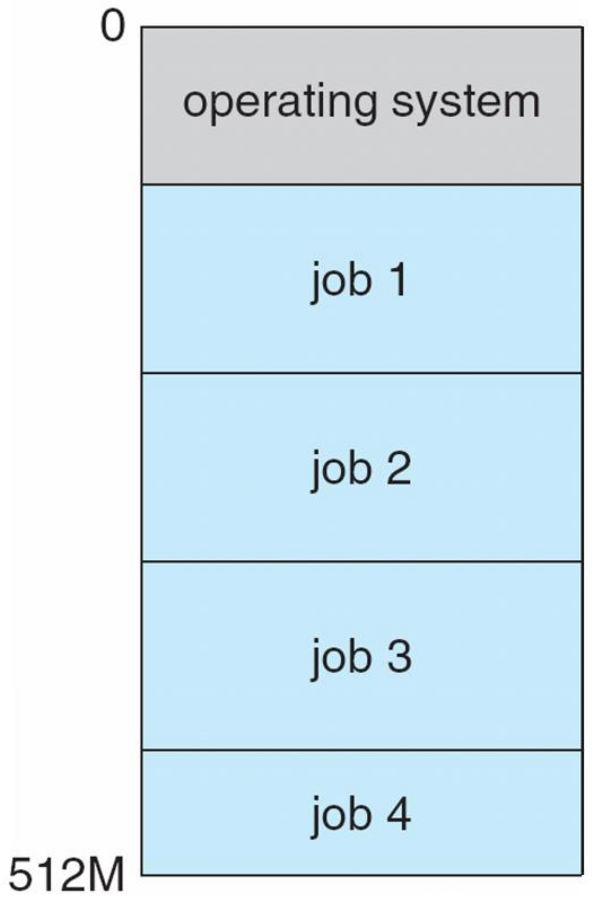
Operating System Structure

- **Multiprogramming batch systems**
  - A bit historical
  - Needed for efficiency: Single user cannot keep CPU and I/O devices busy at all times
  - Multiprogramming organizes jobs (code and data) so CPU always has one to execute
  - A subset of total jobs in system is kept in memory
  - One job selected and run via **job scheduling**
  - When it has to wait (for I/O for example), OS switches to another job
  - Typically used **non-preemptive = cooperative** multitasking

- **Timesharing/interactive systems**
  - Logical extension in which CPU switches jobs so frequently that users can interact with each job while it is running, creating **interactive** computing (**response time** should be < 1 second) -> **preemptive**
  - Each user has at least one program executing in memory ⇨**process**
  - If several jobs ready to run at the same time ⇨ **CPU scheduling** selects one of them

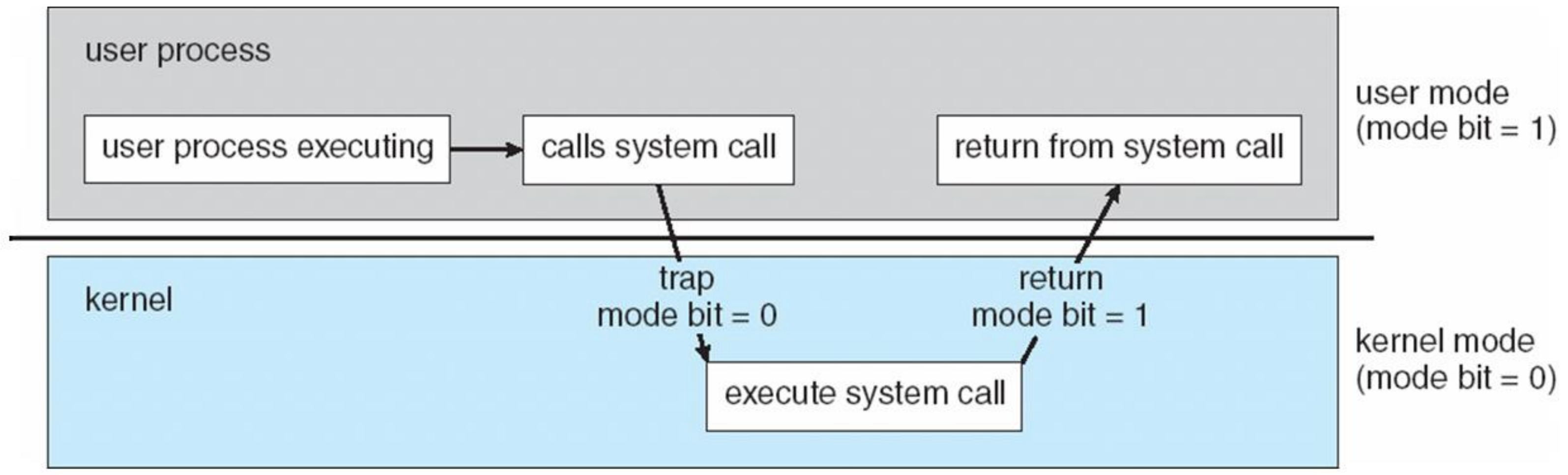


Operating-System Operations

- The operating system is invoked via interrupts. Invocation may be due to either:
  - Hardware interrupts by one of the devices.
    - A **timer** interrupt is a hardware interrupt caused by an on-chip timer, and is used to preempt applications and invoke the OS kernel on regular intervals (called **OS tick**)
      - The interrupt interval is usually 1 to 50 mS.
    - Interrupts **from device controllers**.
  - Software interrupts:
    - Operating system services are requested using the **trap** instruction (aka **system request)**, which is a software interrupt.
    - Illegal operations:
      - Software error (e.g., division by zero) causes an exception
      - Illegal instruction or illegal access also cause exceptions.
- Note that the operating system may also be running its own threads (kernel threads).
  - The OS scheduler thus schedules user jobs/processes AND kernel threads

Operating-System Operations (cont.)

- **Dual-mode** operation of CPU allows OS to **protect itself** and other system components
  - **Mode bit** provided by hardware determines whether the CPU is in **User mode** or **kernel mode.**
    - Provides ability to distinguish when system is running user code or kernel code
    - Some **instructions** designated as **privileged**, only executable in kernel mode
    - Some **memory locations** may be configured to be only accessible in kernel mode.
  - System call (using the trap instruction) changes mode to kernel-mode.
  - Return from a system call resets the mode bit back to user-mode



Transition from user mode to kernel mode

Operating-System Operations (cont.)

- Increasingly CPUs support multi-mode operations:
  - Privileged/kernel modes: e.g. interrupts, kernel threads, etc.
  - Non-privileged/user modes: e.g. user threads/processes, **virtual machine manager** (**VMM**) mode for guest **VMs**, etc.

Open-Source Operating Systems

- Operating systems made available in source-code format rather than just binary **closed-source**
- Examples include **GNU/Linux** and **BSD UNIX** (e.g. FreeBSD and NetBSD) and a few more.
- When building or developing applications and drivers for an open source OS, good attention must be paid to the licensing model:
  - Reciprocal: e.g. GPLv1, GPLv2, etc.
  - Permissive: e.g. BSD, MIT, etc.
- Can be installed on VMMs like VMware Workstation (Free on Windows) or Virtualbox (open source and free on many platforms)
  - Use to run guest operating systems for exploration, testing or educational purposes.

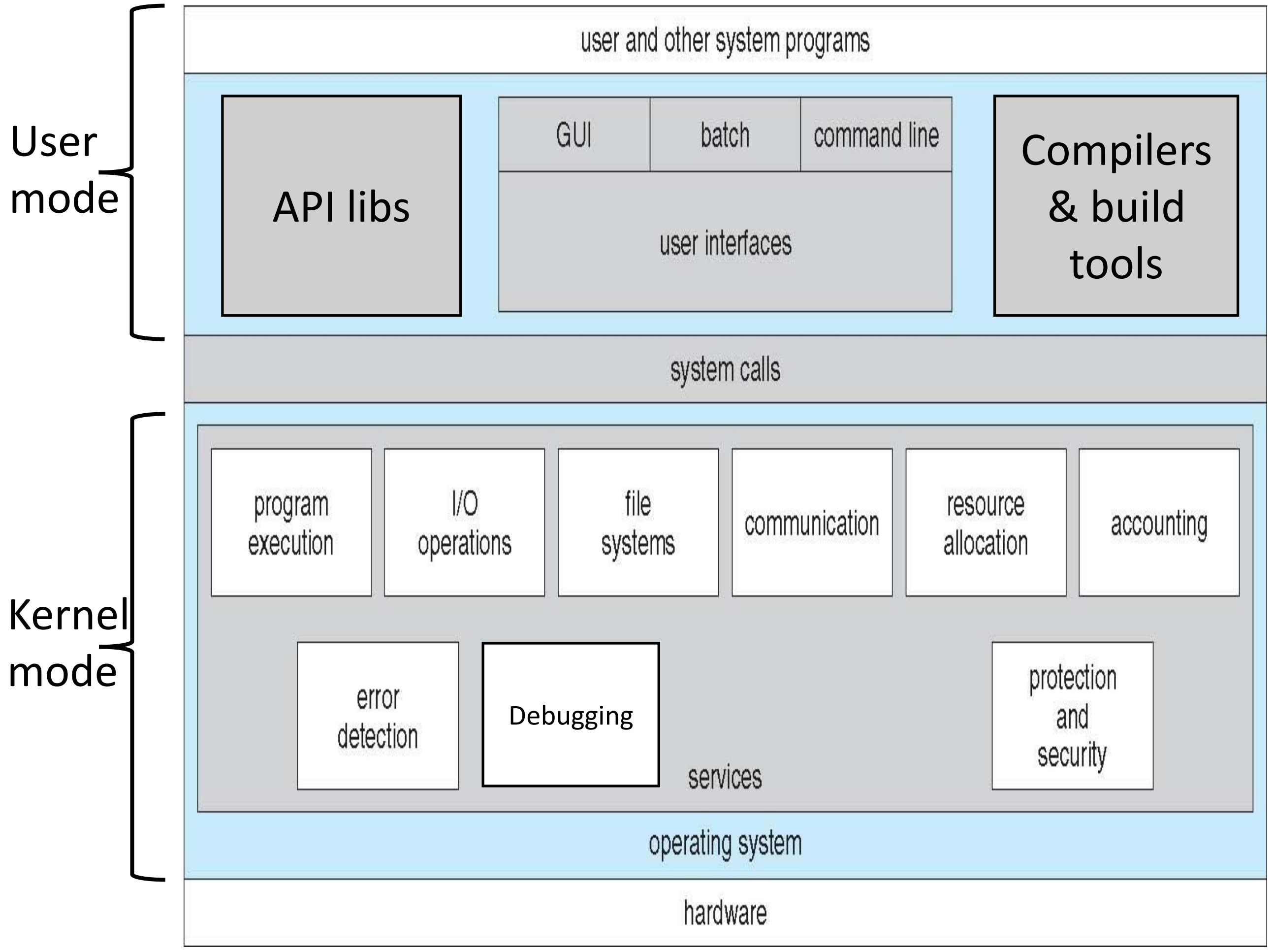
Some notes

- Vector table can be remapped from ROM to another area of the main memory, once the kernel boots, and thus the kernel can then setup its interrupt handlers.

- C cannot access CPU registers. Also, before a C program can run, its environment needs to be initialized (e.g stacks, constants, pre-initialized variables, etc.).
    - For that reason, portions of the startup code in the boot ROM is often written in assembly language, but once it initializes the C environment, C code can be run thereafter.

2.1 Operating System Services

- One of the operating systems' main tasks is to provide **an environment and services** for application programs to run:
  - **API Libraries**
  - **Compilers and build tools**
  - **User interface** - Almost all operating systems have a user interface (**UI**).
    - Varies between **Command-Line (CLI)**, **Graphics User Interface (GUI)**, **Batch**
  - **Program execution** - The system must be able to load a program into memory and to run that program and also end its execution (normally or abnormally - indicating error)
  - **I/O operations** - A running program may require I/O, which may involve a file or an I/O device

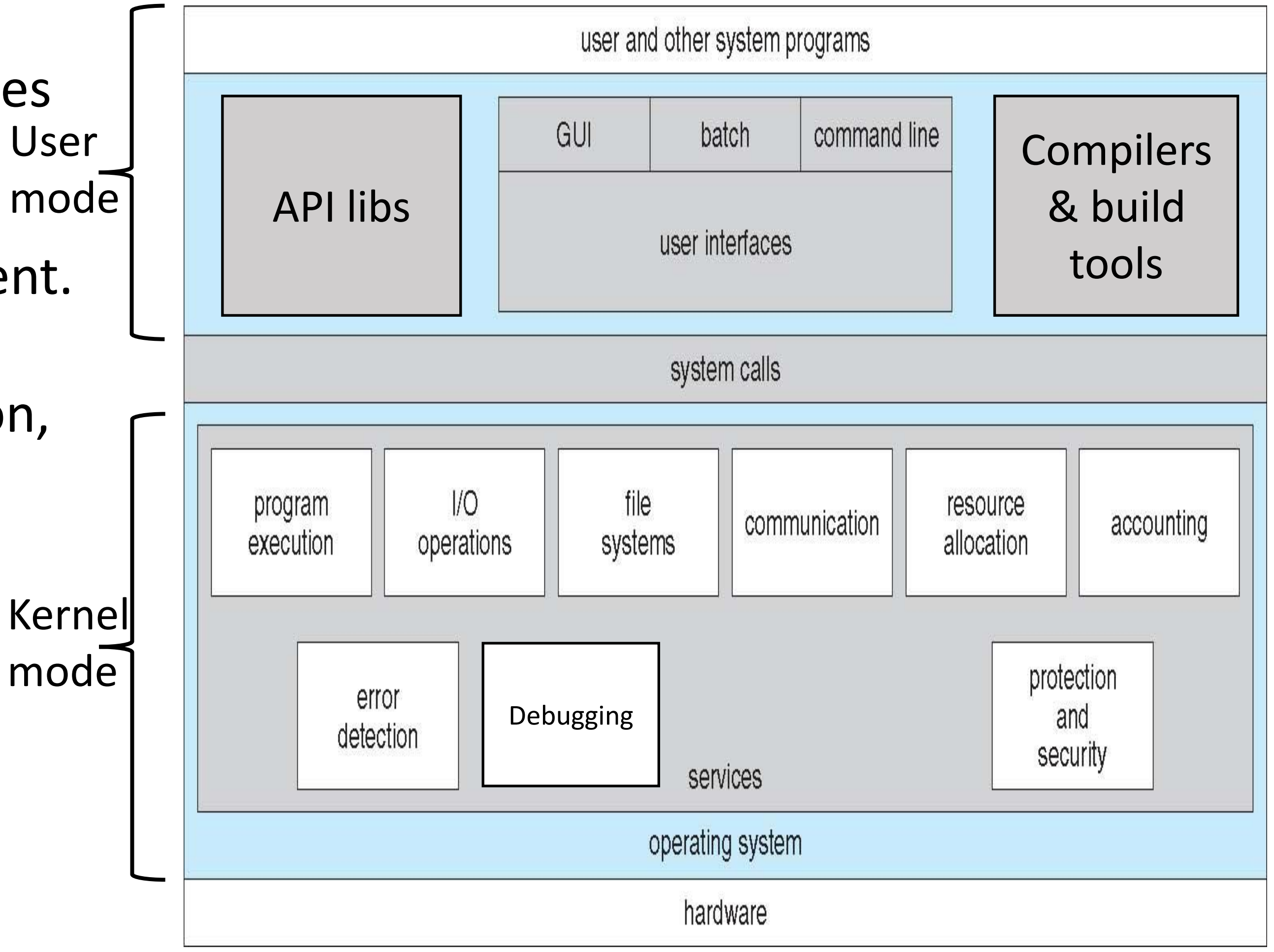


Operating-System Operations (cont.)

- Timer to prevent infinite loop / process hogging resources (**in preemptive multitasking**)
  - Timer is set to interrupt the CPU after some time period (e.g. 1 – 50 ms)
    - The interrupt is handled by the OS kernel.
    - The timer registers are memory-mapped to a memory area that can be accessed only in privileged mode.
  - The kernel sets up the timer for the next interrupt (timer tick) before scheduling a process to run.
    - This is in order to regain control or preempt a running process that exceeds its allotted time

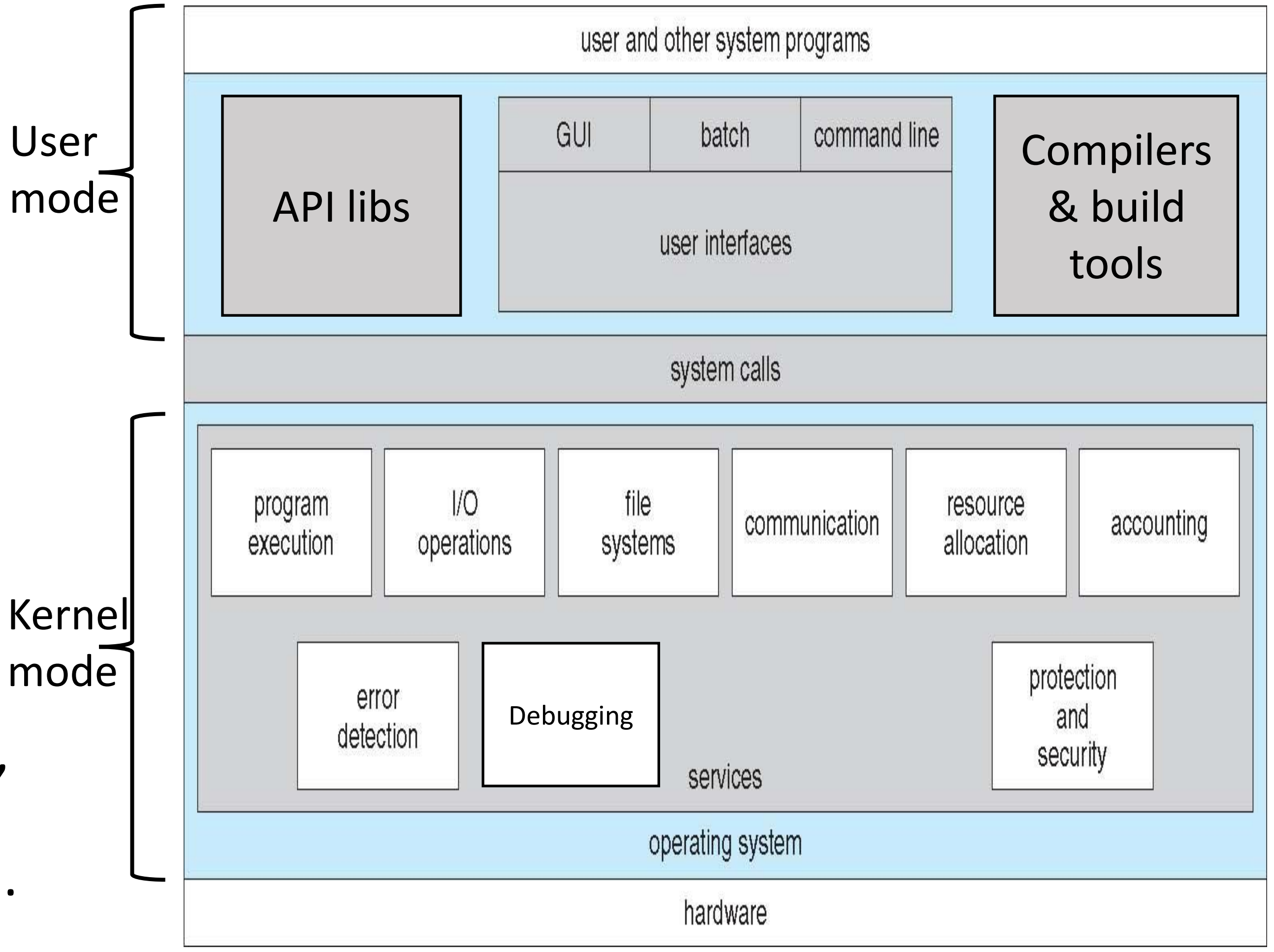
Operating System Services (Cont.)

- **File-system manipulation** - The file system is of particular interest. Programs need to read and write files and directories, create and delete them, search them, list file Information, permission management.
- **Interprocess Communications** – Processes may exchange information, on the same computer or between computers over a network
  - Communications may be via shared memory or through message passing (packets or messages moved by the OS)



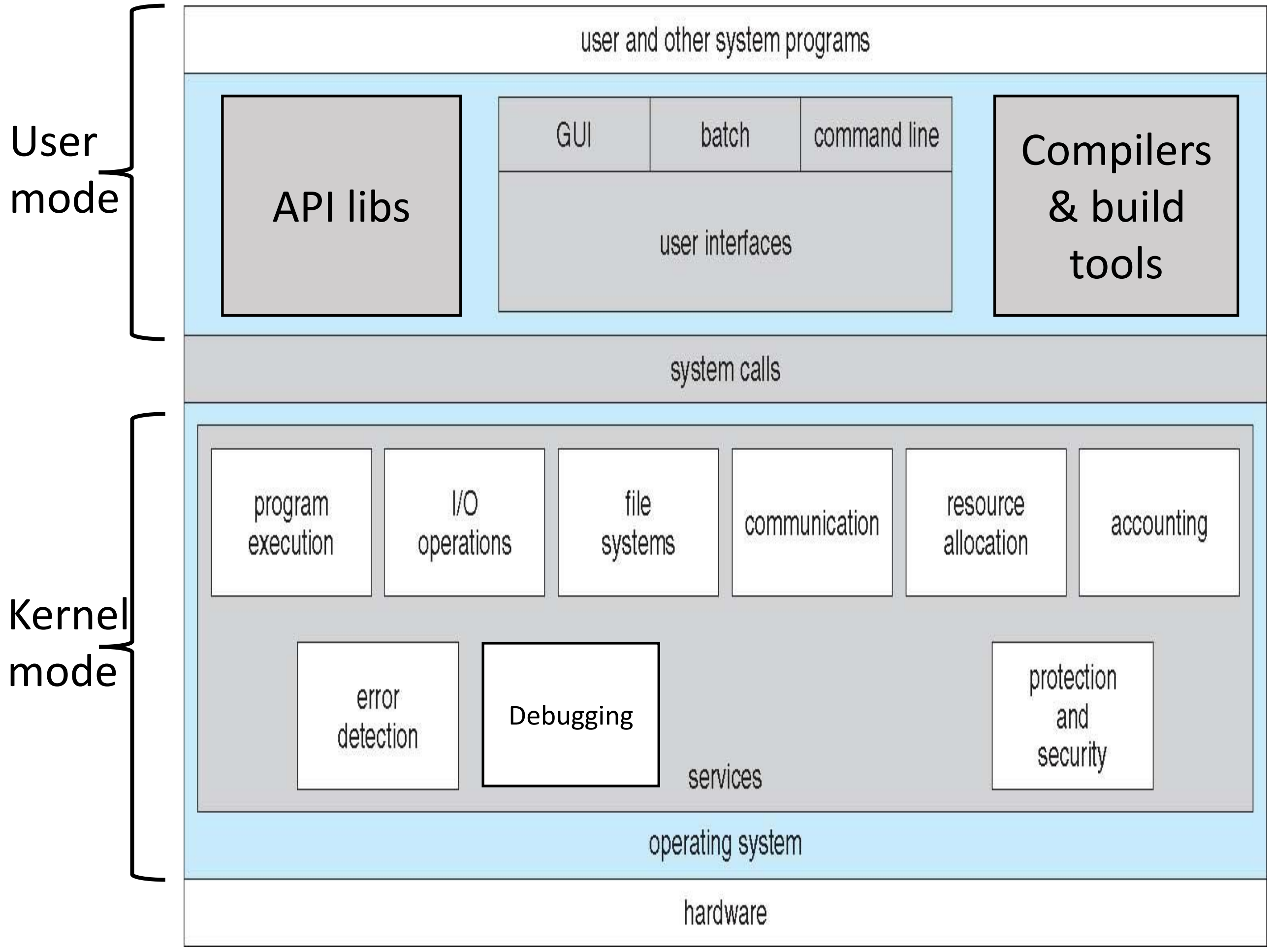
Operating System Services (Cont.)

- **Error detection** – OS needs to be constantly aware of possible errors
  - May occur in the CPU and memory hardware, in I/O devices, or in user programs.
  - For each type of error, OS should take the appropriate action to ensure correct and consistent computing
- **Debugging:** OS provides debugging facilities which can greatly enhance the user's and programmer's abilities to efficiently use the system. In Linux, the kernel component that provides this facility is the "**ptrace**" system call. User mode debuggers (e.g. **gdb**) use this kernel facility to debug user programs.



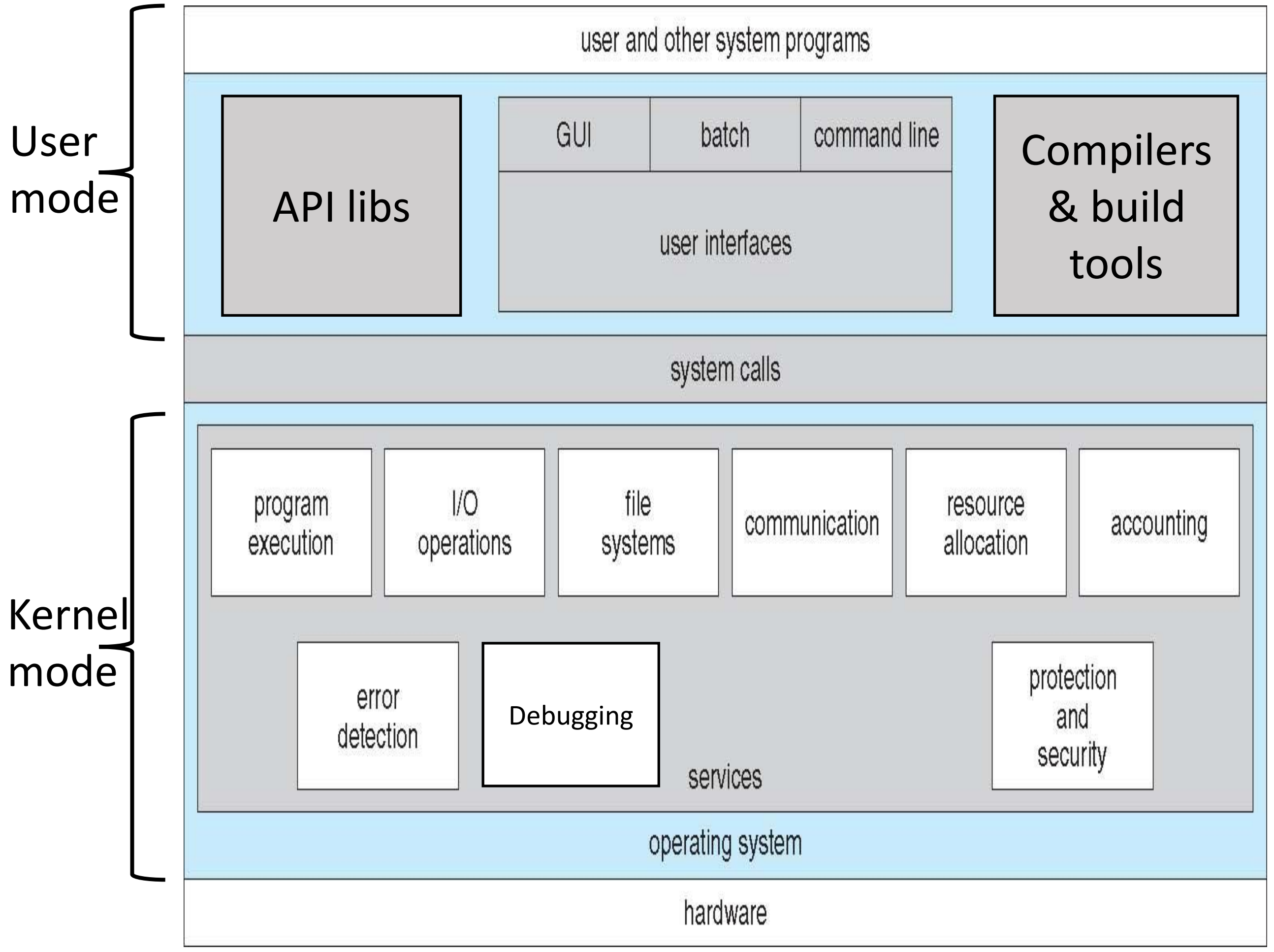
Operating System Services (Cont.)

- The second main goal of an OS is to provide functions that ensure the efficient and secure operation of the system:
  - **Resource allocation -** When multiple jobs (or processes) are running concurrently, resources must be allocated to each of them
    - Many types of resources - CPU cycles, main memory, file storage, I/O devices.
  - **Accounting -** To keep track of which **users/processes** use how much and what kinds of computer **resources** (may collect statistics useful for researchers or system admins for detecting system misuse or intrusion).



Operating System Services (Cont.)

- **Protection and security -** The owners of information stored in a multiuser or networked computer system may want to control use of that information, concurrent processes should not interfere with each other
  - **Protection** involves ensuring that all access to system resources is controlled
  - **Security** of the system from outsiders requires user authentication, extends to defending external I/O devices from invalid access attempts

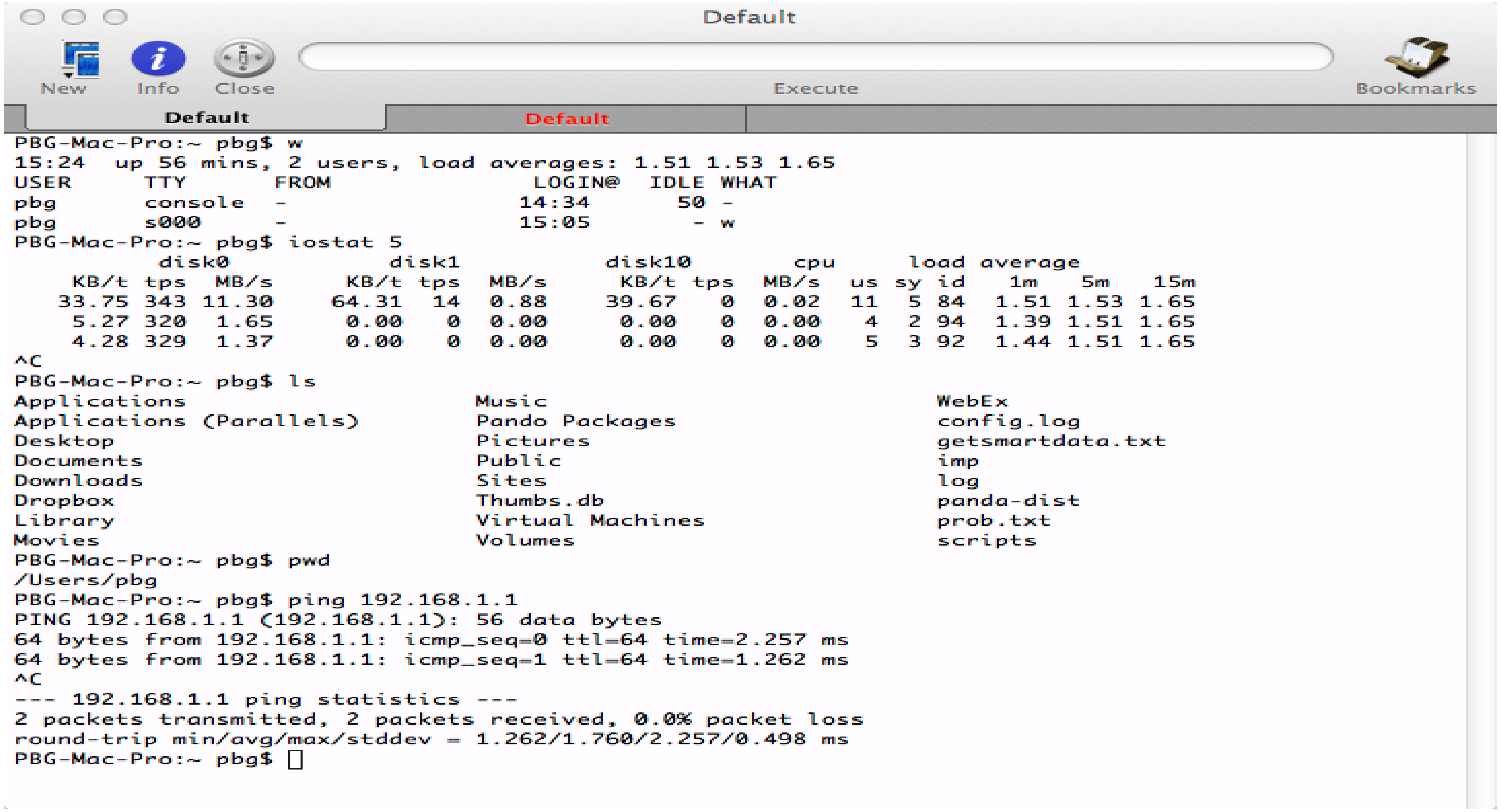


2.2 Operating System's User Interface - CLI

CLI or **command interpreter** allows direct command entry

- Usually implemented as a systems program
- Sometimes multiple flavors implemented – **shells** (e.g. c shell, Bourne shell, bash, korn shell, ash, etc.).
- Primarily fetches a command from user and executes it
- Two different implementations:
  - Commands are built into the shell (e.g. **BusyBox**, widely used in embedded Linux).
  - Commands are just names of other programs that execute the command. In this implementation, adding new features doesn't require shell modification (For Linux, usually placed at /bin, as required by the Linux Foundation "File System Hierarchy", FSH)

Bourne Shell Command Interpreter



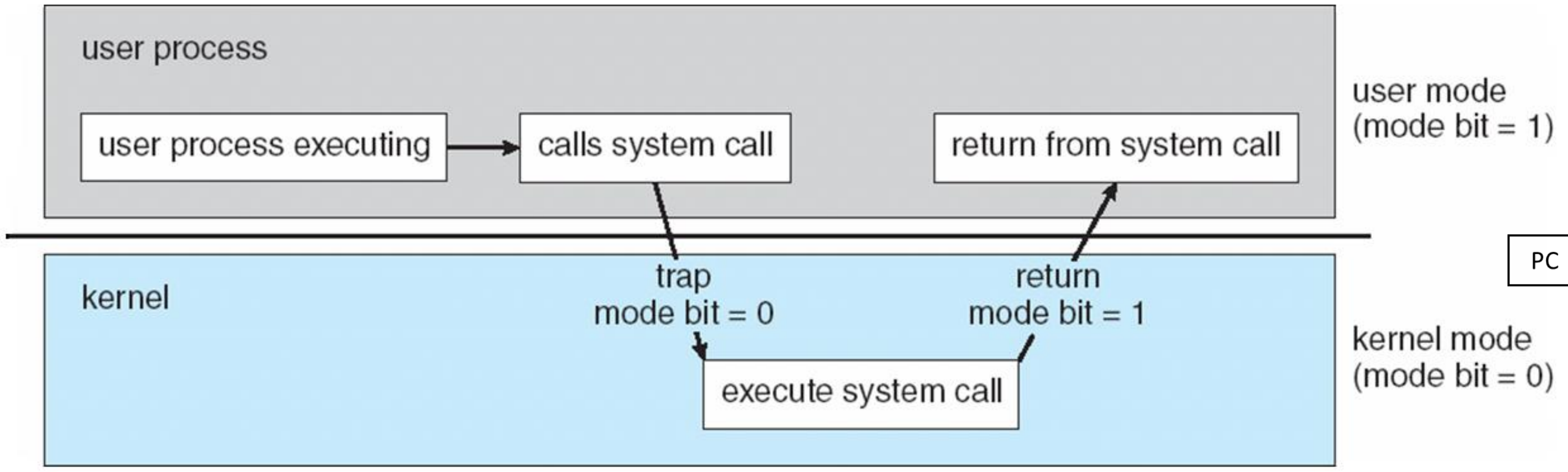
```
PBG-Mac-Pro:~ pbg$ w
15:24  up 56 mins, 2 users, load averages: 1.51 1.53 1.65
USER     TTY      FROM              LOGIN@  IDLE WHAT
pbg      console  -                14:34      50 -
pbg      s000     -                15:05       - w
PBG-Mac-Pro:~ pbg$ iostat 5
          disk0           disk1           disk10         cpu      load average
    KB/t tps  MB/s    KB/t tps  MB/s    KB/t tps  MB/s  us sy id   1m   5m   15m
   33.75 343 11.30   64.31  14  0.88   39.67   0  0.02  11  5 84  1.51 1.53 1.65
    5.27 320  1.65    0.00   0  0.00    0.00   0  0.00   4  2 94  1.39 1.51 1.65
    4.28 329  1.37    0.00   0  0.00    0.00   0  0.00   5  3 92  1.44 1.51 1.65
^C
PBG-Mac-Pro:~ pbg$ ls
Applications                    Music                           WebEx
Applications (Parallels)        Pando Packages                  config.log
Desktop                         Pictures                        getsmartdata.txt
Documents                       Public                          imp
Downloads                       Sites                           log
Dropbox                         Thumbs.db                       panda-dist
Library                         Virtual Machines                prob.txt
Movies                          Volumes                         scripts
PBG-Mac-Pro:~ pbg$ pwd
/Users/pbg
PBG-Mac-Pro:~ pbg$ ping 192.168.1.1
PING 192.168.1.1 (192.168.1.1): 56 data bytes
64 bytes from 192.168.1.1: icmp_seq=0 ttl=64 time=2.257 ms
64 bytes from 192.168.1.1: icmp_seq=1 ttl=64 time=1.262 ms
^C
--- 192.168.1.1 ping statistics ---
2 packets transmitted, 2 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 1.262/1.760/2.257/0.498 ms
PBG-Mac-Pro:~ pbg$ 
```

User Operating System Interface - GUI

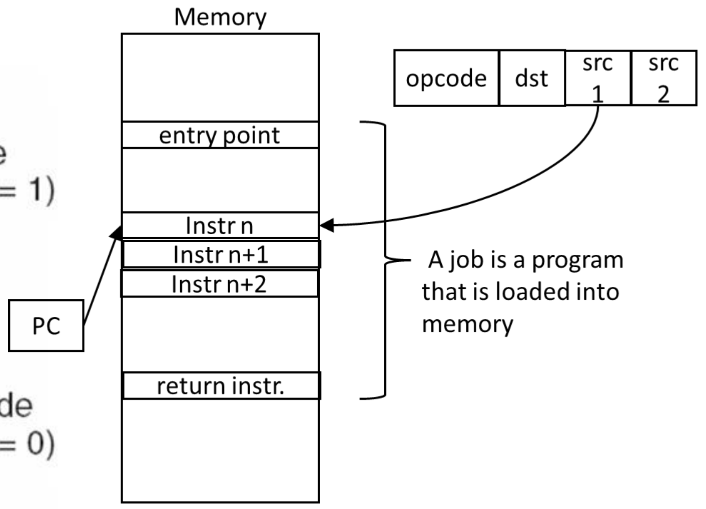
- User-friendly **desktop** metaphor interface
  - Invented at Xerox PARC in 1970 (The first computer to use an early version of the desktop metaphor was the experimental Xerox Alto and the first commercial computer that adopted this kind of interface was the Xerox Star).
  - Usually mouse, keyboard, and monitor
  - **Icons** represent files, programs, actions, etc
  - Clicking mouse buttons over objects in the interface cause different actions (provide information, options, execute function, open directory, etc.)
- Most of today's systems include both CLI and GUI interfaces
  - Microsoft Windows is GUI with CLI "command" shell
  - Apple Mac OS X is "Aqua" GUI interface with UNIX kernel underneath and shells available
  - Unix and Linux have CLI with optional GUI interfaces (CDE, KDE, GNOME)

2.3 System Calls

- User-mode program's interface to the services provided by the OS
  - User mode programs **cannot** directly call driver or operating system functions. Instead they need to use system calls.
  - System calls may also be referred to as “supervisor call”, “trap” or sometimes “software interrupt”
  - The application program uses a special machine instruction to perform a system call:
    - SWI – ARM CPUs
    - INT – intel CPUs



Transition from user mode to kernel mode
(from Chapter 1)



System Calls – cont.

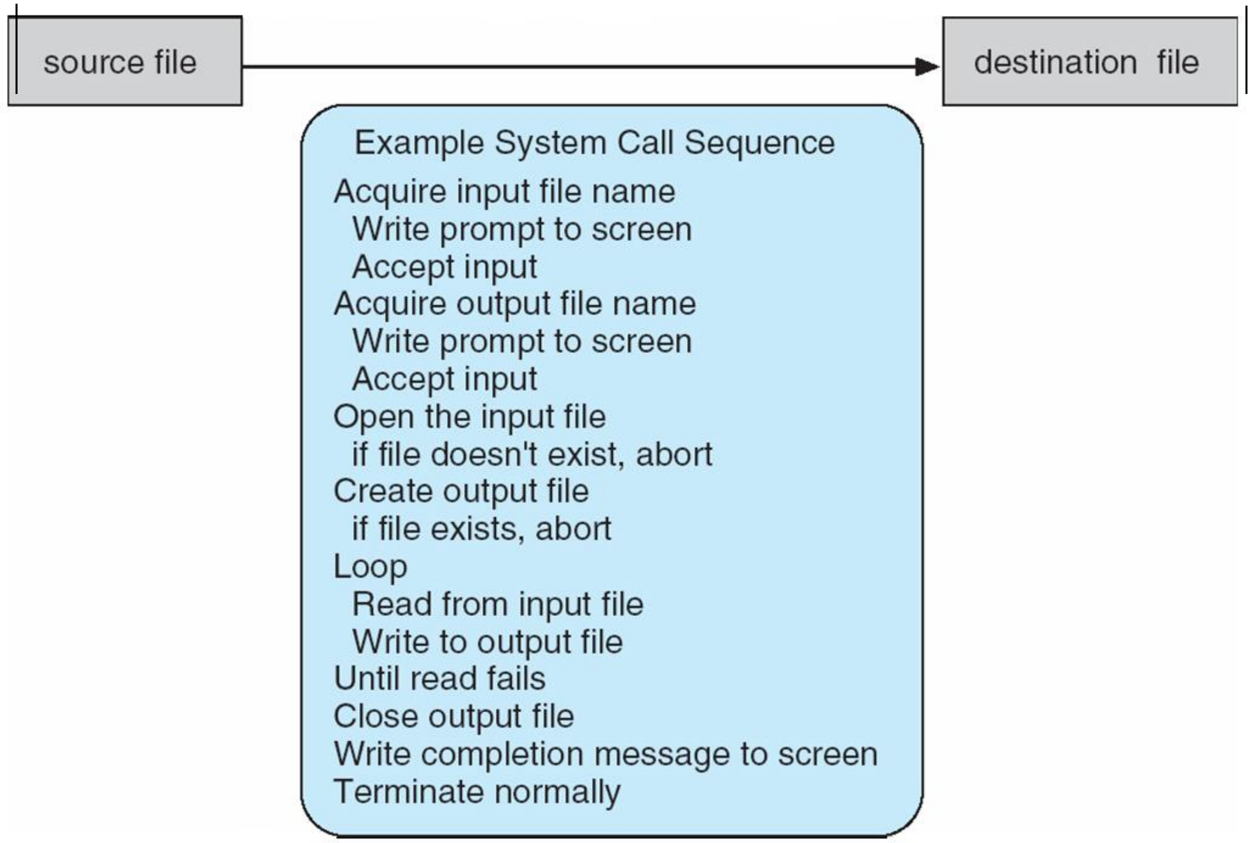
- Mostly accessed by programs via a high-level **Application Programming Interface (API)** libraries (e.g. libc for unix/linux) rather than direct system call use

- Three common APIs are:
  - Win32 API for Windows
  - POSIX API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X)
  - Java API for the Java virtual machine (JVM)

Note that unless otherwise stated, the system-call names used throughout this course are generic

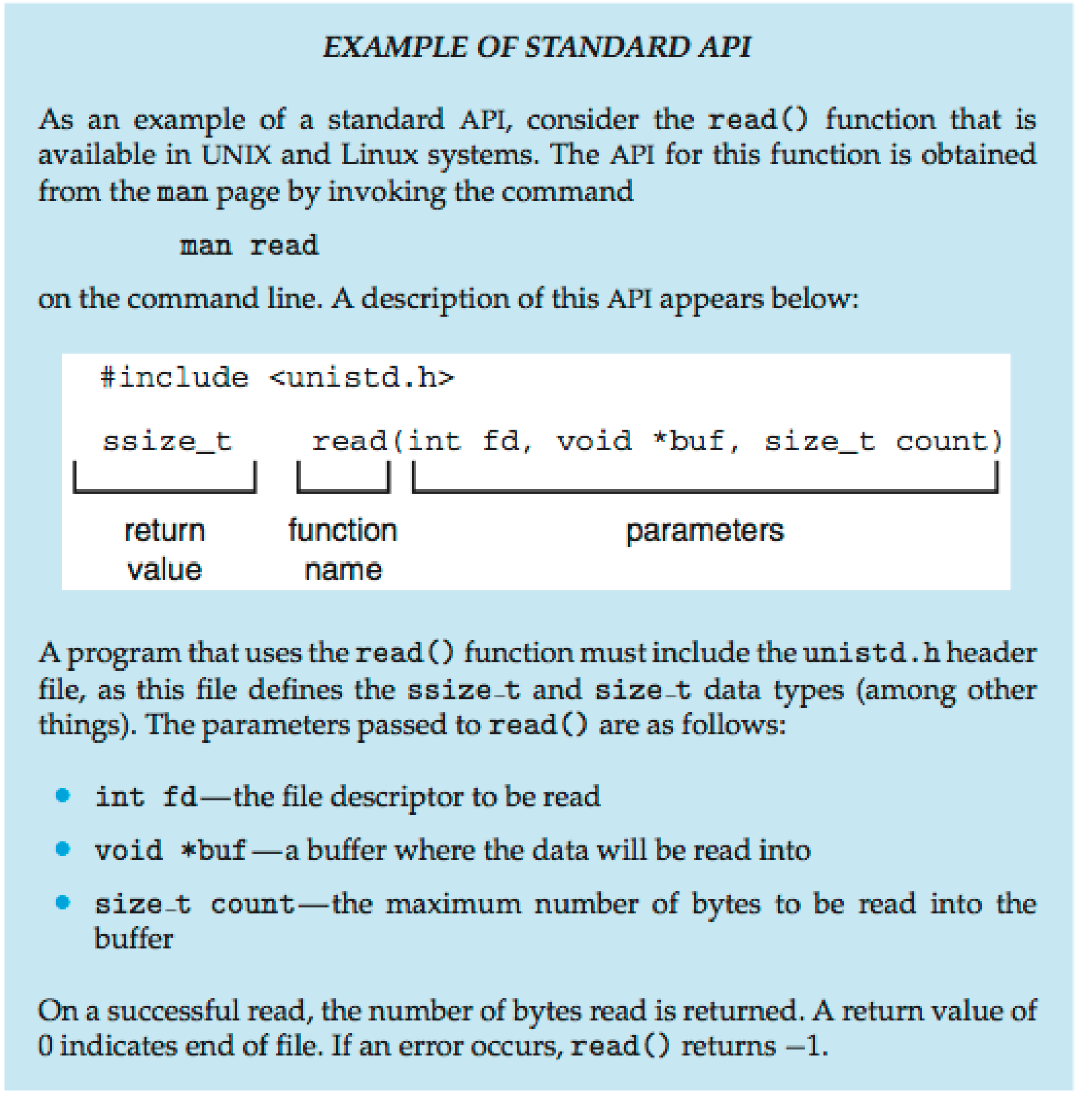
Example of System Calls

- The example below demonstrates the steps needed to copy the contents of one file to another file → A sequence of system calls results



```
Acquire input file name
  Write prompt to screen
  Accept input
Acquire output file name
  Write prompt to screen
  Accept input
Open the input file
  if file doesn't exist, abort
Create output file
  if file exists, abort
Loop
  Read from input file
  Write to output file
Until read fails
Close output file
Write completion message to screen
Terminate normally
```

Example of Standard API



EXAMPLE OF STANDARD API

As an example of a standard API, consider the `read()` function that is available in UNIX and Linux systems. The API for this function is obtained from the `man` page by invoking the command

```
man read
```

on the command line. A description of this API appears below:

```
#include <unistd.h>
```

$$\underbrace{\texttt{ssize\_t}}_{\text{return value}}\quad\underbrace{\texttt{read}}_{\text{function name}}\underbrace{\texttt{(int fd, void *buf, size\_t count)}}_{\text{parameters}}$$

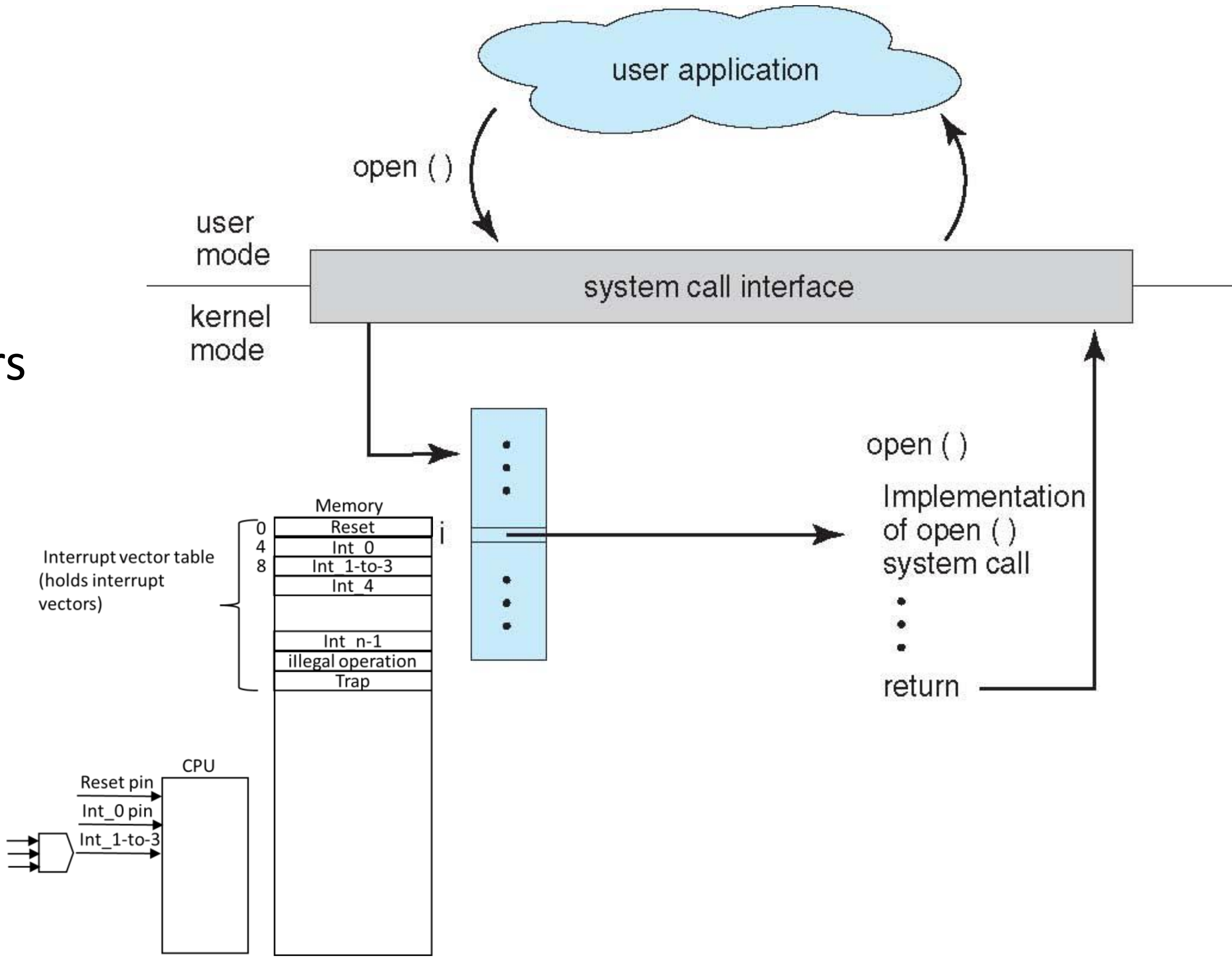
A program that uses the `read()` function must include the `unistd.h` header file, as this file defines the `ssize_t` and `size_t` data types (among other things). The parameters passed to `read()` are as follows:

- `int fd`—the file descriptor to be read
- `void *buf`—a buffer where the data will be read into
- `size_t count`—the maximum number of bytes to be read into the buffer

On a successful read, the number of bytes read is returned. A return value of 0 indicates end of file. If an error occurs, `read()` returns −1.

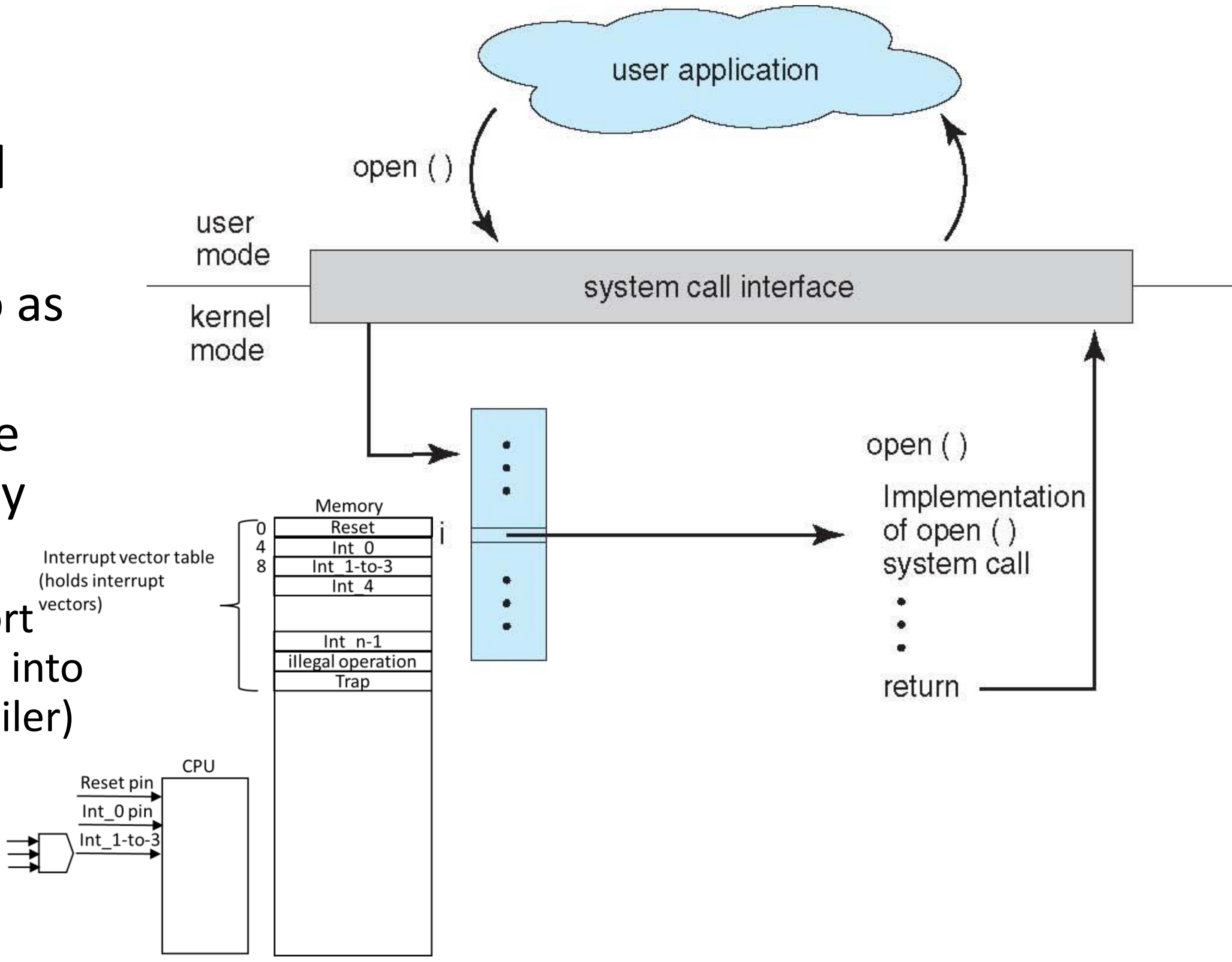
System Call Implementation

- Typically, a number associated with each system call
  - **System-call interface** maintains a table indexed according to these numbers
- The system call interface invokes the intended system call in OS kernel and returns status of the system call and any return values



System Call Implementation (cont.)

- The caller need not know anything about how the system call is implemented
  - Just needs to obey API and understand what OS will do as a result of the call
  - Most details of OS interface hidden from programmer by API
    - Managed by run-time support library (set of functions built into libraries included with compiler)



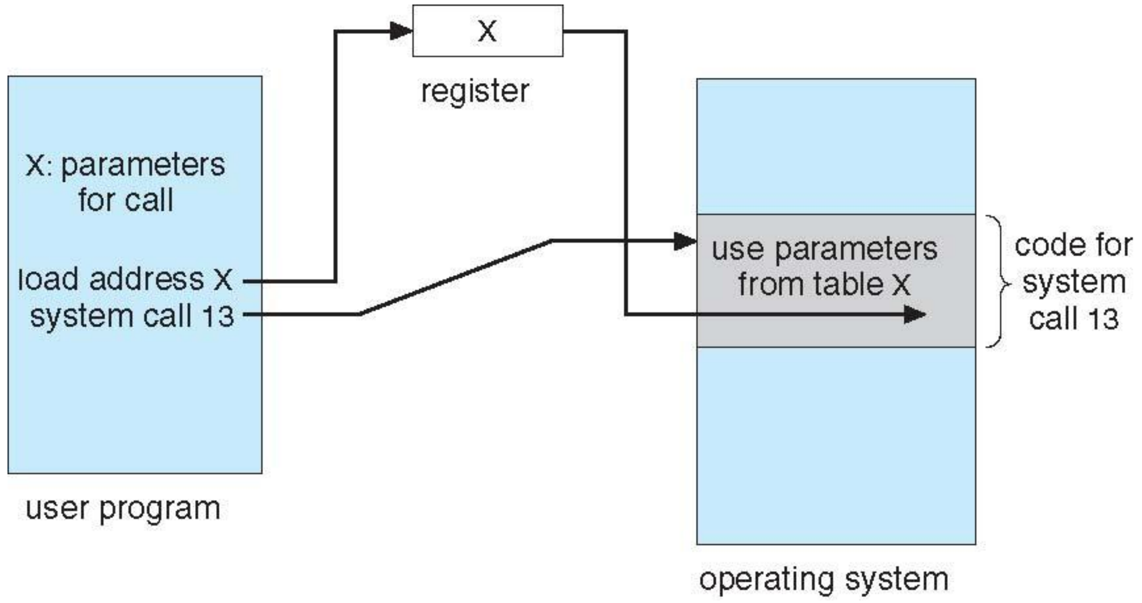
System Call Parameter Passing

- Often, more information is required than simply identity of desired system call
  - Exact type and amount of information vary according to system function called
- Three general methods used to pass parameters to the OS
  1. Simplest: pass the parameters in registers
     - Disadvantage: In some cases, may be more parameters than registers

System Call Parameter Passing – cont.

2. Parameters stored in a block in memory, and address of block/table passed as a parameter in a register
   - This approach taken by Linux and Solaris
3. Parameters placed, or **pushed**, onto the **stack** by the program and **popped** off the stack by the operating system

- Unlike the registers method, the block and stack methods do not limit the number or length of parameters being passed



Parameter Passing via memory block/table

2.4 Types of System Calls

- Process control (and support)
  - create process, terminate process, end or abort
  - load, execute
  - get process attributes, set process attributes
  - wait for time
  - wait event, signal event
  - allocate and free memory
  - Dump memory if error
  - **Debugger** for determining bugs in a program
  - **Locks** for managing access to shared data between processes

Types of System Calls (cont.)

- File management
  - create file, delete file
  - open, close file
  - read, write, reposition
  - get and set file attributes (e.g. filename, type, protection code, accounting info, etc.)
- Device management
  - request device, release device (to exclusive access to a device)
  - read, write, reposition
  - get device attributes, set device attributes
  - logically attach or detach devices (make it look like a file on the file system)

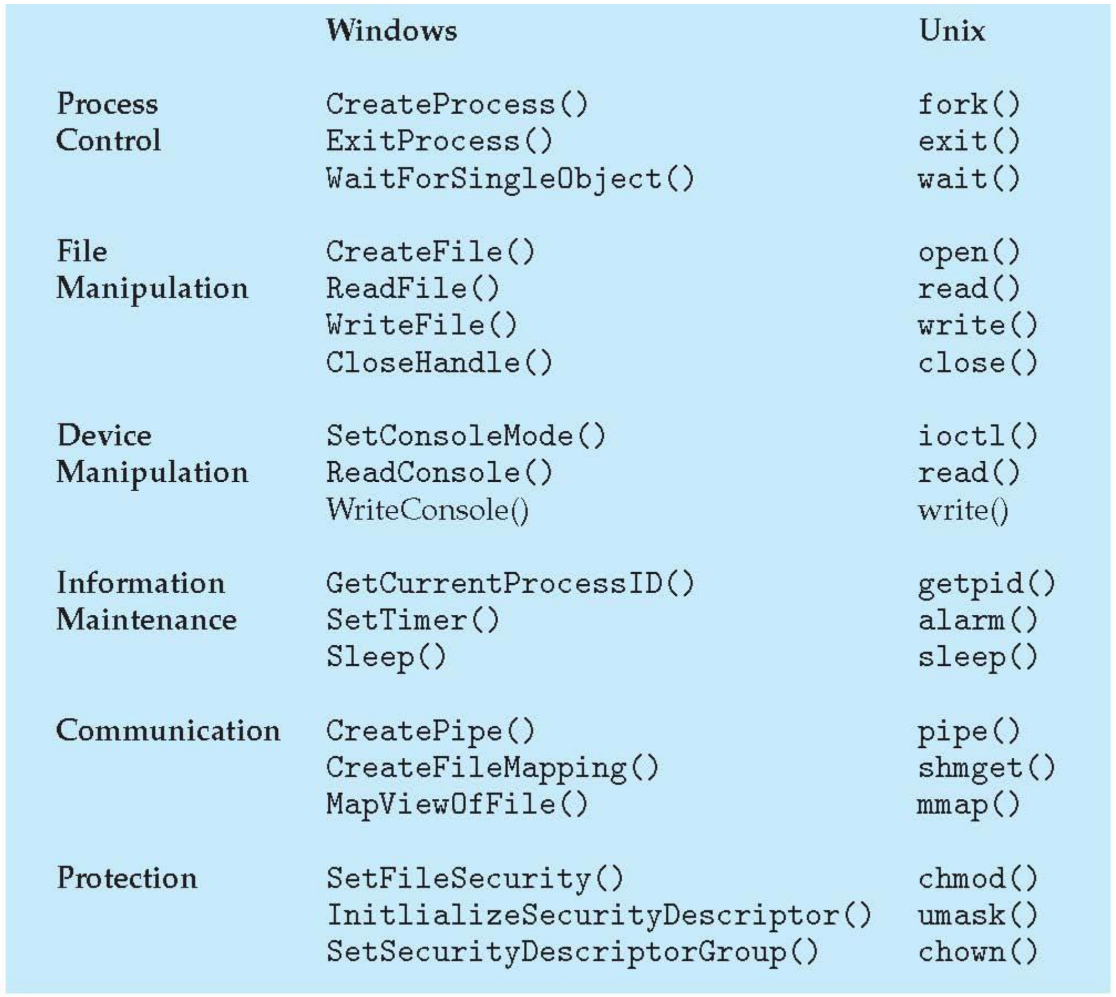
Types of System Calls (Cont.)

- Information
  - get system time or date, set time or date
  - get system data (e.g. OS version, number of users, memory or disk space, etc.).
  - Get the time profile of a process (via regular timer interrupts)
  - get and set process, file, or device attributes
- Communications
  - create, delete, open and close communication connection
  - In **message passing model,** processes can send and receive messages to **host name** or **process name** (name is first translated to a process ID (PID))
    - A process waiting for a connection request is a **server or a Daemon**
  - In **Shared-memory model,** processes create and gain access to memory regions shared with other processes.
  - Message passing is useful for small messages, while shared memory is preferred in larger messages.

Types of System Calls (Cont.)

- Protection (of **resources**)
  - Control access to resources (e.g. files, disks, memory locations, etc.)
  - Get and set permissions of resources
  - Allow and deny user access of resources.

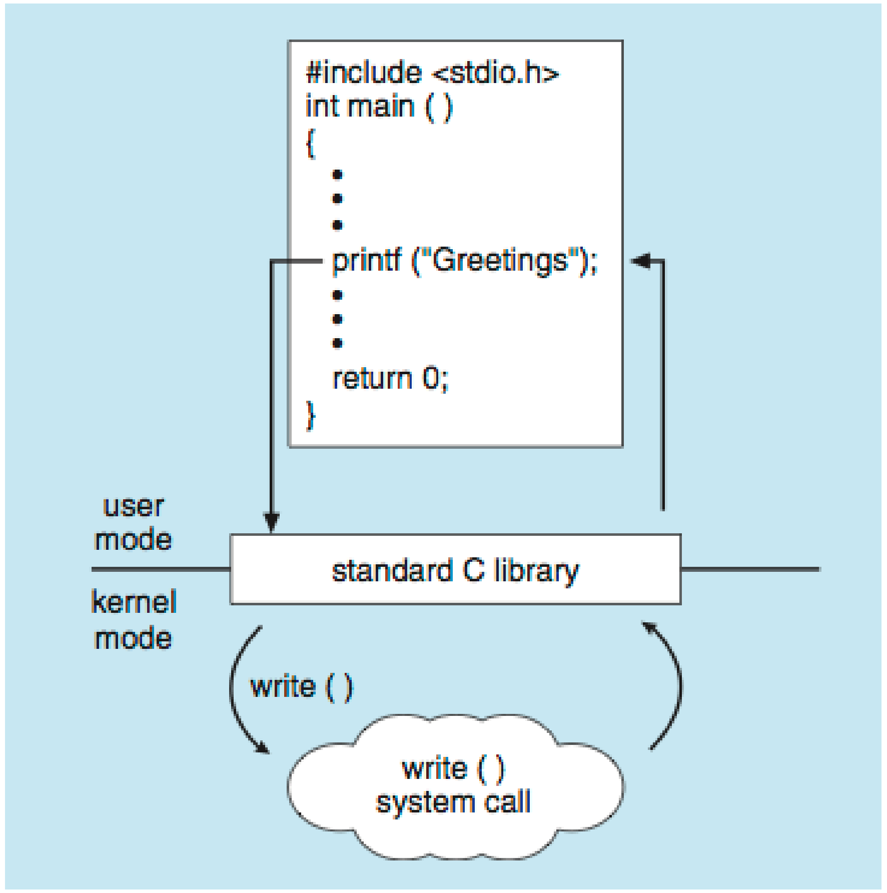
Examples of Windows and Unix System Calls



	Windows	Unix
Process Control	CreateProcess()	fork()
	ExitProcess()	exit()
	WaitForSingleObject()	wait()
File Manipulation	CreateFile()	open()
	ReadFile()	read()
	WriteFile()	write()
	CloseHandle()	close()
Device Manipulation	SetConsoleMode()	ioctl()
	ReadConsole()	read()
	WriteConsole()	write()
Information Maintenance	GetCurrentProcessID()	getpid()
	SetTimer()	alarm()
	Sleep()	sleep()
Communication	CreatePipe()	pipe()
	CreateFileMapping()	shmget()
	MapViewOfFile()	mmap()
Protection	SetFileSecurity()	chmod()
	InitlializeSecurityDescriptor()	umask()
	SetSecurityDescriptorGroup()	chown()

Standard C Library Example

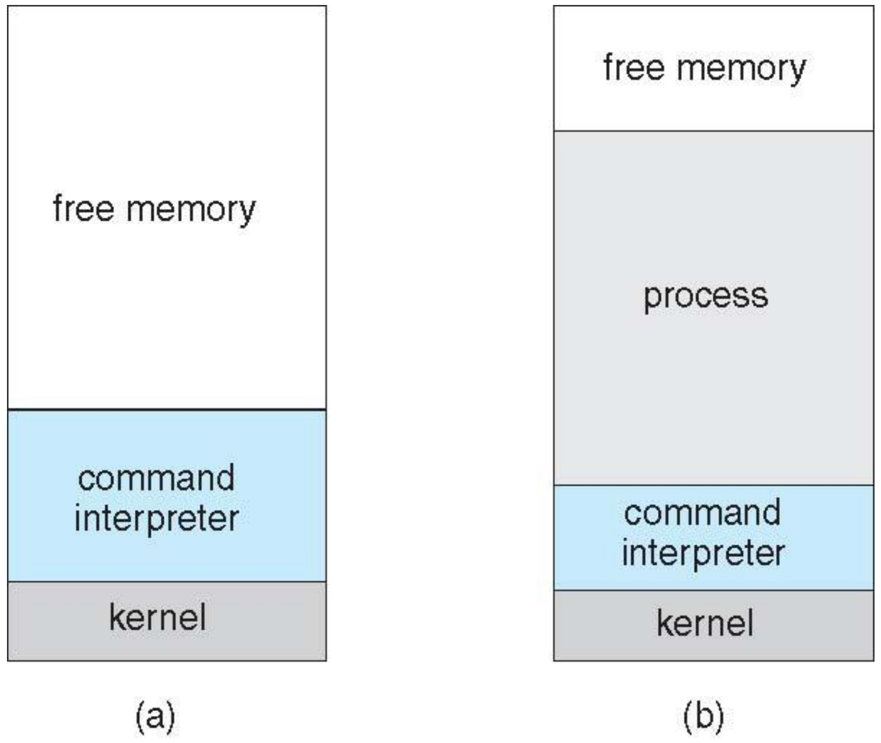
- C program invoking printf() library call, which calls write() system call



```
#include <stdio.h>
int main ( )
{
   •
   •
   •
   printf ("Greetings");
   •
   •
   •
   return 0;
}
```

Process control example - MS-DOS

- Single-tasking
- Shell invoked when system booted
- Simple method to run program
- Initial versions had a single memory space
- Loads program into memory, overwriting all but the kernel. Only a minimal set of shell routines are kept.
- Program exit -> remainder of shell reloaded

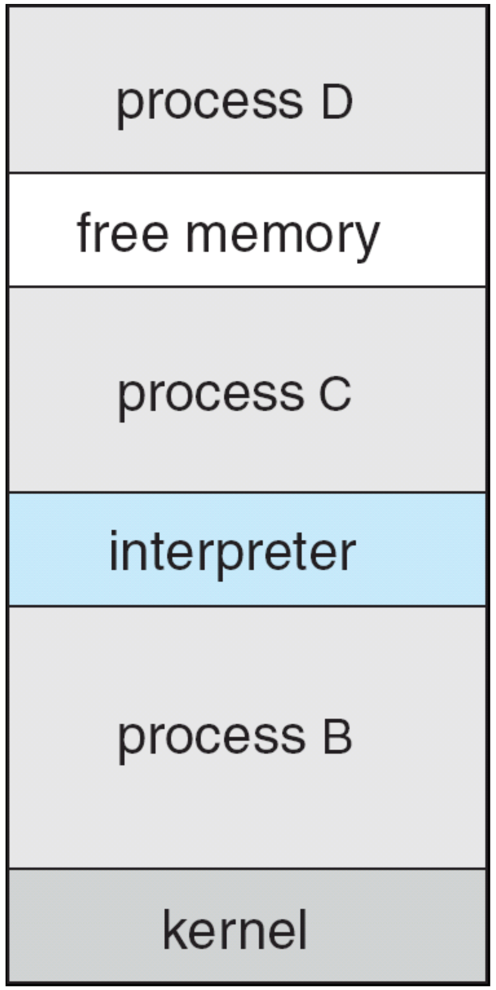


At system startup

running a program

Process control example - FreeBSD

- Unix variant
- Multitasking
- User login -> invoke user's choice of shell
- Shell executes fork() system call to create process
    - Calls exec() to load program into process
    - Shell waits for process to terminate or continues with user commands
- Process exits with:
    - code = 0 – no error
    - code > 0 – error code



2.5 System Programs and utilites

- Most users' view of the operation system is defined by system programs, not the actual system calls
  - Provide a convenient environment for program execution
  - Some of them are simply user interfaces to system calls; others are considerably more complex.
- **File management** - Create, delete, copy, rename, print, dump, list, and generally manipulate files and directories
- **Status information**
  - Provide **simple info** such as date, time, amount of available memory, disk space or number of users
  - May provide **detailed info** such as performance, logging, and debugging information
  - Some systems implement a **registry** - used to store and retrieve configuration information

System Programs (Cont.)

- **File modification**
  - Text editors to create and modify files
  - Special commands to search contents of files or perform transformations of the text (e.g. grep)
- **Programming-language support** - Compilers, assemblers, debuggers and interpreters sometimes provided. May also provide a debugging system (for higher level as well as machine language).
- **Program loading and execution**- provides ability to load a program into main memory, as well as linking it to shared libraries (.dll in windows or .so in linux/unix)
- **Communications** - Provide the **mechanism** for creating virtual connections among processes, users, and computer systems.

System Programs (Cont.)

- **Background Services**
  - These are system programs that are run automatically. Known as **services**, **subsystems**, **daemons**
  - Some run at system startup (or boot) and terminate at some point later.
  - Others run from system boot to shutdown
  - Ex: A network daemon may be listening to connection requests, and then connects them to the appropriate service.
  - Provide other facilities like disk checking, error logging, printing
  - Run in user context, not kernel context

Application programs

- Don’t pertain to system
- Run by users
- Not typically considered part of OS
- Launched by command line, mouse click, finger poke
- e.g. web browsers, email clients, word processors, games, etc.