |
| 1 | Hit | | LRU→ 0, 3, 1 |
| 2 | Miss | 0 | LRU→ 3, 1, 2 |
| 1 | Hit | | LRU→ 3, 2, 1 |

Second-chance Algorithm

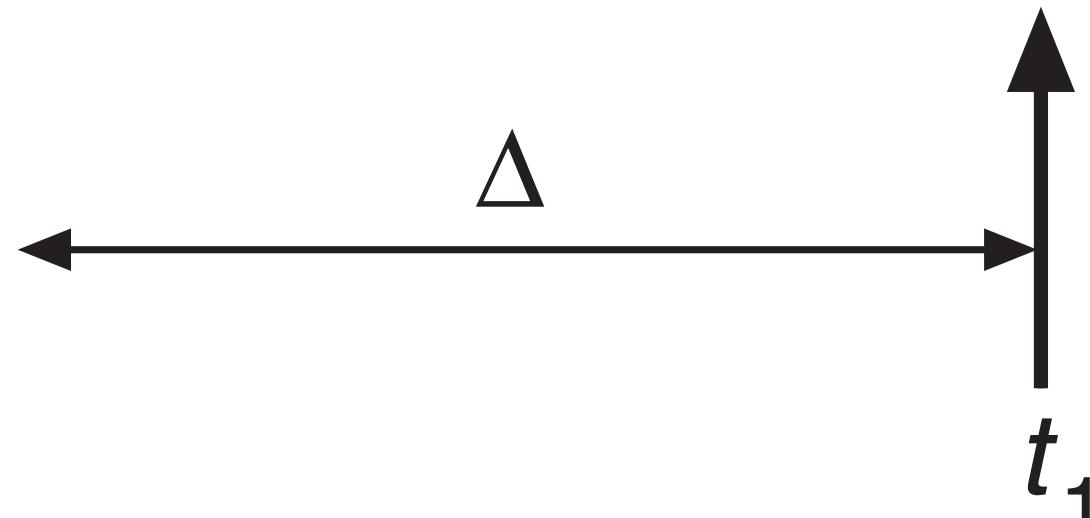
- Whenever a page is referenced, the hardware sets the reference bit to 1.
- The O.S. sets the reference bit to 0 according to some policy.
- Evicting is free if page is not *dirty*.
 - Clock prioritize scan for pages that are both unused and clean.



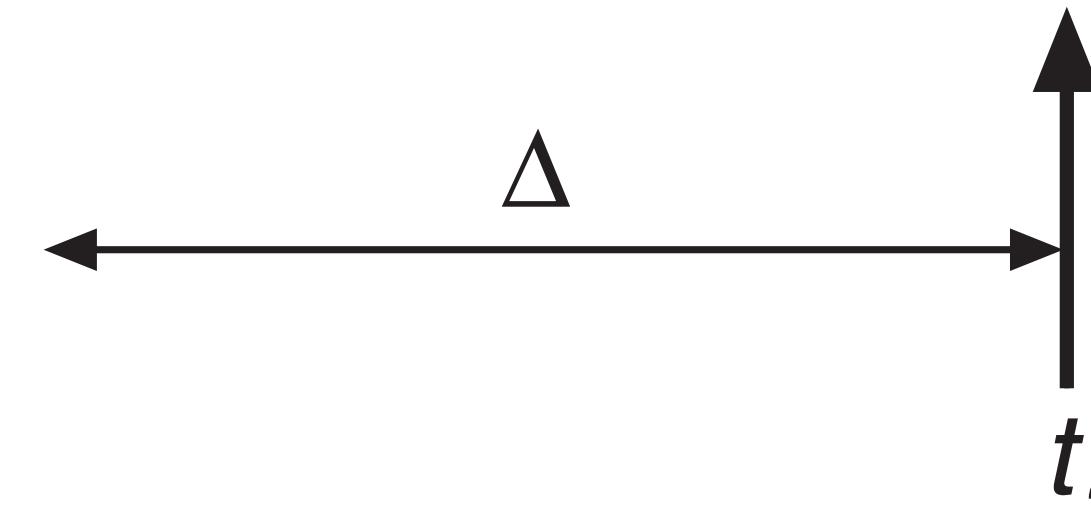
Working-set Model

page reference table

... 2 6 1 5 7 7 7 5 1 6 2 3 4 1 2 3 4 4 4 3 4 3 4 4 4 4 1 3 2 3 4 4 4 3 4 4 4 ...

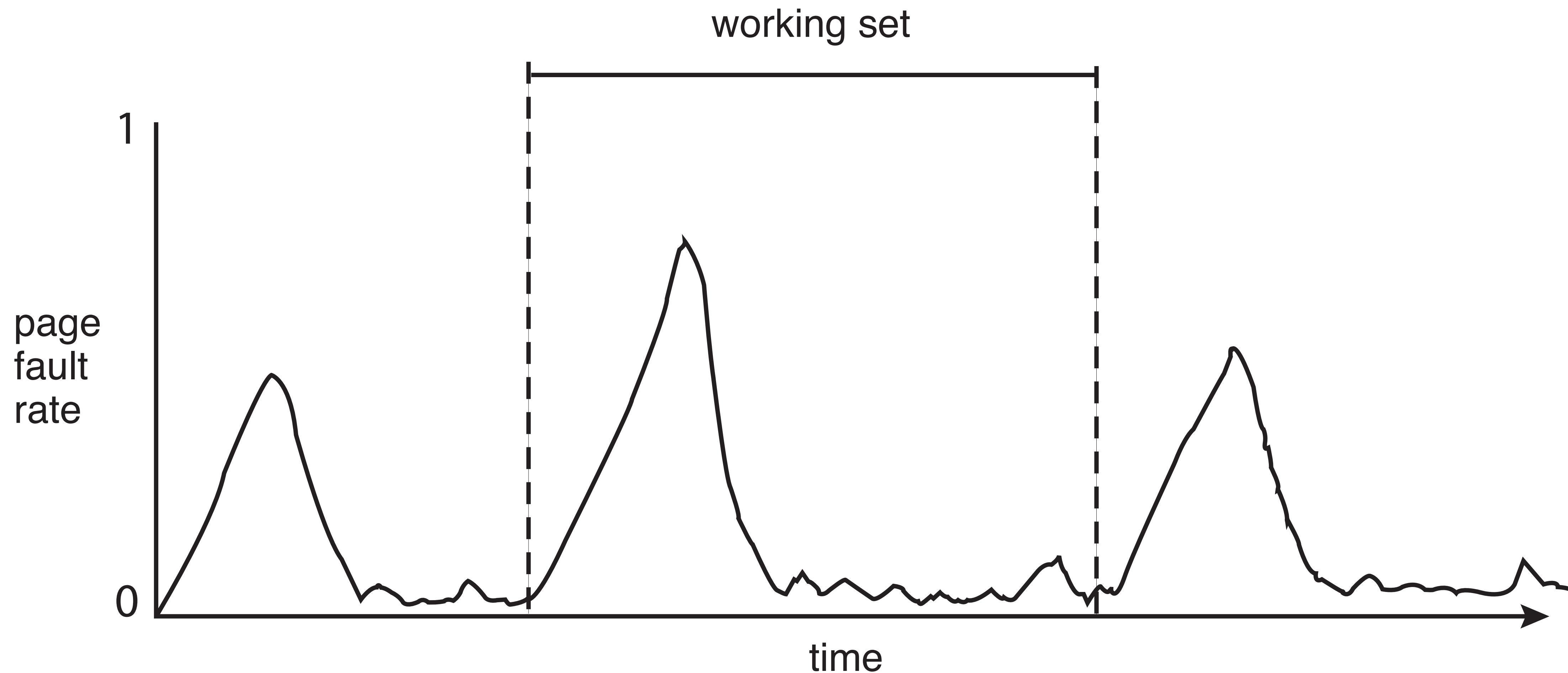


$$WS(t_1) = \{1, 2, 5, 6, 7\}$$

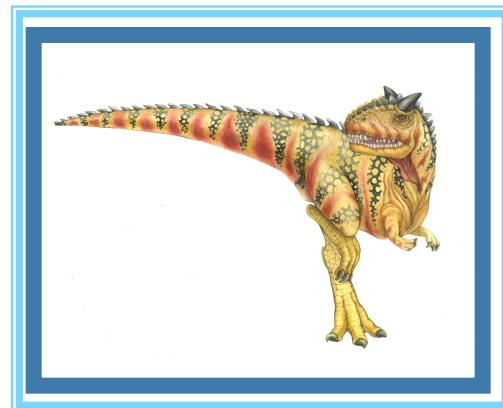


$$WS(t_2) = \{3, 4\}$$

Working Sets and Page-fault frequency



Chapter 10: File-System Interface





File System

- Two main parts:
 - A collection of files
 - A directory structure





File Concept

- Contiguous logical address space
- Types:
 - Data
 - ▶ numeric
 - ▶ character
 - ▶ binary
 - Program





File Structure

- None - sequence of words, bytes
- Simple record structure
 - Lines
 - Fixed length
 - Variable length
- Complex Structures
 - Formatted document
 - Relocatable load file





File Attributes

- **Name** – only information kept in human-readable form
- **Identifier** – unique tag (number) identifies file within file system
- **Type** – needed for systems that support different types
- **Location** – pointer to file location on device
- **Size** – current file size
- **Protection** – controls who can do reading, writing, executing
- **Time, date, and user identification** – data for protection, security, and usage monitoring
- Information about files are kept in the directory structure, which is maintained on the disk





File Operations

■ File is an **abstract data type**

- **Create**
- **Write**
- **Read**
- **Reposition within file** (i.e., seek)
- **Delete**
- **Truncate**
- $\text{Open}(F_i)$ – search the directory structure on disk for entry F_i , and move the content of entry to memory
- $\text{Close } (F_i)$ – move the content of entry F_i in memory to directory structure on disk





Open Files

- Several pieces of data are needed to manage open files:
 - **File pointer:** pointer to last read/write location, per process that has the file open
 - **File-open count:** counter of number of times a file is open – to allow removal of data from open-file table when last processes closes it
 - **Disk location of the file:** cache of data access information
 - **Access rights:** per-process access mode information



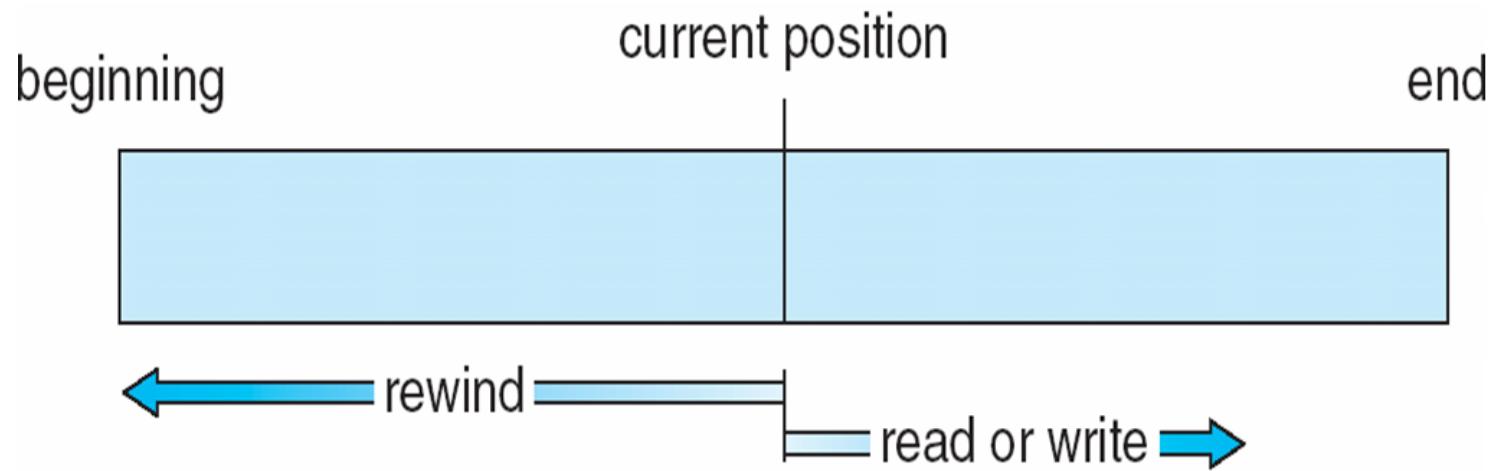


File Types – Name, Extension

| file type | usual extension | function |
|----------------|--------------------------|---|
| executable | exe, com, bin or none | ready-to-run machine-language program |
| object | obj, o | compiled, machine language, not linked |
| source code | c, cc, java, pas, asm, a | source code in various languages |
| batch | bat, sh | commands to the command interpreter |
| text | txt, doc | textual data, documents |
| word processor | wp, tex, rtf, doc | various word-processor formats |
| library | lib, a, so, dll | libraries of routines for programmers |
| print or view | ps, pdf, jpg | ASCII or binary file in a format for printing or viewing |
| archive | arc, zip, tar | related files grouped into one file, sometimes compressed, for archiving or storage |
| multimedia | mpeg, mov, rm, mp3, avi | binary file containing audio or A/V information |



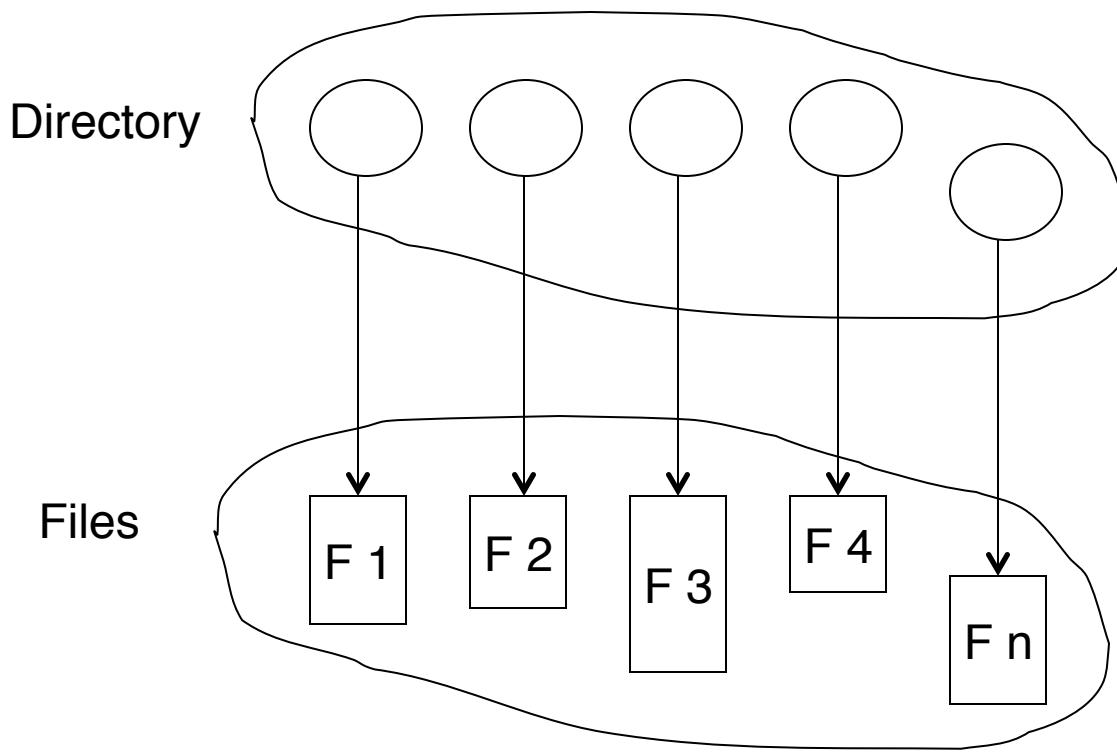
Sequential-access File





Directory Structure

- A collection of nodes containing information about all files



Both the directory structure and the files reside on disk
Backups of these two structures are kept on tapes





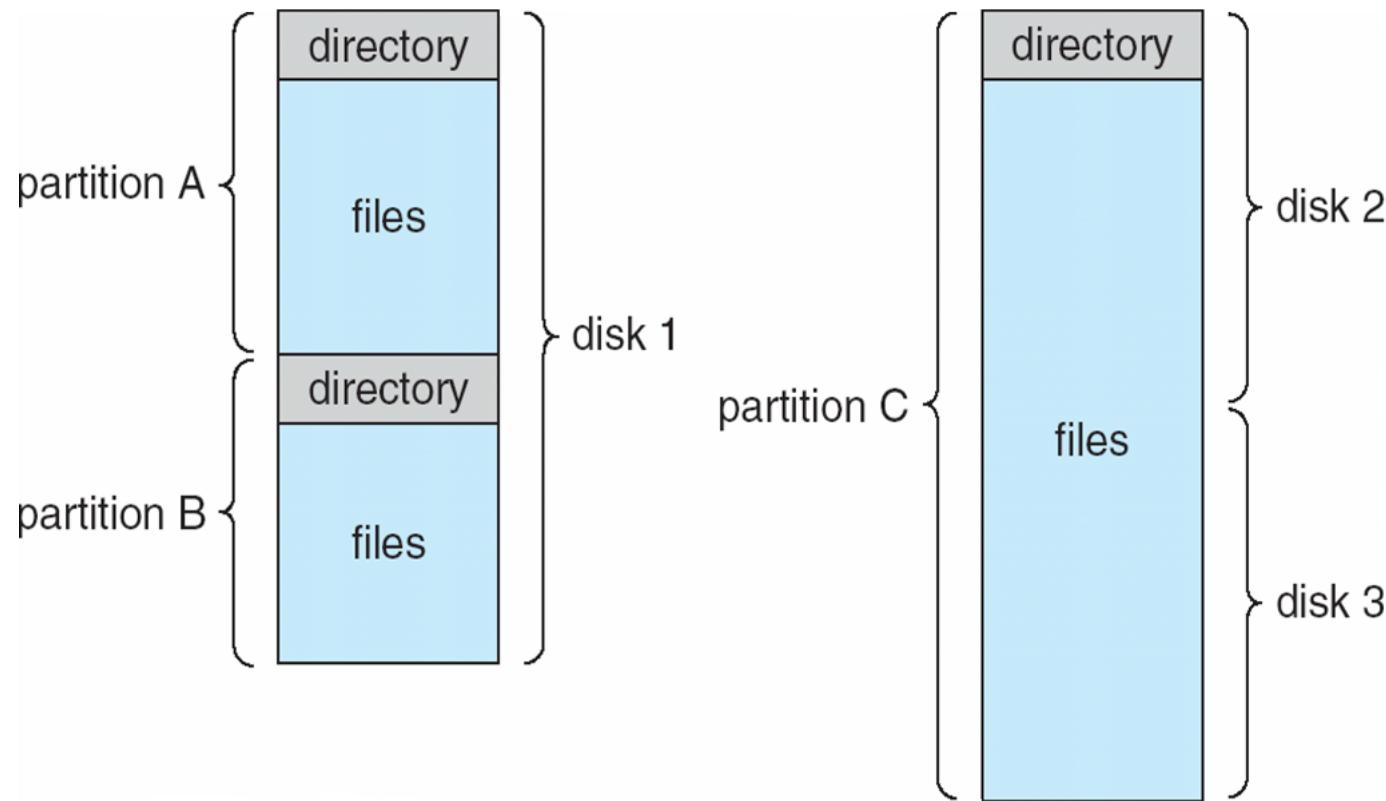
Disk Structure

- Disk can be subdivided into **partitions**
- Disks or partitions can be **RAID** protected against failure
- Disk or partition can be used **raw** – without a file system, or **formatted** with a file system
- Partitions also known as minidisks, slices
- Entity containing file system known as a **volume**
- Each volume containing file system also tracks that file system's info in **device directory** or **volume table of contents**
- As well as **general-purpose file systems** there are many **special-purpose file systems**, frequently all within the same operating system or computer





A Typical File-system Organization





Operations Performed on Directory

- Search for a file
- Create a file
- Delete a file
- List a directory
- Rename a file
- Traverse the file system





Organize the Directory (Logically) to Obtain

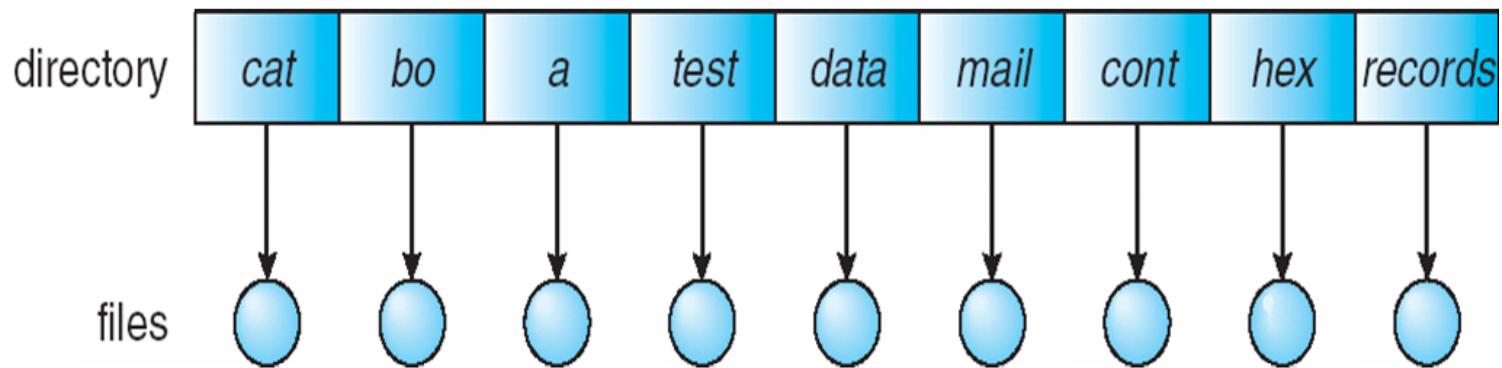
- Efficiency – locating a file quickly
- Naming – convenient to users
 - Two users can have same name for different files
 - The same file can have several different names
- Grouping – logical grouping of files by properties, (e.g., all Java programs, all games, ...)





Single-Level Directory

- A single directory for all users



Naming problem

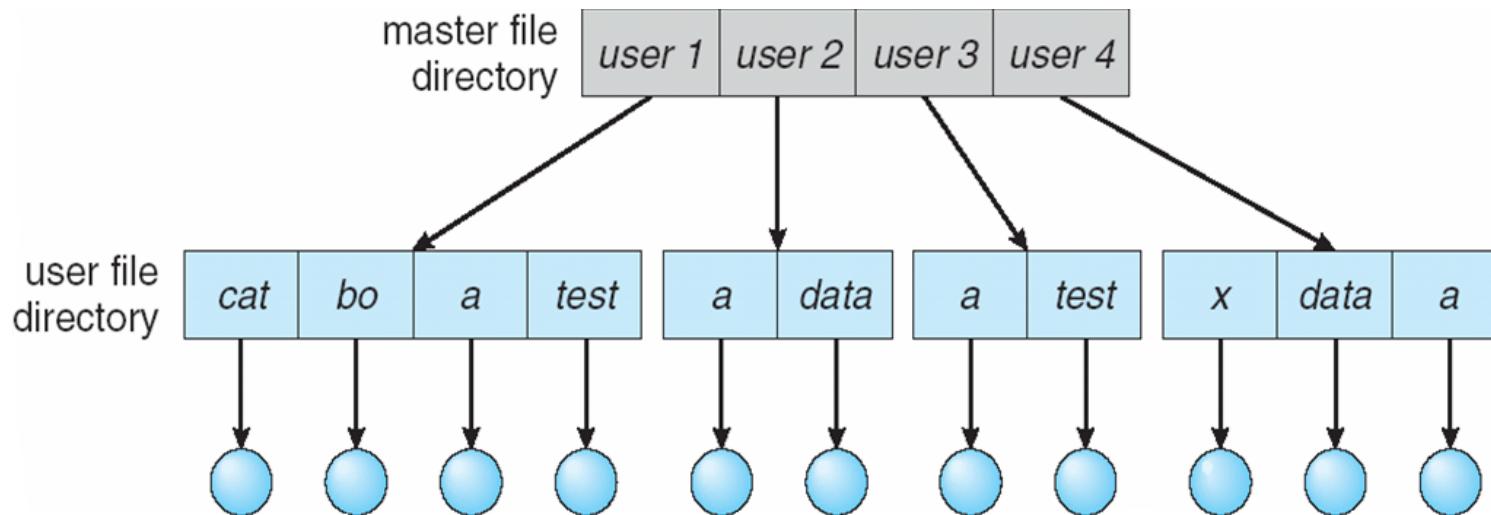
Grouping problem





Two-Level Directory

- Separate directory for each user

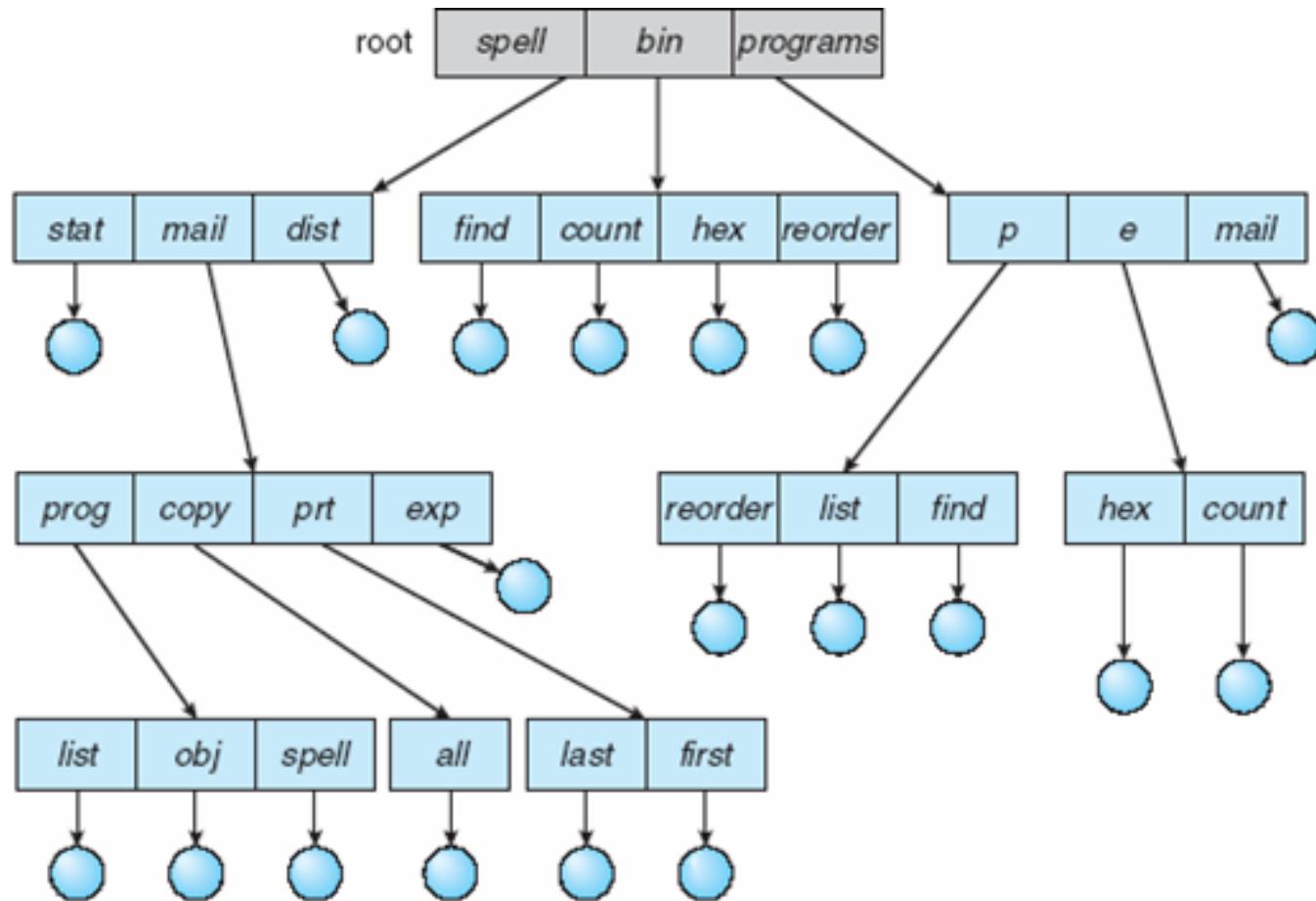


- Path name
- Can have the same file name for different user
- Efficient searching
- No grouping capability





Tree-Structured Directories





Tree-Structured Directories (Cont)

- Efficient searching
- Grouping Capability
- Current directory (working directory)
 - `cd /spell/mail/prog`
 - `type list`





Tree-Structured Directories (Cont)

- **Absolute** or **relative** path name
- Creating a new file is done in current directory
- Delete a file

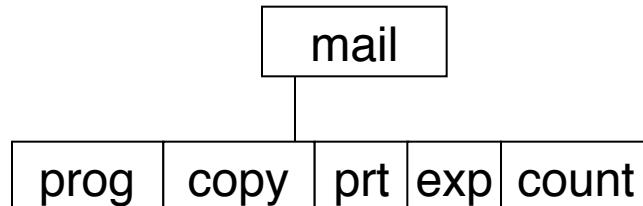
`rm <file-name>`

- Creating a new subdirectory is done in current directory

`mkdir <dir-name>`

Example: if in current directory `/mail`

`mkdir count`



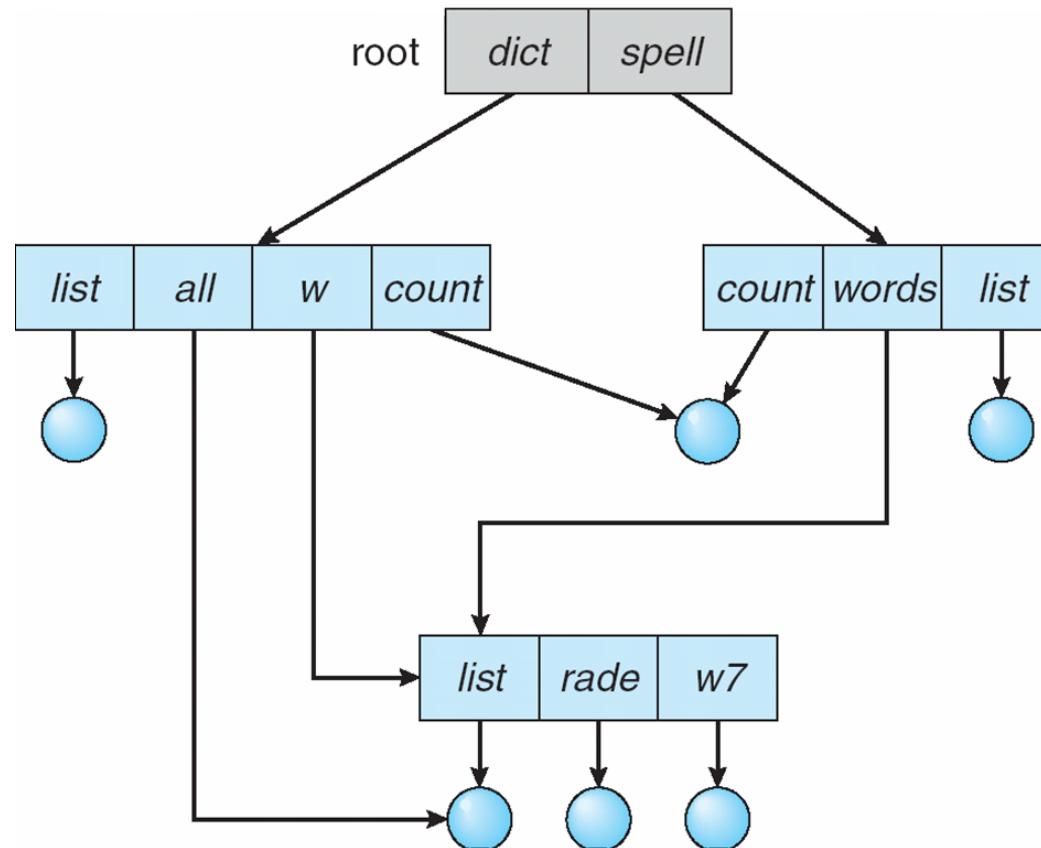
Deleting “mail” \Rightarrow deleting the entire subtree rooted by “mail”





Acyclic-Graph Directories

- Have shared subdirectories and files





Acyclic-Graph Directories (Cont.)

- Two different names (aliasing)
- If *dict* deletes *list* ⇒ dangling pointer

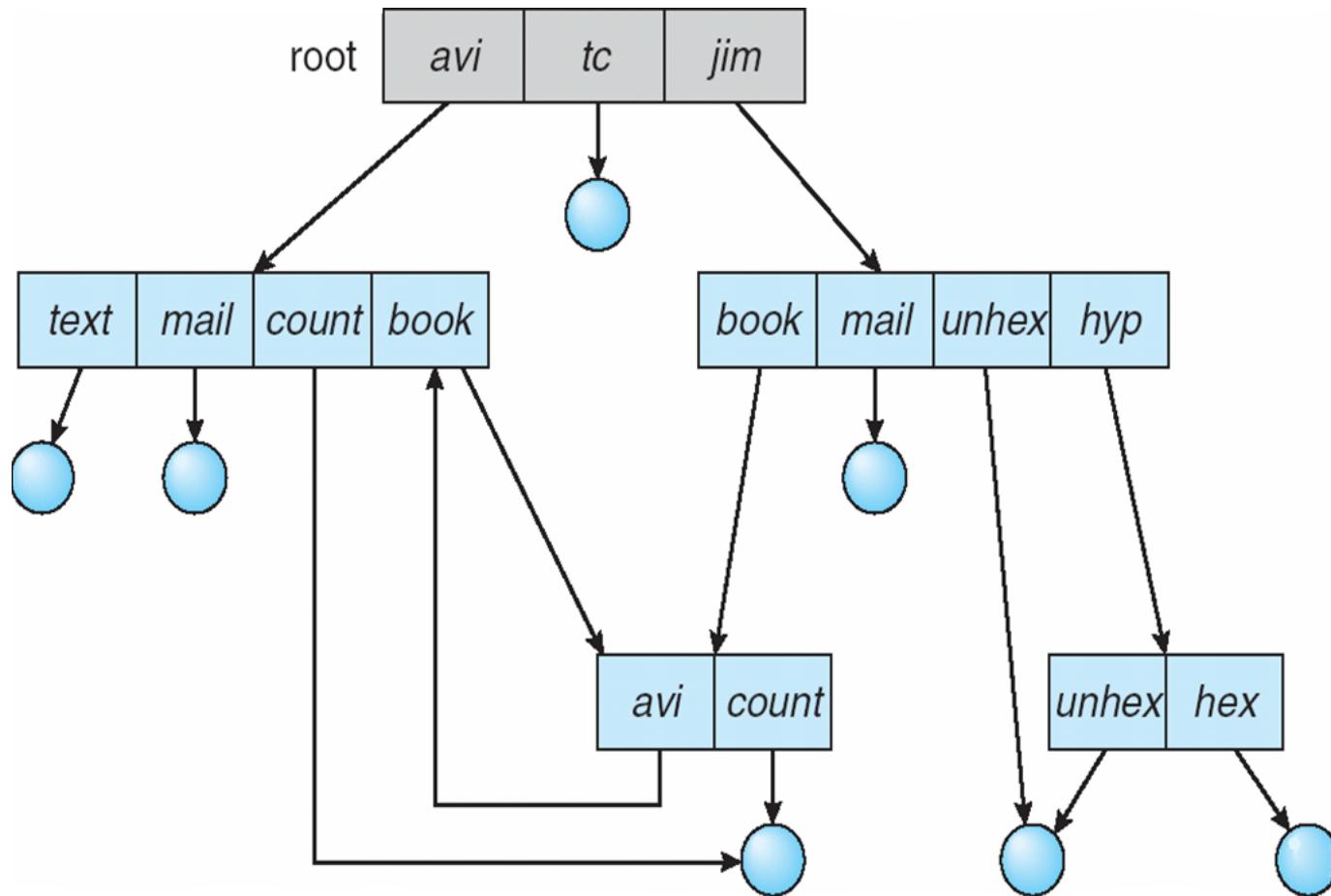
Solutions:

- Backpointers, so we can delete all pointers
Variable size records a problem
 - Backpointers using a daisy chain organization
 - Entry-hold-count solution
- New directory entry type
 - **Link** – another name (pointer) to an existing file
 - **Resolve the link** – follow pointer to locate the file





General Graph Directory





General Graph Directory (Cont.)

- How do we guarantee no cycles?
 - Allow only links to file not subdirectories
 - Garbage collection
 - Every time a new link is added use a cycle detection algorithm to determine whether it is OK





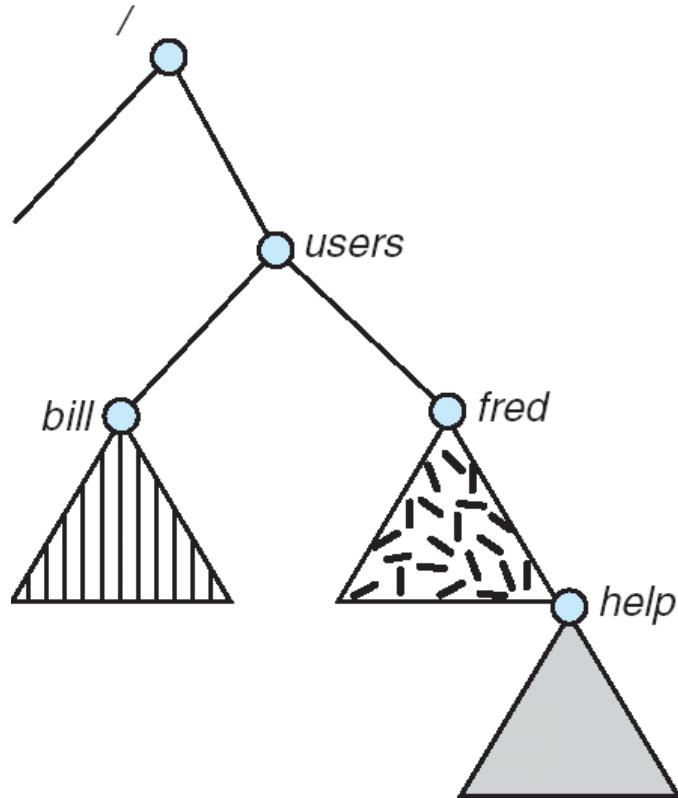
File System Mounting

- A file system must be **mounted** before it can be accessed
- A unmounted file system (i.e. Fig. 11-11(b)) is mounted at a **mount point**

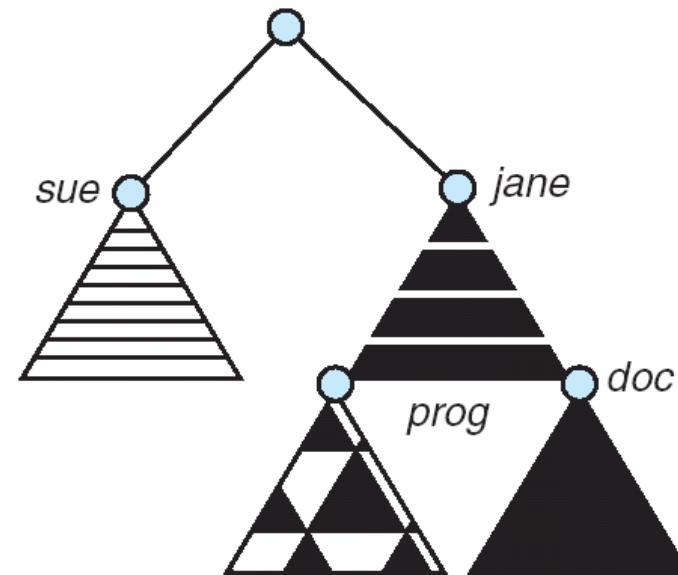




(a) Existing. (b) Unmounted Partition

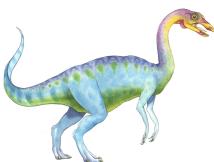


(a)

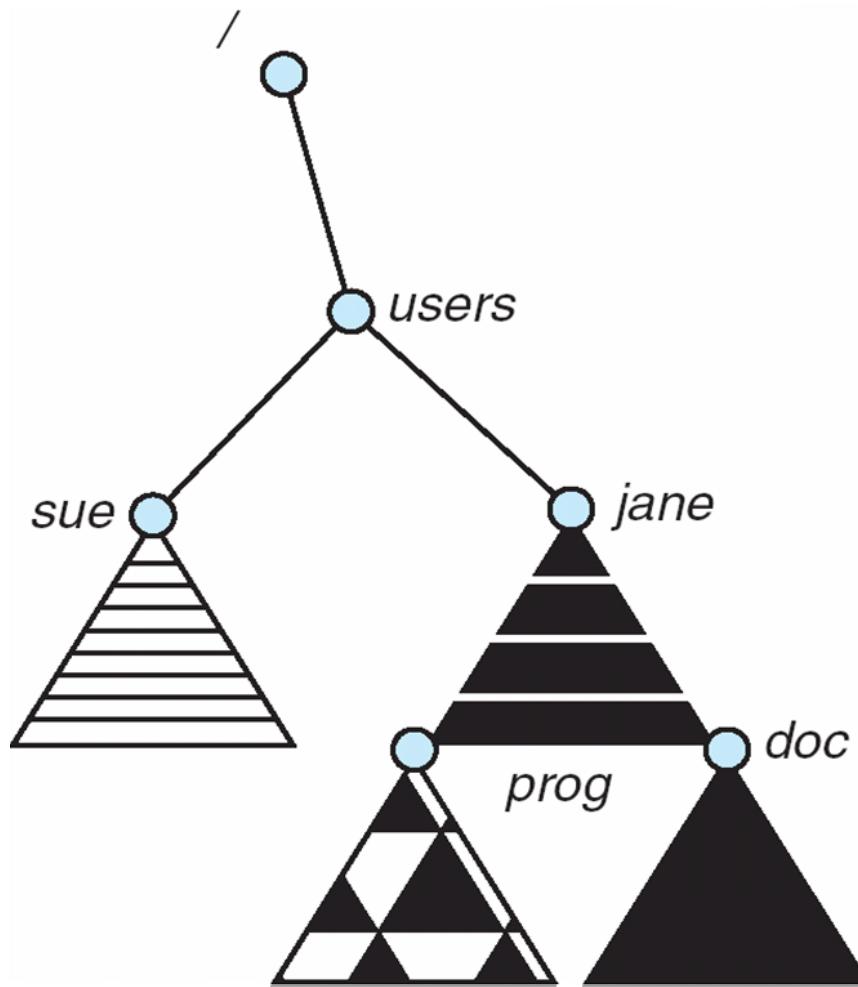


(b)





Mount Point





File Sharing

- Sharing of files on multi-user systems is desirable
- Sharing may be done through a **protection scheme**
- On distributed systems, files may be shared across a network
- Network File System (NFS) is a common distributed file-sharing method





File Sharing – Multiple Users

- **User IDs** identify users, allowing permissions and protections to be per-user
- **Group IDs** allow users to be in groups, permitting group access rights





Protection

- File owner/creator should be able to control:
 - what can be done
 - by whom

- Types of access
 - **Read**
 - **Write**
 - **Execute**
 - **Append**
 - **Delete**
 - **List**



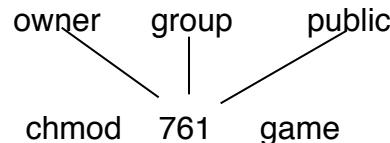


Access Lists and Groups

- Mode of access: read, write, execute
- Three classes of users

| | | | |
|------------------|---|---|-------|
| | | | RWX |
| a) owner access | 7 | ⇒ | 1 1 1 |
| | | | RWX |
| b) group access | 6 | ⇒ | 1 1 0 |
| | | | RWX |
| c) public access | 1 | ⇒ | 0 0 1 |

- Ask manager to create a group (unique name), say G, and add some users to the group.
- For a particular file (say *game*) or subdirectory, define an appropriate access.



Attach a group to a file

```
chgrp G game
```





Windows XP Access-control List Management

10.tex Properties

General Security Summary

Group or user names:

- Administrators (PBG-LAPTOP\Administrators)
- Guest (PBG-LAPTOP\Guest)**
- pbg (CTI\pbg)
- SYSTEM
- Users (PBG-LAPTOP\Users)

Add... Remove

| Permissions for Guest | Allow | Deny |
|-----------------------|--------------------------|-------------------------------------|
| Full Control | <input type="checkbox"/> | <input checked="" type="checkbox"/> |
| Modify | <input type="checkbox"/> | <input checked="" type="checkbox"/> |
| Read & Execute | <input type="checkbox"/> | <input checked="" type="checkbox"/> |
| Read | <input type="checkbox"/> | <input checked="" type="checkbox"/> |
| Write | <input type="checkbox"/> | <input checked="" type="checkbox"/> |
| Special Permissions | <input type="checkbox"/> | <input type="checkbox"/> |

For special permissions or for advanced settings, click Advanced.

Advanced

OK Cancel Apply



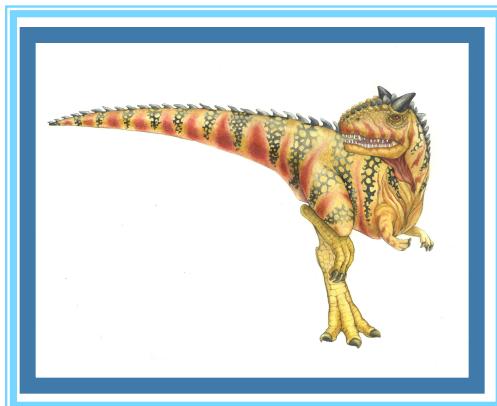


A Sample UNIX Directory Listing

| | | | | | | |
|------------|---|-----|---------|-------|--------------|---------------|
| -rw-rw-r-- | 1 | pbg | staff | 31200 | Sep 3 08:30 | intro.ps |
| drwx----- | 5 | pbg | staff | 512 | Jul 8 09:33 | private/ |
| drwxrwxr-x | 2 | pbg | staff | 512 | Jul 8 09:35 | doc/ |
| drwxrwx--- | 2 | pbg | student | 512 | Aug 3 14:13 | student-proj/ |
| -rw-r--r-- | 1 | pbg | staff | 9423 | Feb 24 2003 | program.c |
| -rwxr-xr-x | 1 | pbg | staff | 20471 | Feb 24 2003 | program |
| drwx--x--x | 4 | pbg | faculty | 512 | Jul 31 10:31 | lib/ |
| drwx----- | 3 | pbg | staff | 1024 | Aug 29 06:52 | mail/ |
| drwxrwxrwx | 3 | pbg | staff | 512 | Jul 8 09:35 | test/ |



End of Chapter 10



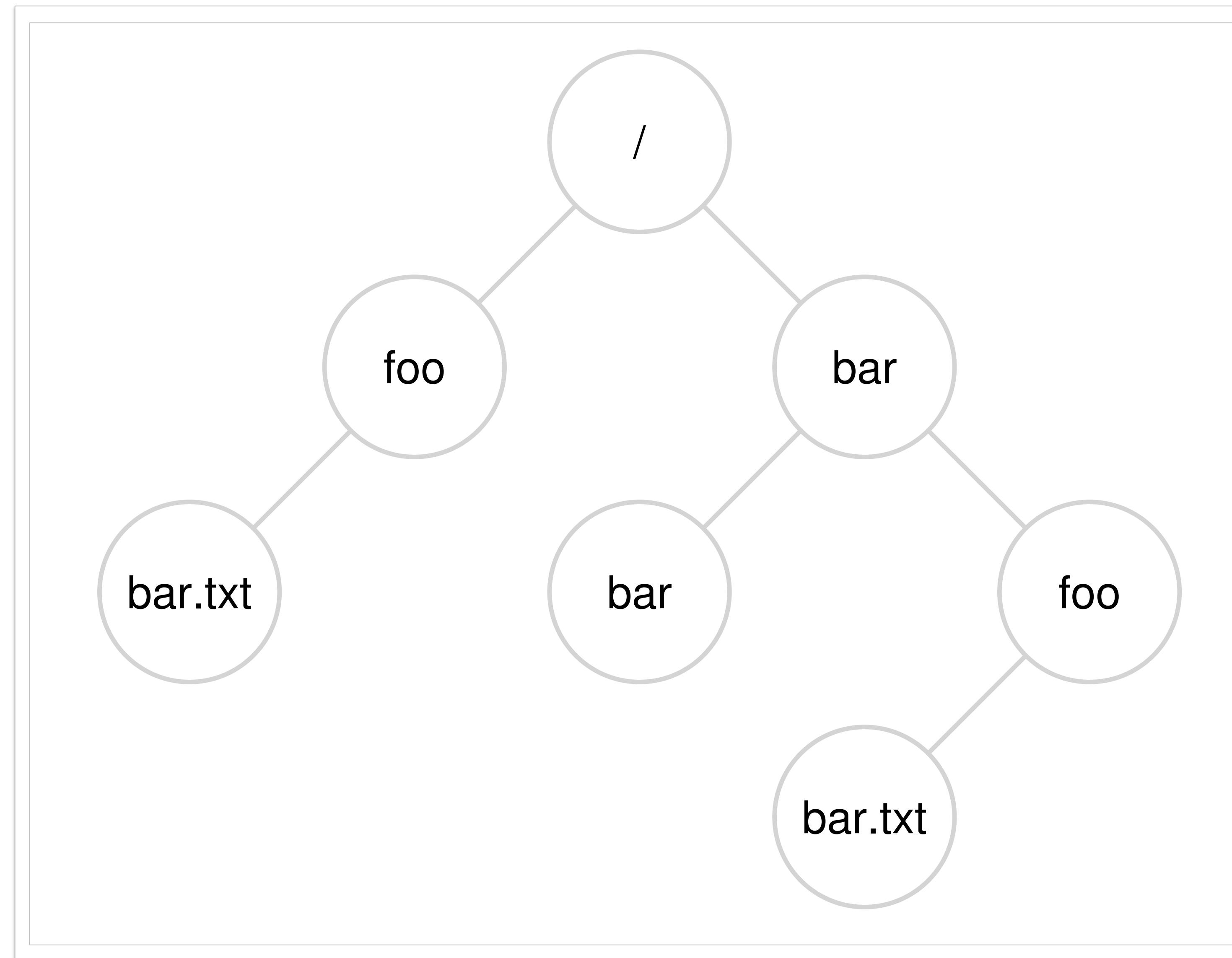
File and Directories

Virtualization of storage

Two key abstractions

- » File
- » Directory

Directory hierarchy



File system interface: Creating files

```
int fd = open("foo", O_CREAT | O_WRONLY | O_TRUNC);
```

File system interface: Reading and writing files

```
prompt> echo hello > foo
prompt> cat foo
hello
prompt>
```

File system interface: Reading and writing files

```
prompt> strace cat foo
...
open("foo", O_RDONLY|O_LARGEFILE)      = 3
read(3, "hello\n", 4096)               = 6
write(1, "hello\n", 6)                 = 6
hello
read(3, "", 4096)                     = 0
close(3)                             = 0
...
prompt>
```

File system interface: Writing immediately with `fsync()`

`write()`: Please, write this data to the disk at some point in the future. The data is written to a buffer, and later written to the persistent storage.

Data loss is rare. But it could happen if program crashes before the data is actually transferred to the disk.

File system interface: Writing immediately with `fsync()`

Application such as Data Base Management Systems need sometimes to force writes to disk.

```
fsync(int fd);
```

File system interface: Writing immediately with fsync()

```
int fd = open("foo", O_CREAT | O_WRONLY | O_TRUNC);  
assert(fd > -1);  
int rc = write(fd, buffer, size);  
assert(rc == size);  
rc = fsync(fd);  
assert(rc == 0);
```

File system interface: Renaming files

```
mv foo bar
```

`strace` shows that `mv` calls the system call `rename()`.

File system interface: Renaming files

- `mv foo bar`
- `strace` shows that `mv` calls the system call
`rename(char *old, char *new)`
- `rename()` is implemented as an atomic call.

File system interface: Renaming files

- **Example:** Editing a file using a text editor (e.g., emacs). Editor needs to guarantee that the new file has the original content as well as the recently updated text.

File system interface: Renaming files

```
int fd = open("foo.txt.tmp", O_WRONLY|O_CREAT|O_TRUNC);  
write(fd, buffer, size); // write out new version of file  
fsync(fd);  
close(fd);  
rename("foo.txt.tmp", "foo.txt");
```

File system interface: Removing files

- `rm foo`

File system interface: Removing files

- rm foo

```
prompt> strace rm foo
```

```
...
```

```
unlink("foo") = 0
```

```
...
```

File system interface: Creating directories

```
prompt> strace mkdir foo
...
mkdir("foo", 0777) = 0
...
prompt>
```

File system interface: Creating directories

```
prompt> strace mkdir foo  
...  
mkdir("foo", 0777) = 0  
...  
prompt>
```

```
prompt> ls -a  
./ .. /
```

File System Implementation



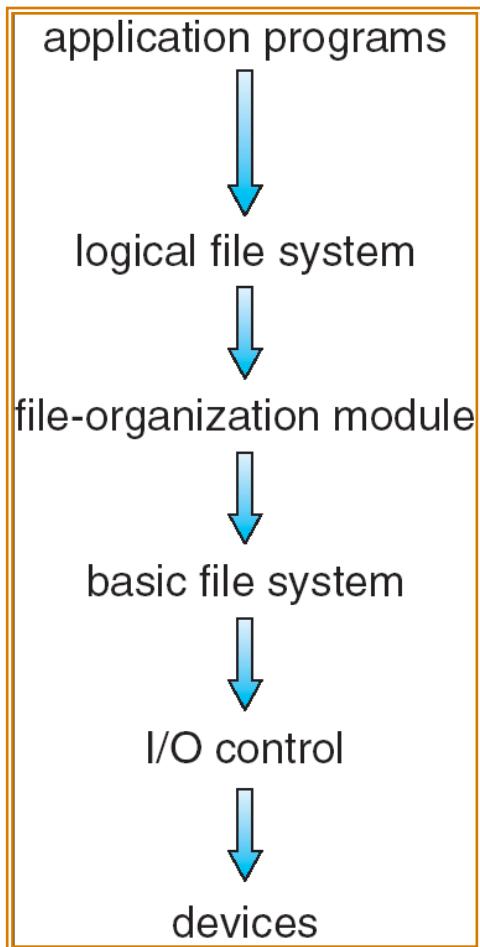
Chapter 11: File System Implementation

- File-System Structure
- File-System Implementation
- Directory Implementation
- Allocation Methods

File-System Structure

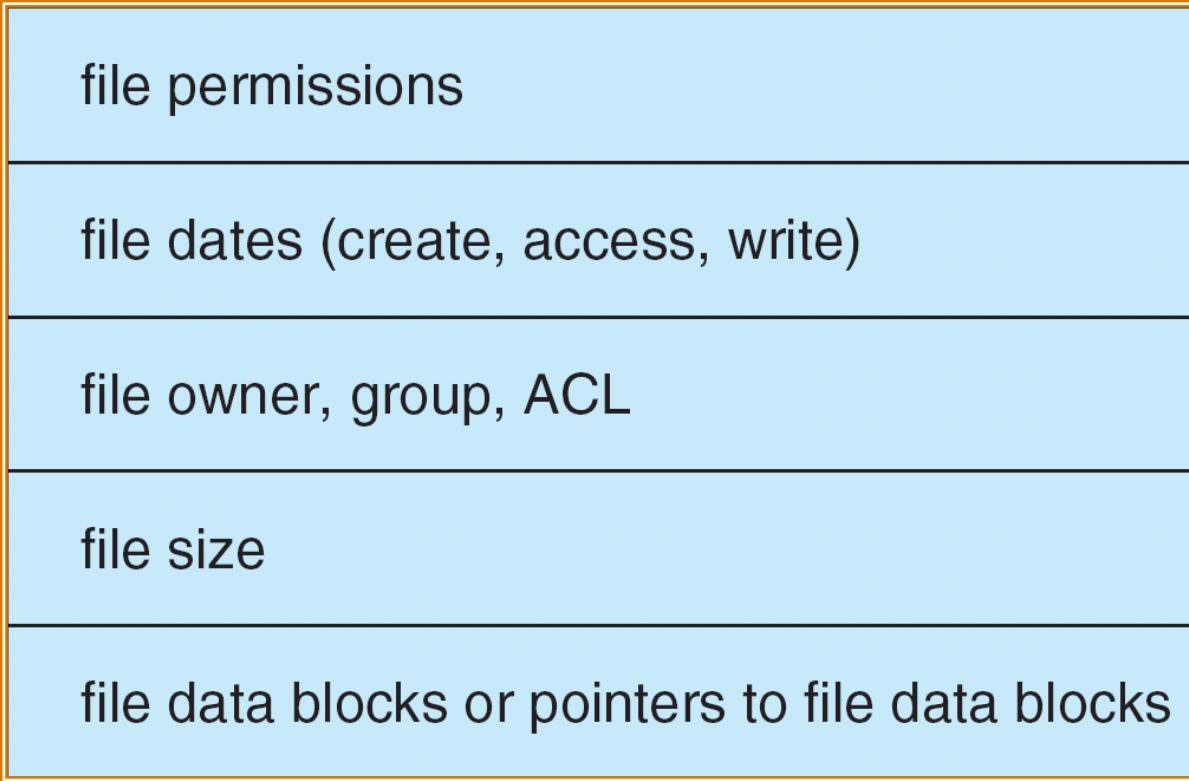
- File structure
 - Logical storage unit
 - Collection of related information
- File system resides on secondary storage (disks)
- File system organized into layers
- **File control block** – storage structure consisting of information about a file

Layered File System



- I/O Control: device drivers and interrupt handlers that talk to the disk
- Basic file system: Generic block reads and writes
 - e.g., read cylinder 73, track 2, sector 10
- File organization: Files and logical blocks
 - Translate logical blocks to physical
 - Manage free space
- Logical file system: Metadata information
 - e.g., owner, permissions, size, etc.

A Typical File Control Block



- FCB has all meta-information about a file
 - Linux calls these *i-nodes*

Implementing File Operations (1)

- **Create a file:**

- Find space in the file system, add directory entry.

- **Open a file:**

- System call specifying name of file.
 - System searches directory structure to find file.
 - System keeps ***current file position pointer*** to the location where next write/read occurs
 - System call returns ***file descriptor*** (a handle) to user process

- **Reading a file:**

- System call specifying file descriptor and number of bytes to read (and possibly where in memory to stick contents).

Implementing File Operations (2)

□ Writing in a file:

- System call specifying file descriptor and information to be written
- Writes information at location pointed by the files current pointer

□ Repositioning within a file:

- System call specifying file descriptor and new location of current pointer
- (also called a file **seek** even though does not interact with disk)

□ Closing a file:

- System call specifying file descriptor
- Call removes current file position pointer and file descriptor associated with process and file

□ Deleting a file:

- Search directory structure for named file, release associated file space and erase directory entry

□ Truncating a file:

- Keep attributes the same, but reset file size to 0, and reclaim file space.

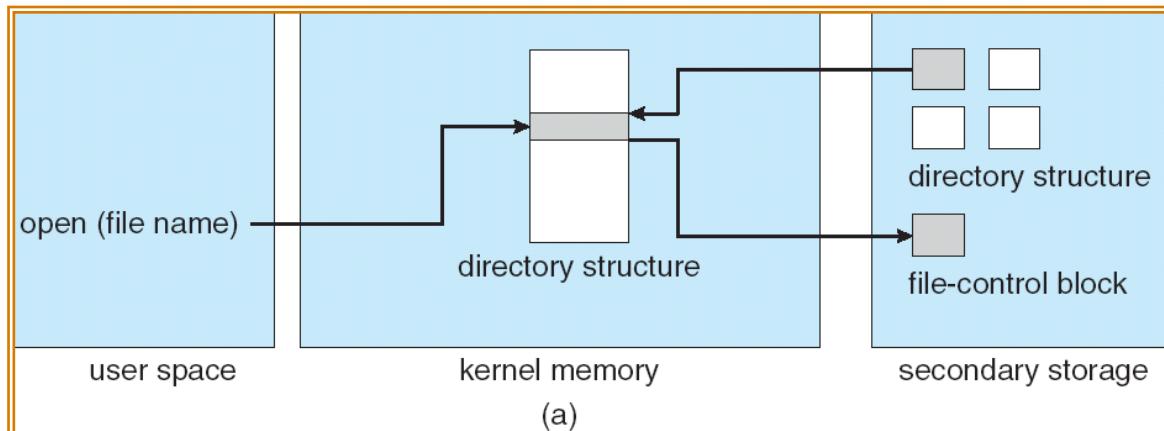
Other File Operations

- Most FS require an open() system call before using a file.
- OS keeps an in-memory table of open files, so when reading or writing is requested, they refer to entries in this table via a file descriptor.
- On finishing with a file, a close() system call is necessary. (creating & deleting files typically works on closed files)

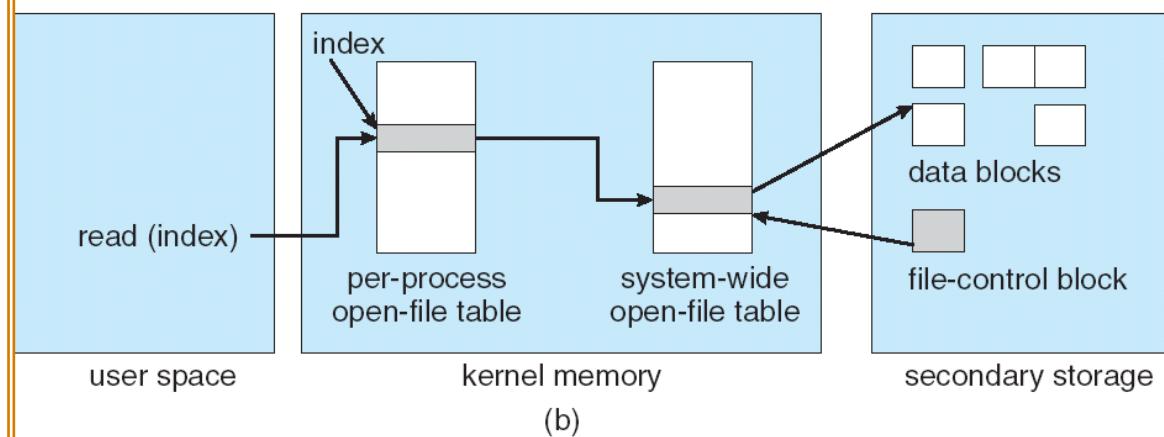
Multiple Users of a File

- OS typically keeps two levels of internal tables:
- **Per-process table**
 - Information about the use of the file by the user (e.g. current file position pointer)
- **System-wide table**
 - Gets created by first process which opens the file
 - Location of file on disk
 - Access dates
 - File size
 - Count of how many processes have the file open (used for deletion)

In-Memory File System Structures



opening a file

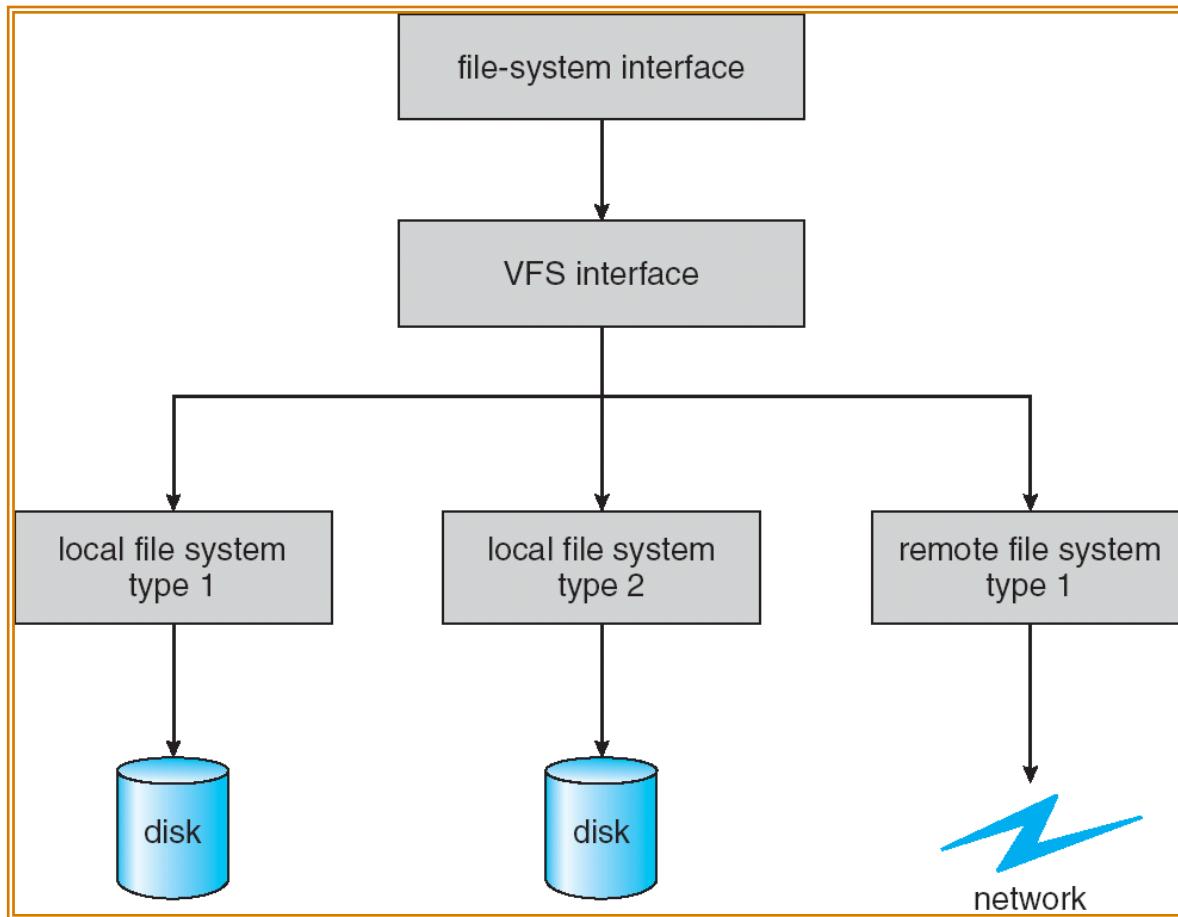


reading a file

Virtual File Systems

- Virtual File Systems (VFS) provide an object-oriented way of implementing file systems.
- VFS allows the same system call interface (the API) to be used for different types of file systems.
- The API is to the VFS interface, rather than any specific type of file system.

Schematic View of Virtual File System



Allocating and Storing Files

- An allocation method refers to how disk blocks are allocated for files:

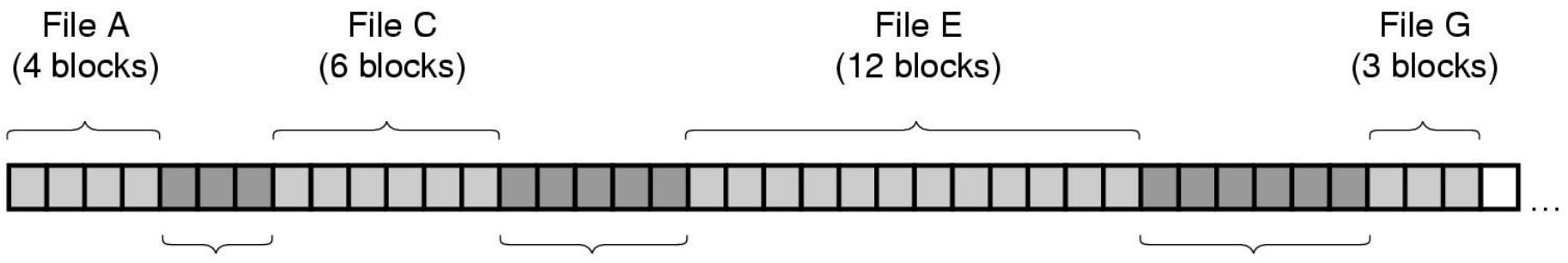
- **Contiguous allocation**
 - All bytes together, in order

- **Linked allocation**
 - Each block points to the next block

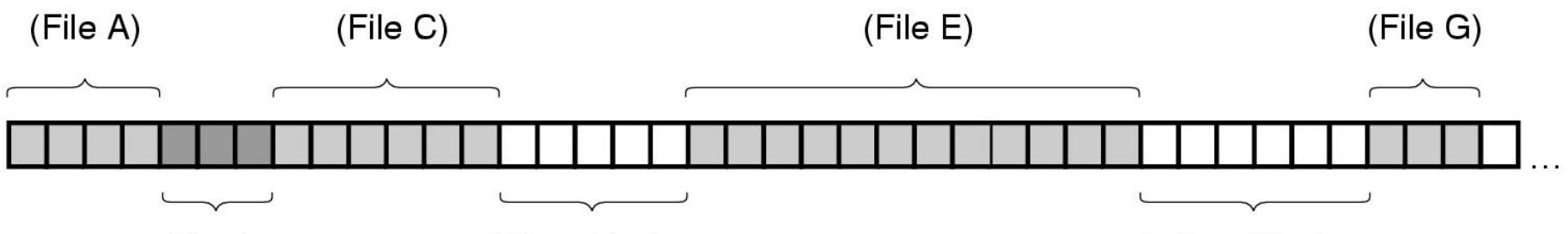
- **Indexed allocation**
 - An index block contains pointers to many other blocks

Contiguous Allocation

- Allocate files contiguously on disk

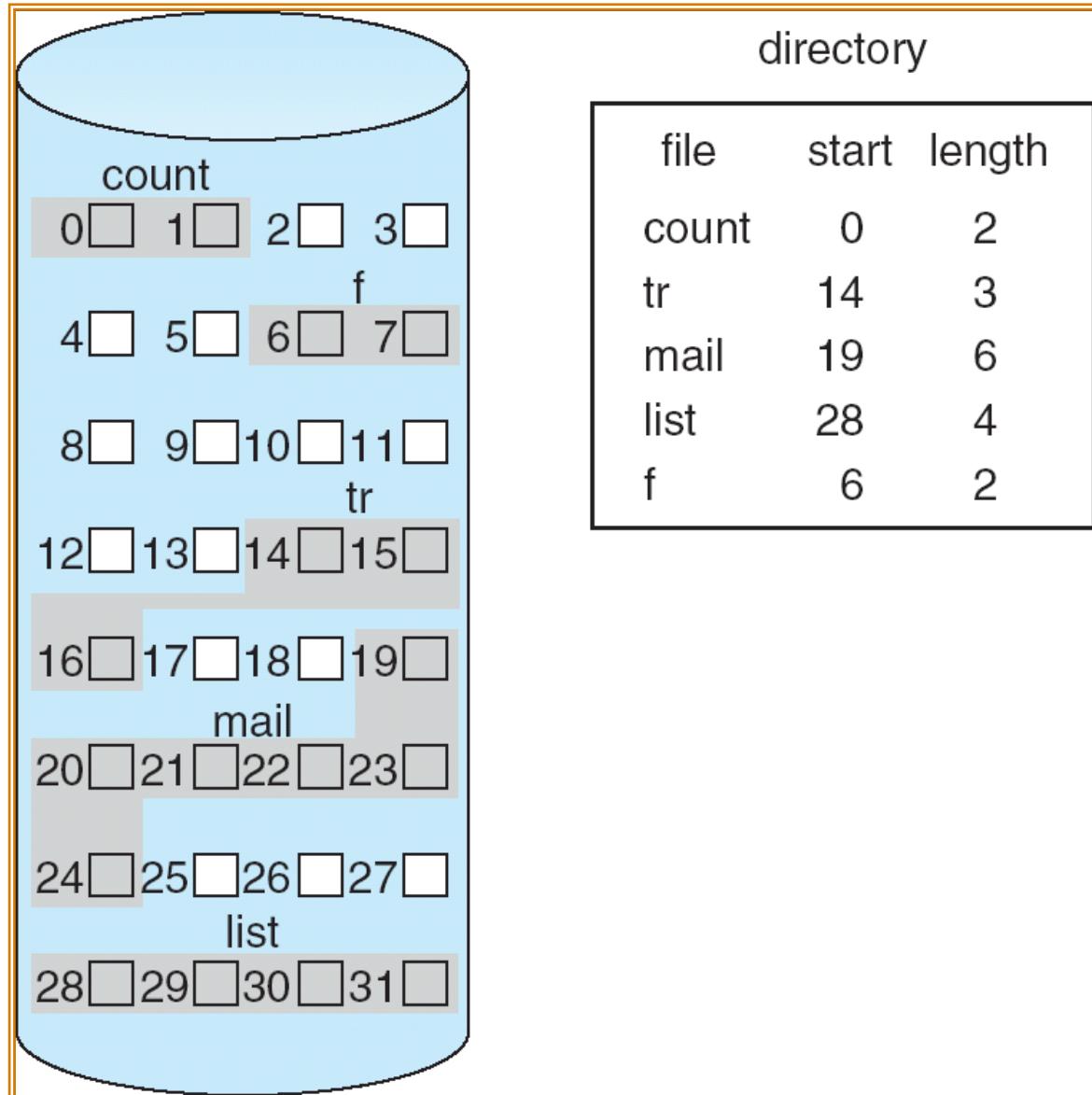


(a)



(b)

Contiguous Allocation of Disk Space



Contiguous Allocation

- Pros:

- Simple: state required per file is start block and size
- Performance: entire file can be read with one seek

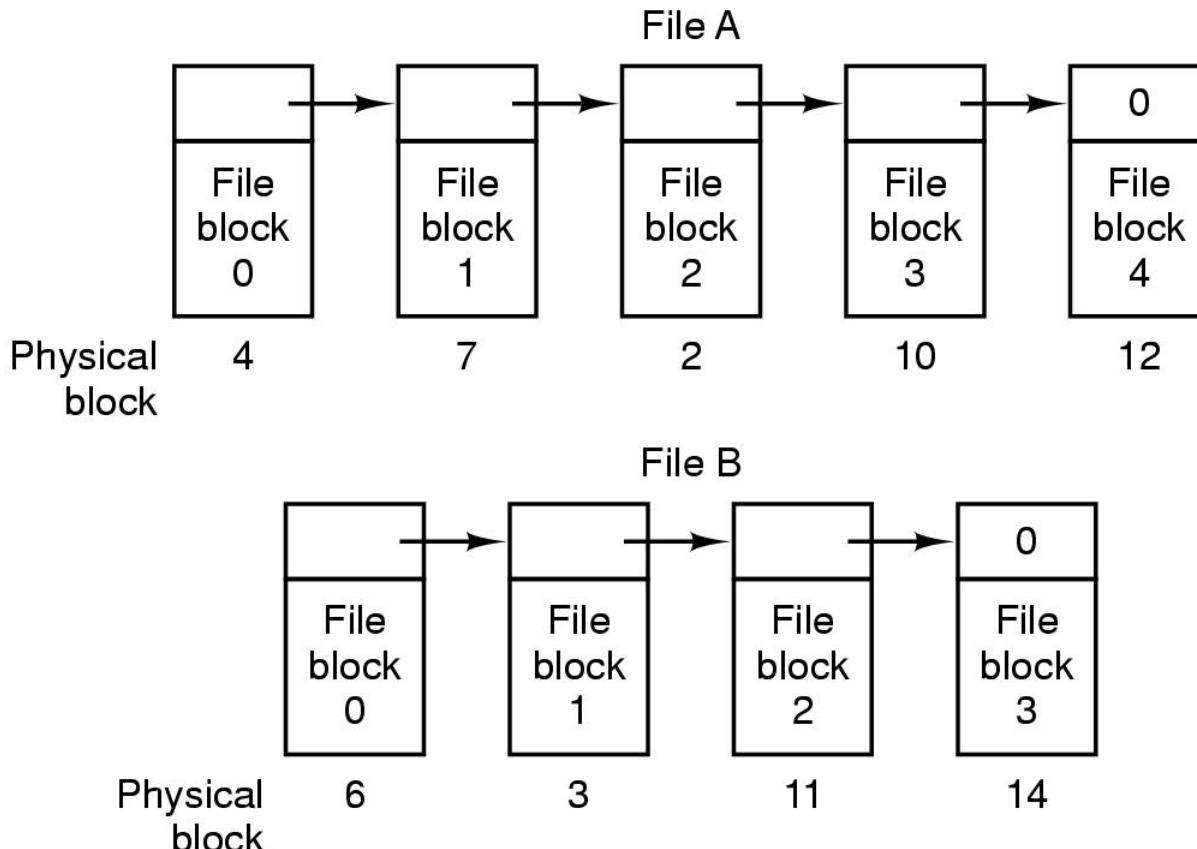
- Cons:

- Files can't grow
- Fragmentation: external frag is bigger problem
- Wastes space

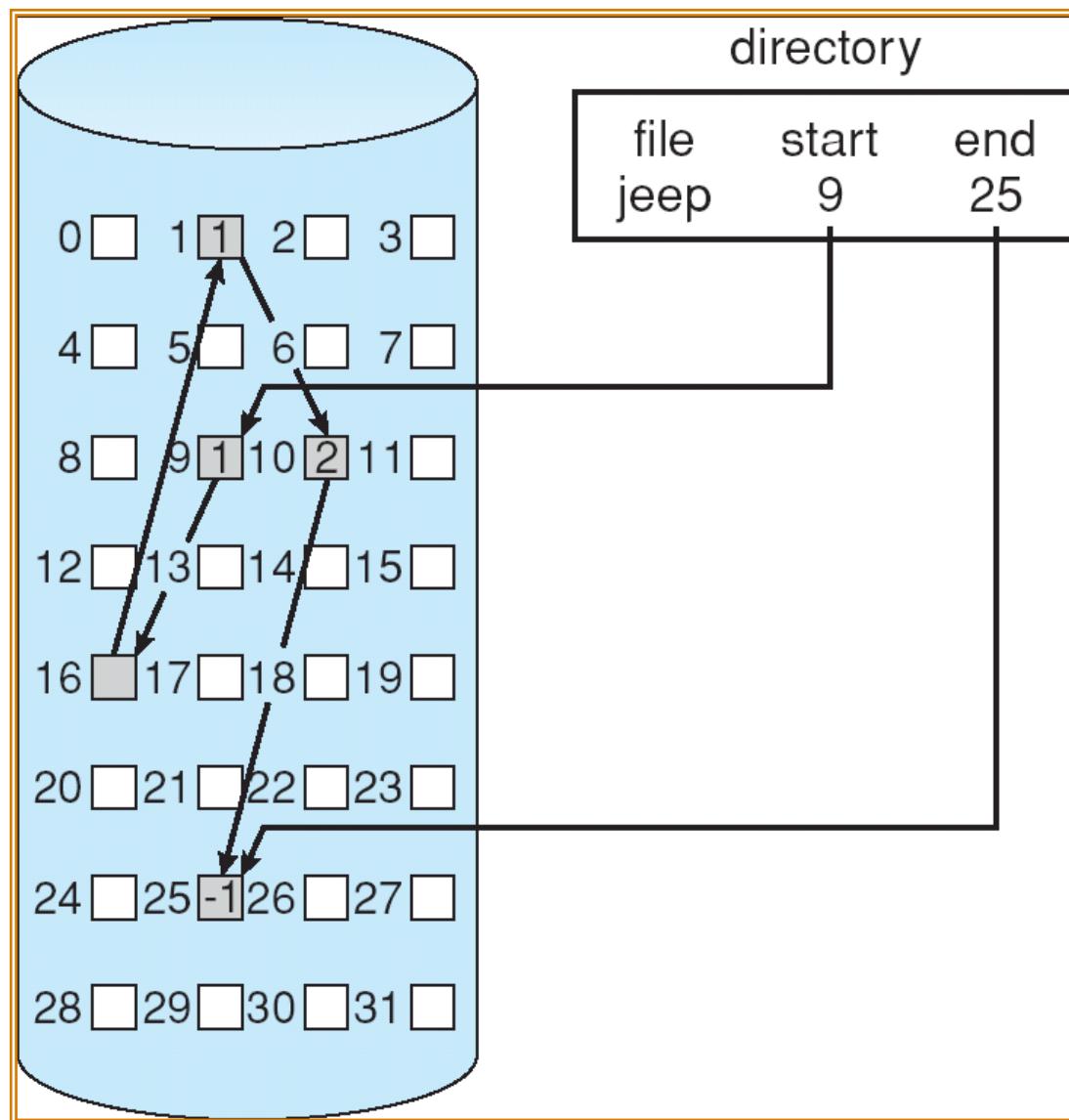
- Used in CDROMs, DVDs

Linked List Allocation

- Each file is stored as linked list of blocks
 - First word of each block points to next block
 - Rest of disk block is file data



Linked Allocation



Linked List Allocation

- **Pros:**

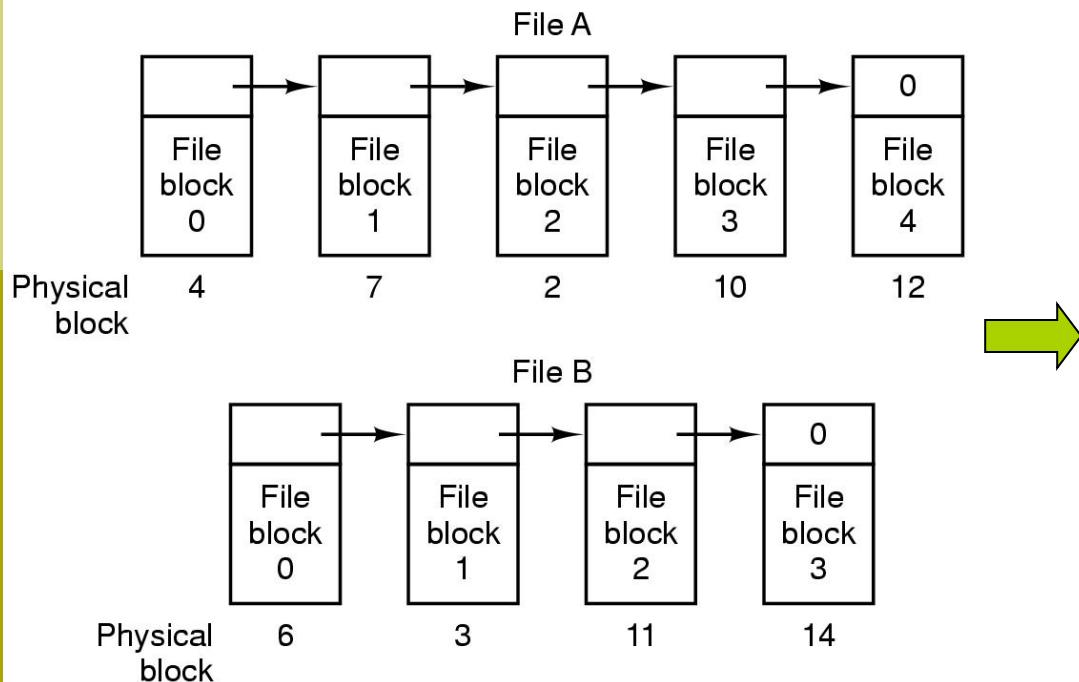
- No space lost to external fragmentation
- Disk only needs to maintain first block of each file

- **Cons:**

- Random access is costly
- Overheads of pointers

Example: MS-DOS File System

- Implement a linked list allocation using a table
 - Called File Allocation Table (FAT)
 - Take pointer away from blocks, store in this table
 - Can cache FAT in-memory



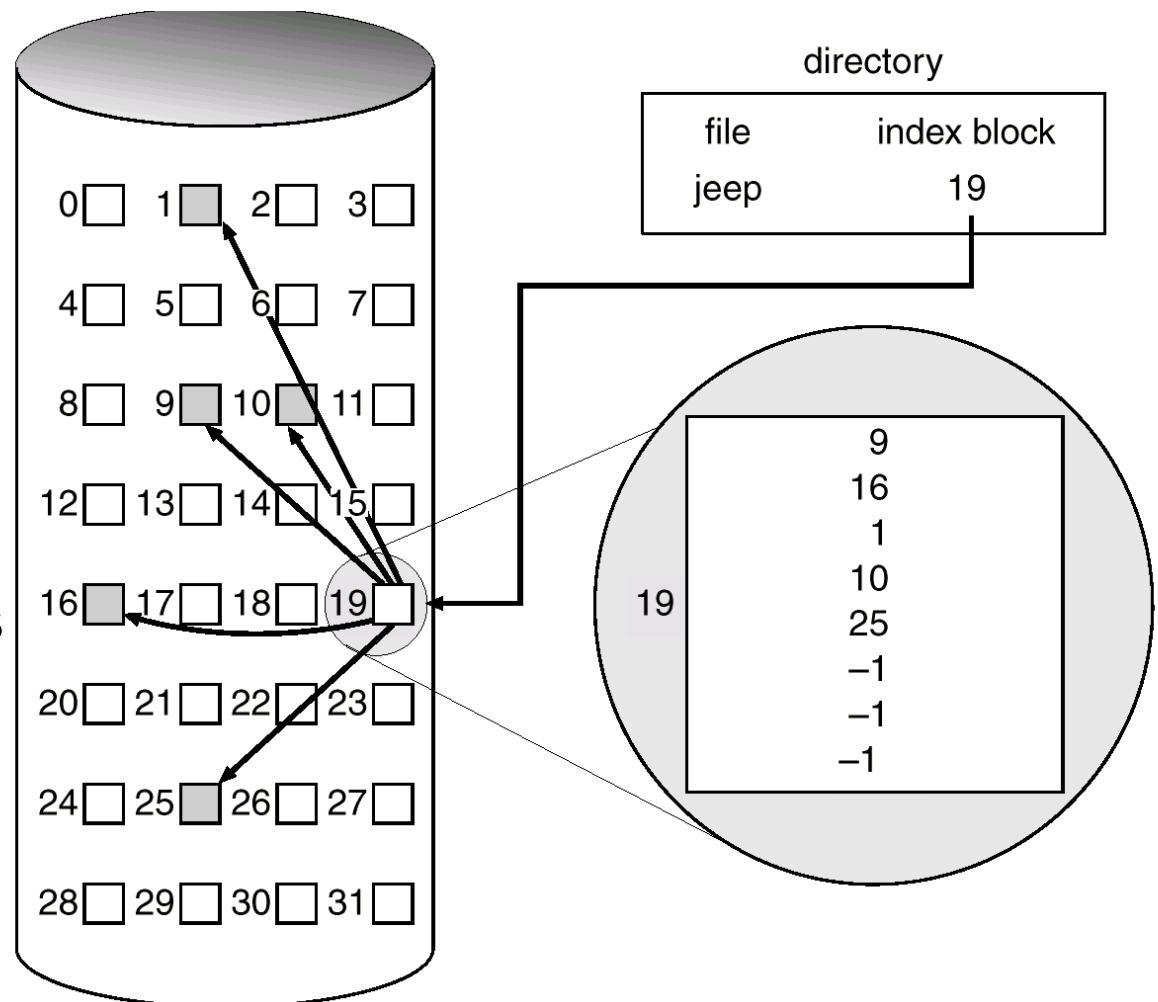
| Physical block |
|----------------|
| 0 |
| 1 |
| 2 |
| 3 |
| 4 |
| 5 |
| 6 |
| 7 |
| 8 |
| 9 |
| 10 |
| 11 |
| 12 |
| 13 |
| 14 |
| 15 |

Annotations on the right side of the table:

- File A starts here (points to block 2)
- File B starts here (points to block 6)
- Unused block (points to block 15)

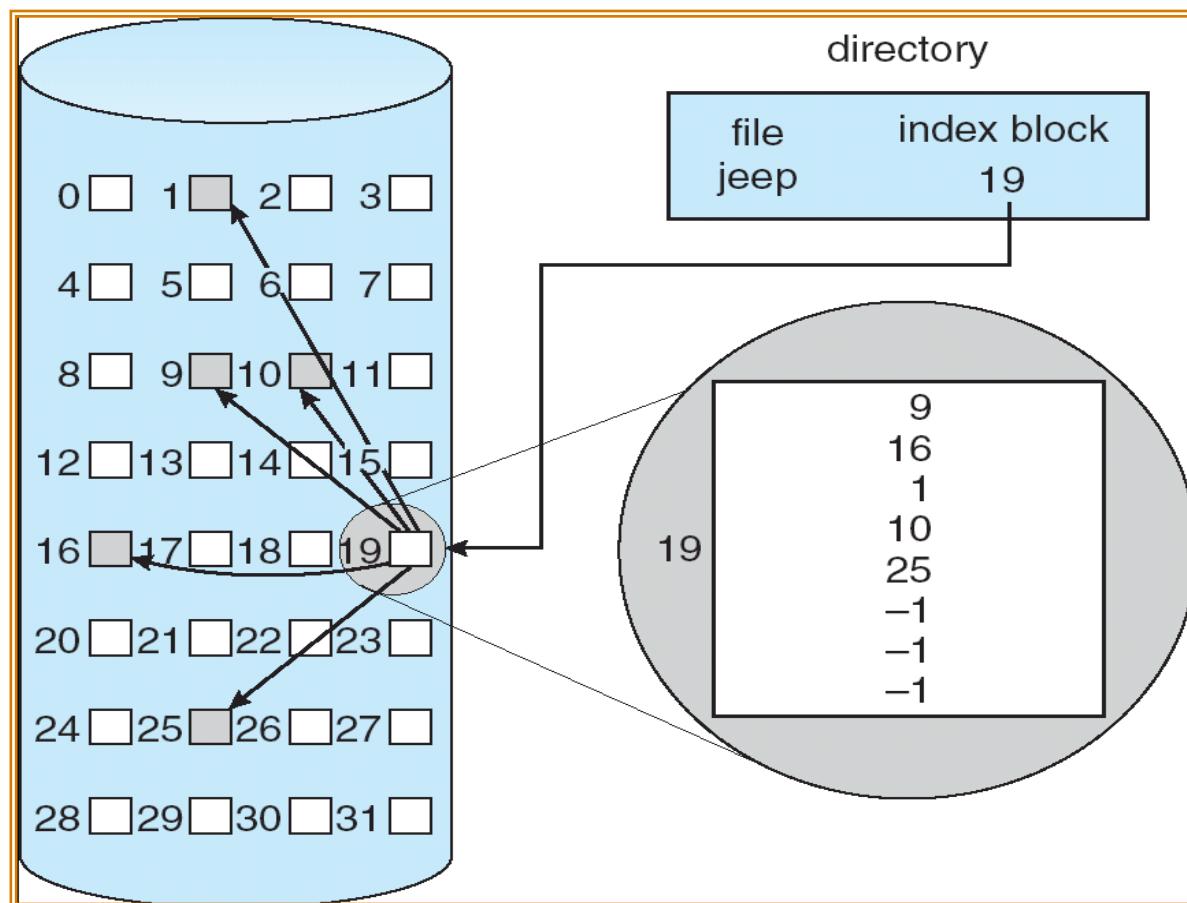
Indexed Allocation

- Index block contains pointers to each data block
- Pros?
 - Space (max open files * size per I-node)
- Cons?
 - what if file expands beyond I-node address space?



Indexed Allocation

- Brings all pointers to data blocks together into an *index block*.



Implementing Directories

- ❑ Directory: map ASCII file name to file attributes & location
- ❑ When a file is opened, OS uses path name to find dir
 - Directory has information about the file's disk blocks
 - ❑ Whole file (contiguous), first block (linked-list) or I-node (indexed allocation)
 - Directory also has attributes of each file
- ❑ 2 options: entries have all attributes, or point to file I-node

| | |
|-------|------------|
| games | attributes |
| mail | attributes |
| news | attributes |
| work | attributes |

(a)

