ZHEJIANG UNIVERSITY

Review 01

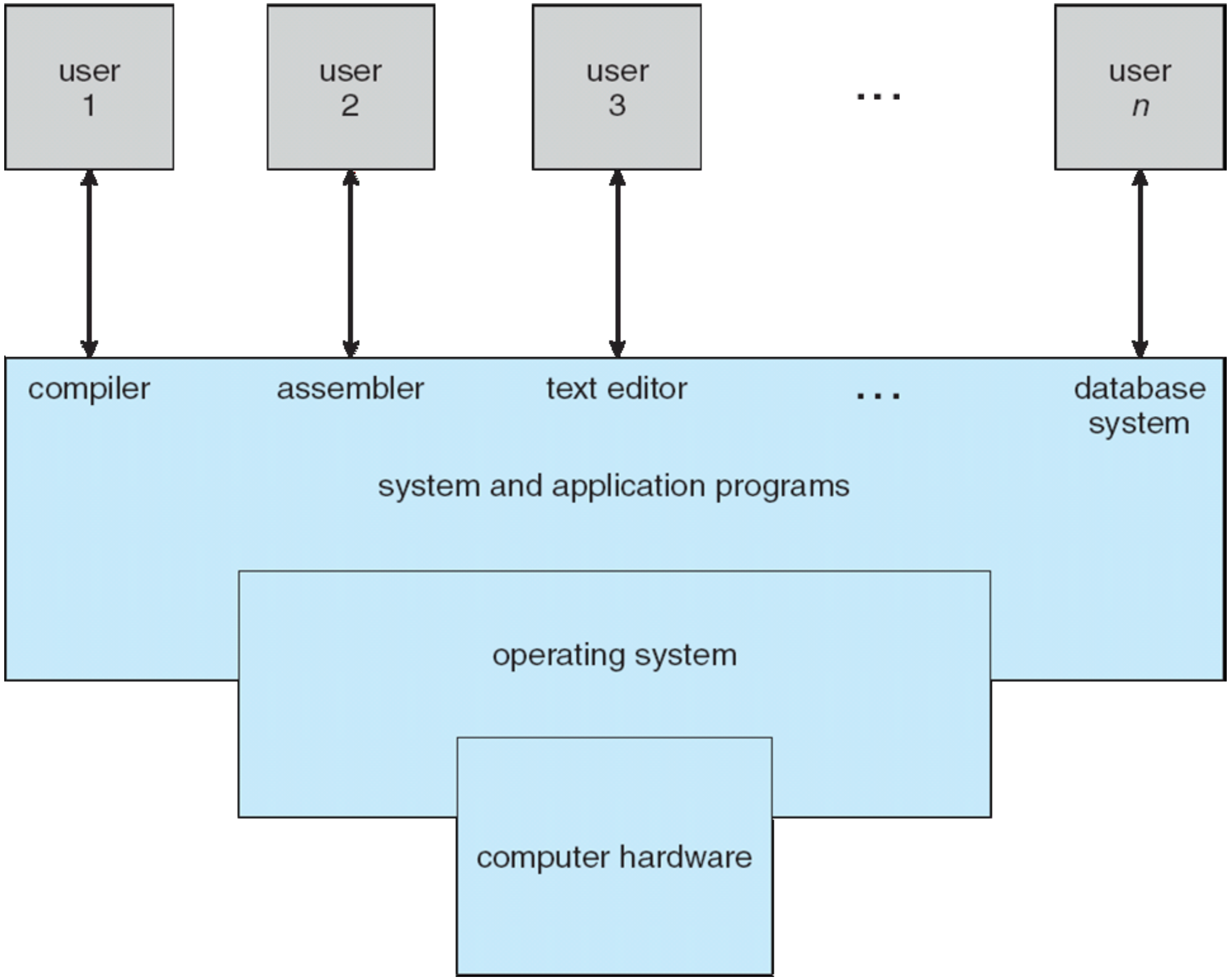
Yajin Zhou (http://yajin.org)

Zhejiang University

Acknowledgement: Some slides are based on Zhi Wang (FSU)'s slides.

01: Introduction

Four Components of a Computer System



Interrupts and Traps

- Interrupt transfers control to the interrupt service routine
  - **interrupt vector:** a table containing addresses of all the service routines
  - incoming interrupts are disabled while serving another interrupt to prevent a lost interrupt
  - **interrupt handler** must save the (interrupted) execution states
- A **trap** is a **software-generated interrupt**, caused either by an error or a user request
  - an **interrupt** is asynchronous; a **trap** is synchronous
  - e.g., **system call**, divided-by-zero exception, general protection exception…
    - Int 0x80 - this is **not** a privileged instruction
- Operating systems are usually **interrupt-driven**

Interrupt Handling

- Operating system preserves the execution state of the CPU
  - save registers and the program counter (PC)
- OS determines which device caused the interrupt
  - **polling**
  - **vectored** interrupt system
- OS handles the interrupt by calling the device's driver
- OS restores the CPU execution to the saved state

I/O: from System Call to Devices, and Back

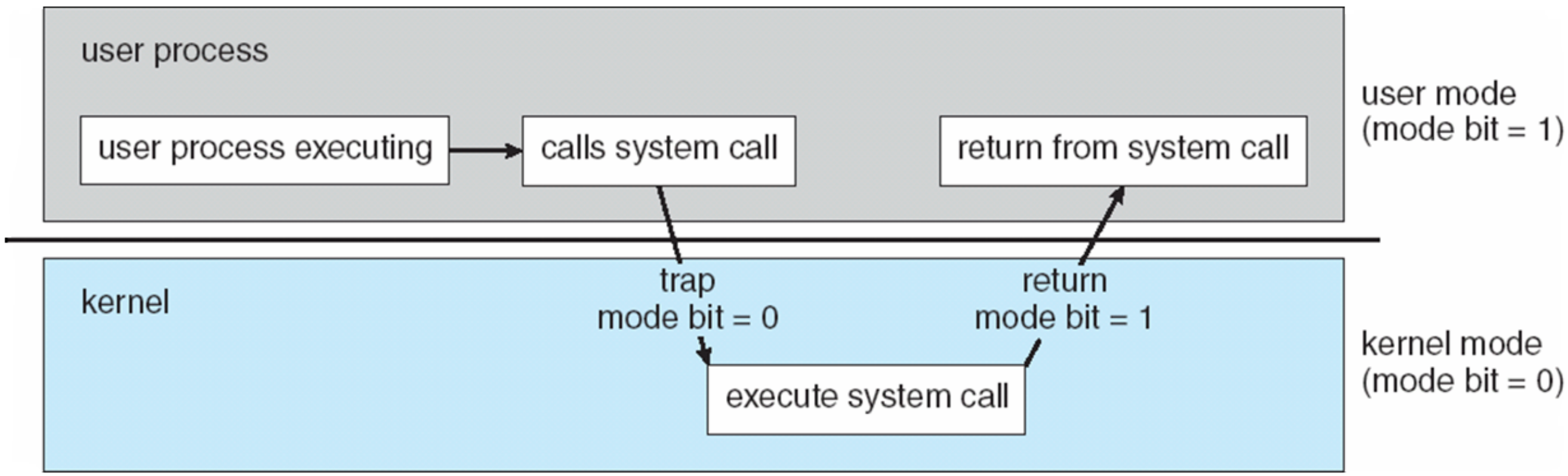
- A program uses a **system call** to access system resources
  - e.g., files, network
- Operating system converts it to device access and issues I/O requests
  - I/O requests are sent to the device driver, then to the controller
  - e.g., read disk blocks, send/receive packets…
- OS puts the program to wait (**synchronous I/O**) or returns to it without waiting (**asynchronous I/O**)
  - OS may switches to another program when the requester is waiting
- I/O completes and the controller **interrupts** the OS
- OS processes the I/O, and then wakes up the program (synchronous I/O) or send its a signal (asynchronous I/O)

Dual-mode operation

- Operating system is usually interrupt-driven (why?)
  - Efficiency, regain control (timer interrupt)
- **Dual-mode operation** allows OS to protect itself and other system components
  - **user mode** and **kernel mode (or other names)**
  - a **mode** bit distinguishes when CPU is running user code or kernel code
  - some instructions designated as **privileged**, **only executable in kernel and cannot executed in user mode (and certain memory cannot be accessed in user mode!)**
  - **system call** changes mode to kernel, return from call resets it to user

Transition between Modes

- **System calls, exceptions, interrupts** cause transitions between kernel/user modes



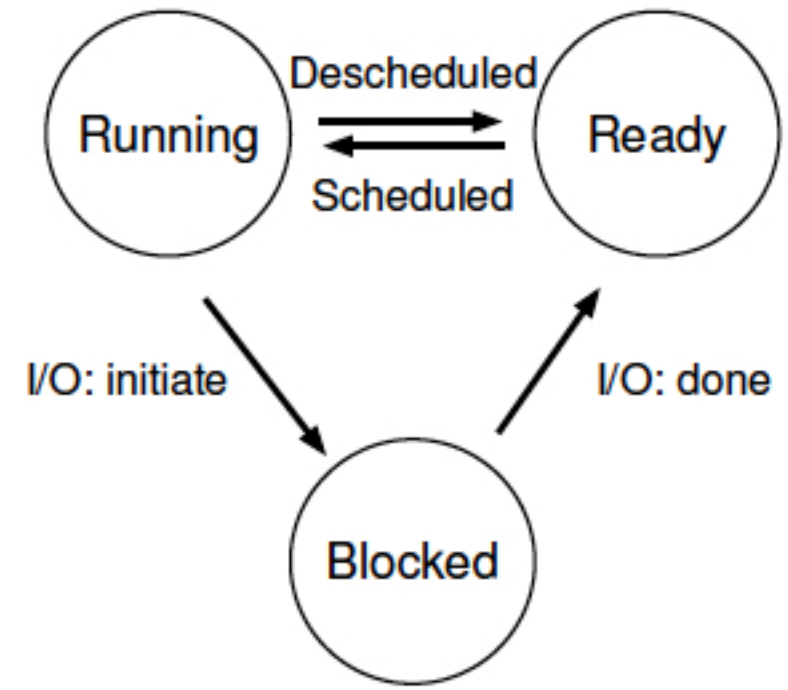
In user mode, certain instructions cannot be executed

Resource Management: Process Management

- A process is **a program in execution**
  - program is a ***passive*** entity, process is an ***active*** entity
  - a system has many processes running concurrently
- Process needs resources to accomplish its task
  - OS reclaims all reusable resources upon process termination
  - e.g., CPU, memory, I/O, files, initialization data

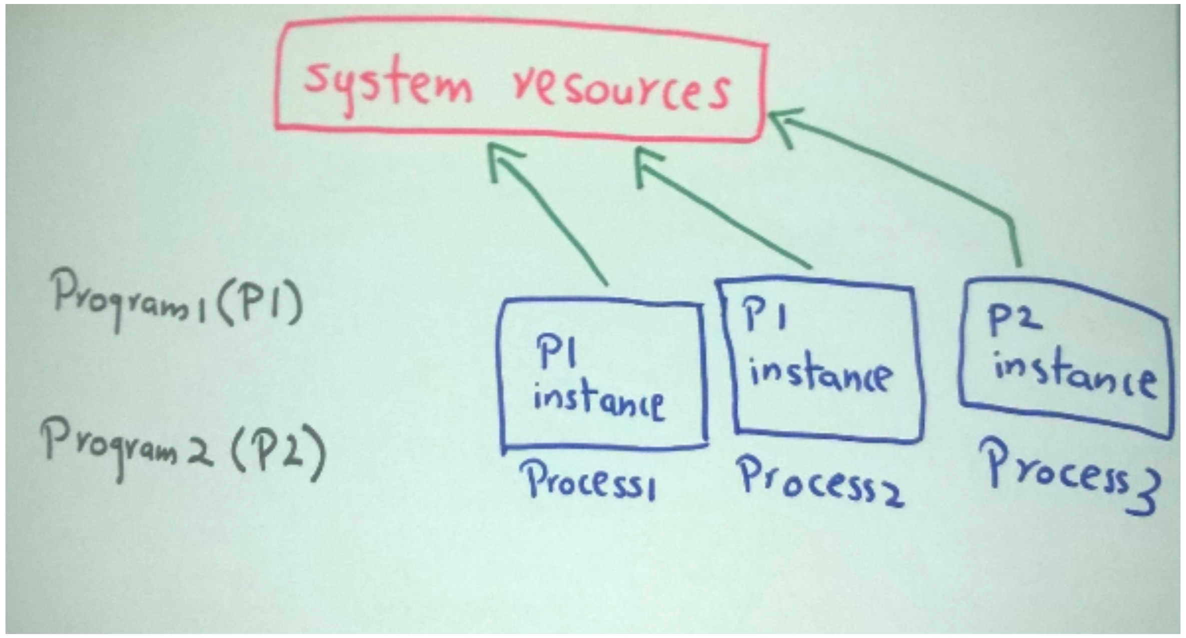
Process Management Activities

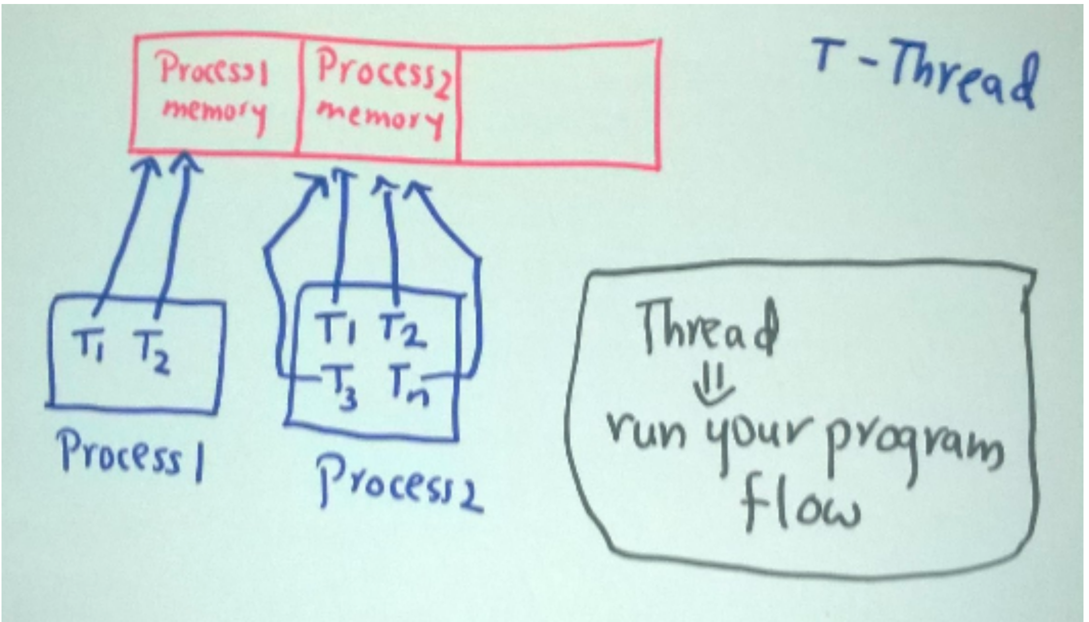
- Process creation and termination
- Processes suspension and resumption
- Process synchronization primitives
- Process communication primitives
- Deadlock handling



From Process to Thread

- Single-threaded process has one program counter
  - **program counter** specifies *location of next instruction to execute*
  - processor executes instructions sequentially, one at a time, until completion
- Multi-threaded process has **one program counter per thread**
- **Process is the unit of resource allocation and protection, not thread!**





Figures: https://medium.com/coderscorner/threads-vs-processes-a0b199f56c2d

Resource Management: Memory Management

- Memory is the main storage directly accessible to CPU
  - data needs to be kept in memory before and after processing
  - all instructions should be in memory in order to execute
- Memory management determines what is in memory to **optimize CPU utilization** and **response time, provides a virtual view of memory for programmer**
- Memory management activities:
  - keeping track of which parts of memory are being used and by whom
  - deciding which processes and data to move into and out of memory
  - allocating and deallocating memory space as needed

Resource Management: File Systems

- OS provides a uniform, logical view of data storage
  - **file** is a logical storage unit that abstracts physical properties
    - files are usually organized into **directories**
    - **access control** determines who can access the file
- File system management activities:
  - creating and deleting files and directories
  - primitives to manipulate files and directories
  - mapping files onto secondary storage
  - backup files onto stable (non-volatile) storage media

A special FS: /proc file system

02: Operating System Services & Structures

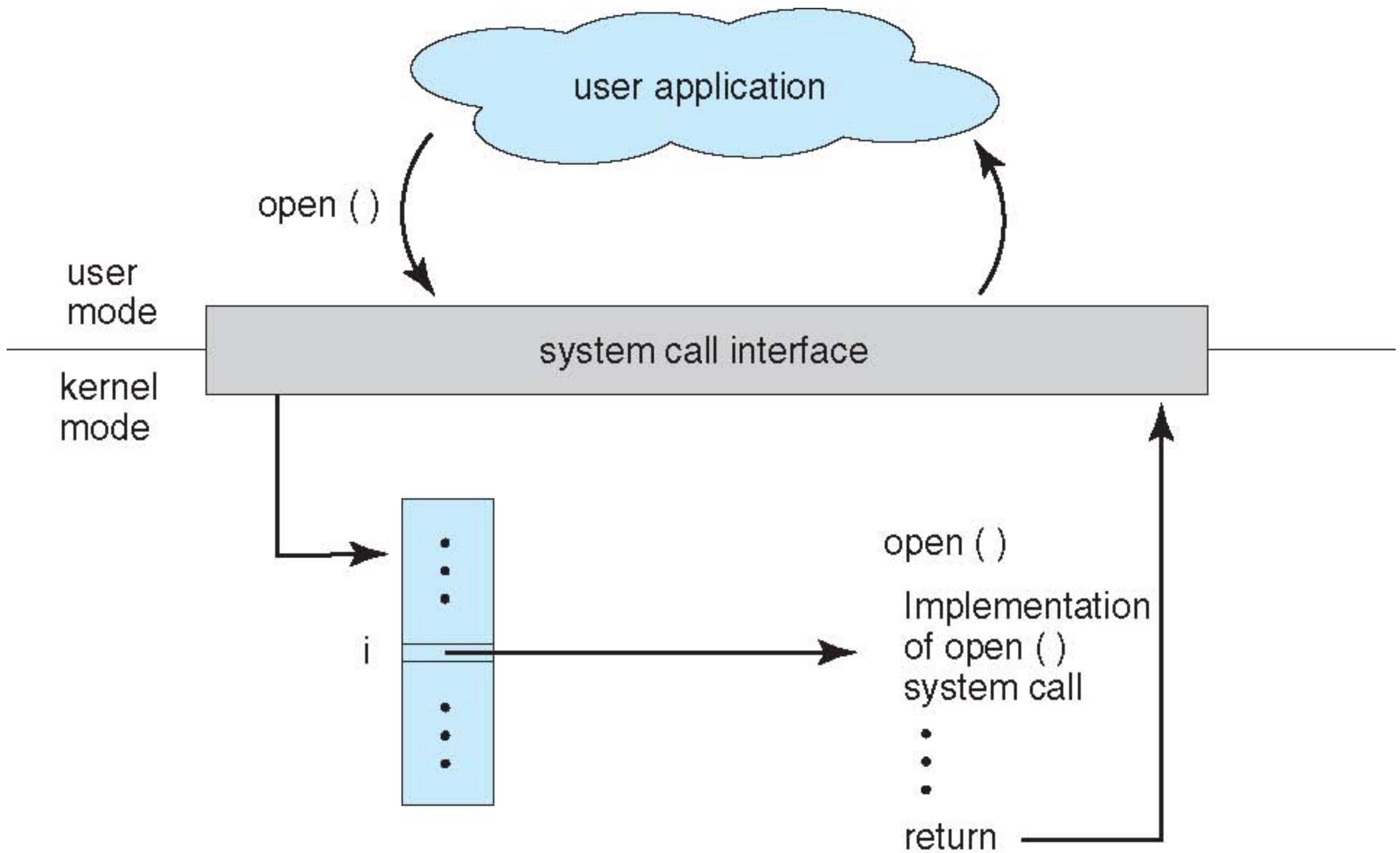
User Interface

- GUI
- Shell
  - How to chain multiple commands: cmd1;cmd2;cmd3
  - cat
  - Find . -name test -print
  - Mv
  - Pipe: cat xxx | grep xxx

System Calls

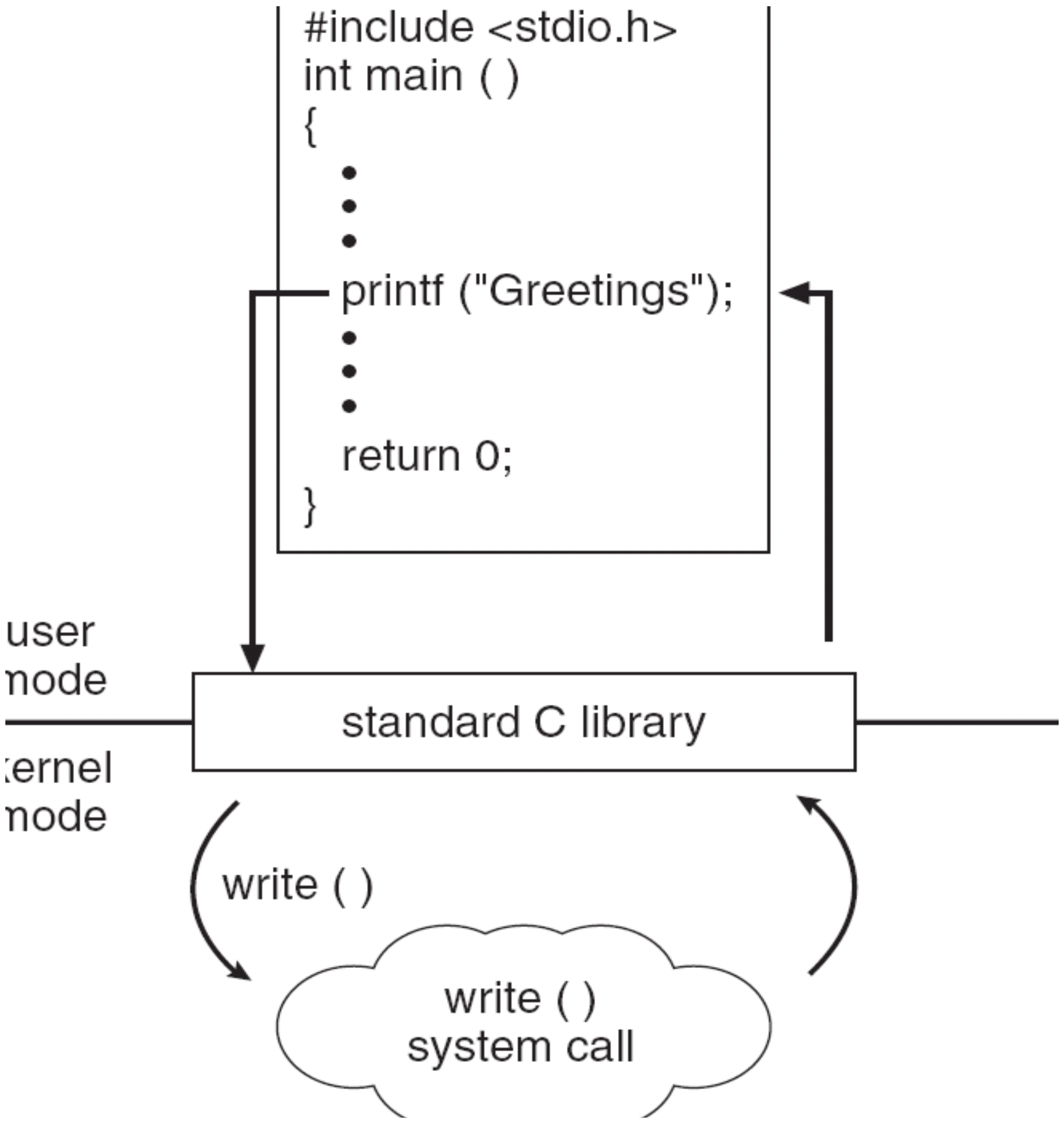
- System call is **a programming interface to access the OS services**
  - Typically written in a high-level language (C or C++)
  - Certain low level tasks are in assembly languages

API – System Call – OS Relationship



Standard C Library Example

- C program invoking printf() library call, which calls write() system call



System Call Parameter Passing

- Parameters are required besides the **system call number**
  - exact type and amount of information vary according to OS and call
- Three general methods to pass parameters to the OS
  - **Register**:
    - pass the parameters in registers
    - simple, but there may be more parameters than registers
  - **Block**:
    - parameters stored in a memory block (or table)
    - address of the block passed as a parameter in a register
    - taken by Linux and Solaris
  - **Stack**:
    - parameters placed, or pushed, onto the stack by the program
    - popped off the stack by the operating system
  - Block and stack methods don't limit number of parameters being passed

Operating System Design and Implementation

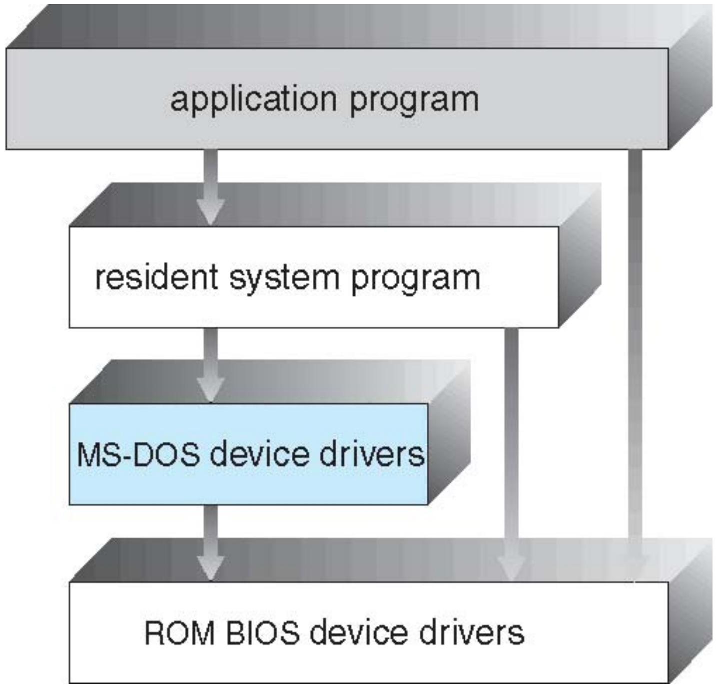
- Much variation
  - Early OSes in **assembly language**
  - Then system programming languages like Algol, PL/1
  - Now C, C++
- Actually usually a mix of languages
  - Lowest levels in **assembly**
  - Main body in C
  - Systems programs in C, C++, scripting languages like PERL, Python, shell scripts
- More high-level language easier to port to other hardware
  - But slower

Operating System Structure

- Many structures:
  - **simple structure - MS-DOS**
  - **more complex -- UNIX**
  - **layered structure - an abstraction**
  - **microkernel system structure - L4**
  - **hybrid: Mach, Minix**
  - research system: **exokernel**

Simple Structure: MS-DOS

- No structure at all!: (1981~1994)
  - written to provide the most functionality in the least space
- A typical example: MS-DOS
  - Has some structures:
    - its interfaces and levels of functionality are not well separated
  - the kernel is not divided into modules



Monolithic Structure – Original UNIX

- Limited by hardware functionality, the original UNIX had limited structure
- UNIX OS consists of two separable layers
  - **systems programs**
  - **the kernel**: everything below the system-call interface and above physical hardware
    - a large number of functions for one level: file systems, CPU scheduling, memory management …

Microkernel System Structure

- Microkernel moves as much from the kernel (e.g., file systems) into “user” space
- Communication between user modules uses **message passing**
- Benefits:
  - easier to extend a microkernel
  - **easier to port the operating system to new architectures: code base is small**
  - **more reliable: since less code is running in kernel mode**
  - **more secure: most code is running in unprivileged mode, small code base**
- Detriments:
  - performance overhead of user space to kernel space communication
- Examples: Minix, Mach, QNX, L4…

03: Process

Process Concept

- Process is **a program in execution**, its execution must progress in sequential fashion
  - a program is static and passive, process is dynamic and active
  - **one program can be several processes** (e.g., **multiple instances** of browser, or **even on instance** of the program)
  - process can be started via GUI or command line entry of its name
    - through system calls
- **Process is the basic unit for resource allocation and protection**

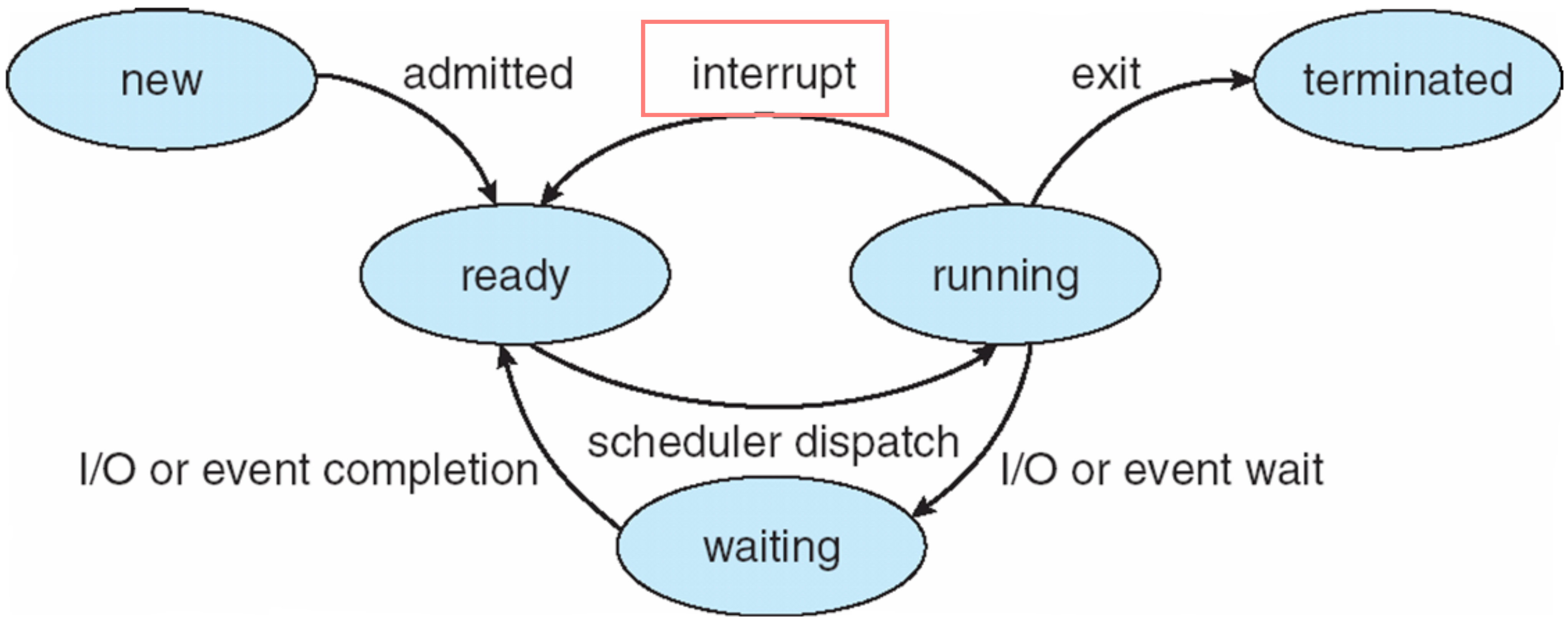
Process Concept

- A process has multiple parts:
  - the program **code**, also called **text section**
  - runtime **CPU states**, including program counter, registers, etc
  - various types of memory:
    - **stack**: temporary data
      - e.g., function parameters, local variables, and *return addresses*
    - **data** section: global variables
    - **heap**: memory dynamically allocated during runtime

Process State

- As a process executes, it changes state
  - **new**: the process is being created
  - **running**: instructions are being executed
  - **waiting/blocking**: the process is waiting for some event to occur
  - **ready**: the process is waiting to be assigned to a processor
  - **terminated**: the process has finished execution

Diagram of Process State



Process Control Block (PCB)

- In the kernel, each process is associated with a **process control block**
  - process number (pid)
  - **process state**
  - **program counter (PC)**
  - CPU registers
  - CPU scheduling information
  - memory-management data
  - accounting data
  - I/O status

```c
struct task_struct {
#ifdef CONFIG_THREAD_INFO_IN_TASK
	/*
	 * For reasons of header soup (see current_thread_info()), this
	 * must be the first element of task_struct.
	 */
	struct thread_info		thread_info;
#endif
	/* -1 unrunnable, 0 runnable, >0 stopped: */
	volatile long			state;

	/*
	 * This begins the randomizable portion of task_struct. Only
	 * scheduling-critical items should be added above here.
	 */
	randomized_struct_fields_start

	void				*stack;
	atomic_t			usage;
	/* Per task flags (PF_*), defined further below: */
	unsigned int			flags;
	unsigned int			ptrace;
```

- Linux's PCB is defined in struct task_struct: http://lxr.linux.no/linux+v3.2.35/include/linux/sched.h#L1221

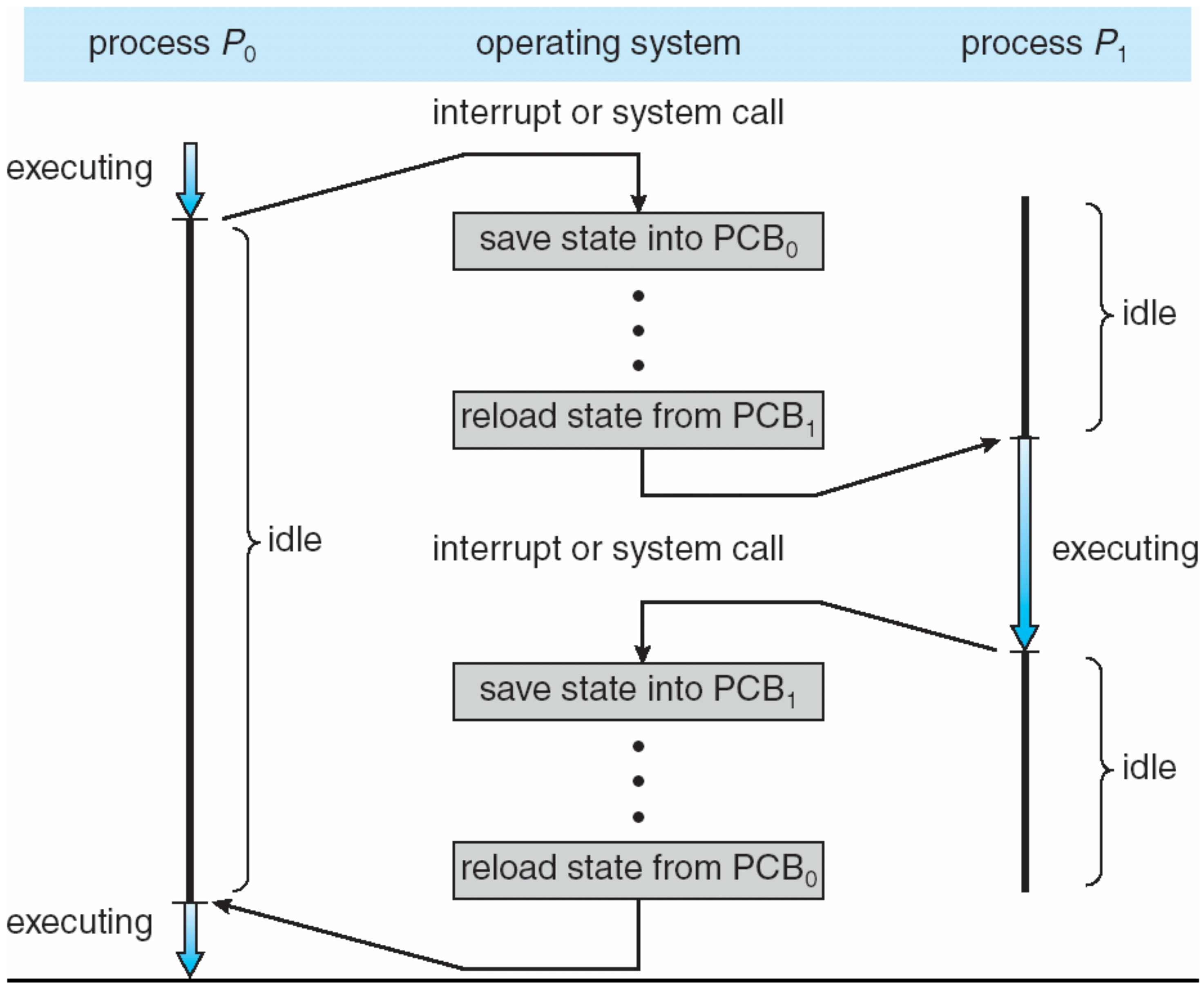
Threads

- So far, process has a single thread of execution
- Consider having multiple program counters per process
  - **Multiple locations** can execute at once
    - Multiple threads of control -> threads
- Must then have storage for thread details, multiple program counters in PCB
- Which resources are shared between threads?
  - Stack (no), heap (Y), global data (Y), code (Y)

Context Switch

- **Context switch**: the kernel switches to **another process for execution**
  - save the state of the old process
  - load the saved state for the new process
- **Context-switch is overhead**; CPU does no useful work while switching
  - the more complex the OS and the PCB, longer the context switch
- Context-switch time depends on hardware support
  - some hardware provides multiple sets of registers per CPU: multiple contexts loaded at once

Context Switch



Process Creation

- Parent process creates children processes, which, in turn create other processes, forming **a tree of processes**
  - process identified and managed via a process identifier (pid)
- Design choices:
  - three possible levels of **resource sharing**: all, subset, none
  - parent and children's **address spaces**
    - child duplicates parent address space (e.g., Linux)
    - child has a new program loaded into it (e.g., Windows)
  - **execution** of parent and children
    - **parent and children execute concurrently**
    - parent waits until children terminate

Process Creation

- UNIX/Linux system calls for process creation
  - **fork** creates a new process
  - **exec** overwrites the process' address space with a new program
  - **wait** waits for the child(ren) to terminate

C Program Forking Separate Process

```
#include <sys/types.h>
#include <studio.h>
#include <unistd.h>
int main()
{
    pid_t  pid;
    pid = fork();                      /* fork another process */
    if (pid < 0) {                     /* error occurred while forking */
        fprintf(stderr, "Fork Failed");
        return -1;
    } else if (pid == 0) {             /* child process */
        execlp("/bin/ls", "ls", NULL);
    } else {                           /* parent process */
        wait (NULL);
        printf ("Child Complete");
    }
    return 0;
}
```

fork() multiple times

```
#include <sys/types.h>
#include <studio.h>
#include <unistd.h>
int main()
{
   pid_t  pid;
   pid = fork();
   If (pid == 0) {
      fork();
   }
    fork();
    return 0;
}
```

How many unique processes are created? (including itself)

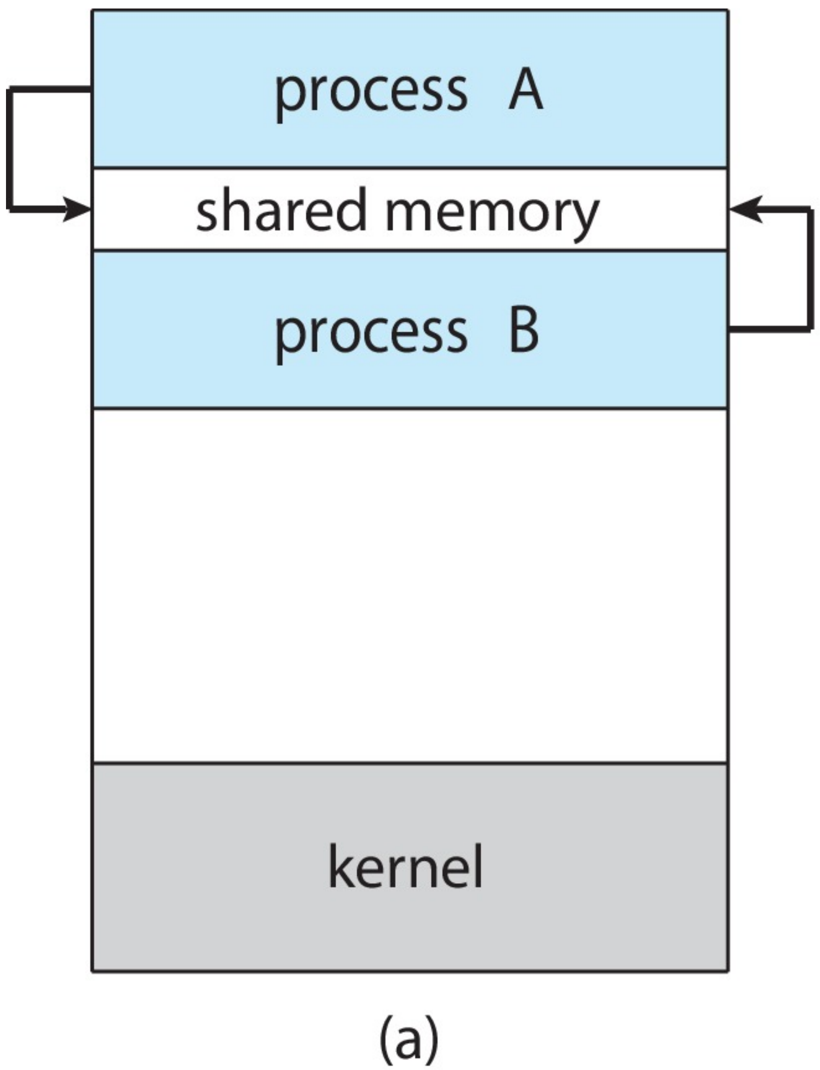
Interprocess Communication

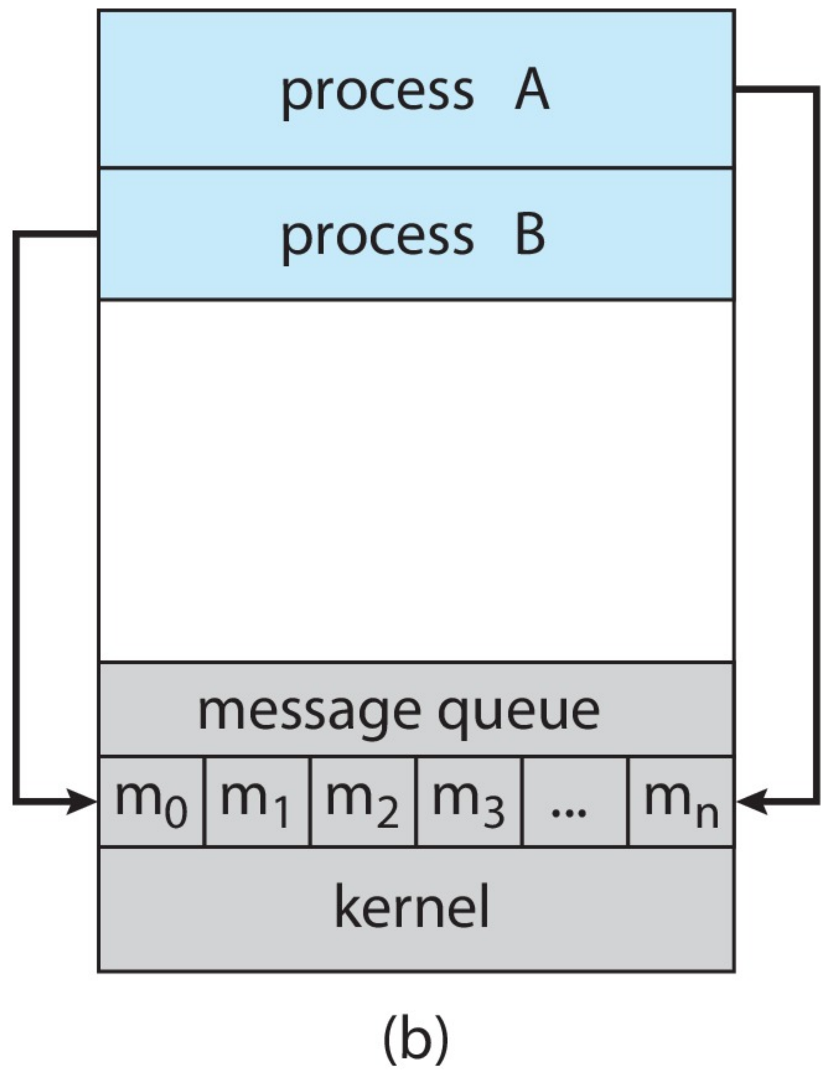
- Processes within a system may be independent or cooperating
  - **independent process**: process that cannot affect or be affected by the execution of another process
  - **cooperating process**: processes that can affect or be affected by other processes, including sharing data
    - reasons for cooperating processes: information sharing, computation speedup, modularity, convenience, Security
- Cooperating processes need **interprocess communication** (IPC)
- Two models of IPC
  - **Shared memory**
  - Message passing

Communications Models

(a) Shared memory.

(b) Message passing.





Very fast!

POSIX Shared Memory

- POSIX Shared Memory
  - Process first creates shared memory segment

    shm_fd = **shm_open**(name, O CREAT | O RDWR, 0666);

  - Also used to open an existing segment
  - Set the size of the object: **ftruncate**(shm_fd, 4096);
  - Use **mmap**() to memory-map a file pointer to the shared memory object
  - Reading and writing to shared memory is done by using the pointer returned by mmap().

A Simple Kernel Module

```c
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/printk.h>
#include <linux/sched.h>
#include <linux/sched/signal.h>

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Yajin Zhou");
MODULE_DESCRIPTION("A simple example Linux module.")
MODULE_VERSION("0.01");


static int __init os_lkm_example_init(void) {
        struct task_struct *task;

        for_each_process(task)
                printk(KERN_INFO "%s [%d]\n", task-

    return 0;


}

static void __exit os_lkm_example_exit(void) {
        printk(KERN_INFO "Goodbye, World!\n");
}

module_init(os_lkm_example_init);
module_exit(os_lkm_example_exit);
```

```c
/*
 *  hello-1.c - The simplest kernel module.
 */
#include <linux/module.h>       /* Needed by all modules */
#include <linux/kernel.h>       /* Needed for KERN_INFO */

int init_module(void)
{
        printk(KERN_INFO "Hello world 1.\n");

        /*
         * A non 0 return means init_module failed; module can't be loaded.
         */
        return 0;
}

void cleanup_module(void)
{
        printk(KERN_INFO "Goodbye world 1.\n");
}
```

insmod xxx.ko

Make menuconfig

Further reading: https://blog.sourcerer.io/writing-a-simple-linux-kernel-module-d9dc3762c234