ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Real-Time Systems for Automation M

2. System Structures

Notice

The course material includes slides downloaded from:

http://codex.cs.yale.edu/avi/os-book/

(slides by Silberschatz, Galvin, and Gagne, associated with Operating System Concepts, 9th Edition, Wiley, 2013)

and

http://retis.sssup.it/~giorgio/rts-MECS.html

(slides by Buttazzo, associated with Hard Real-Time Computing Systems, 3rd Edition, Springer, 2011)

which have been edited to suit the needs of this course.

The slides are authorized for personal use only.

Any other use, redistribution, and any for profit sale of the slides (in any form) requires the consent of the copyright owners.

Previously

- The operating system act as intermediary between applications and hardware
- User applications can request the **services** of the OS through **system calls**
- **Program**: passive entity
- **Process**: program in exectution, active entity, unit of work
  - Each process has a Process ID associated: **PID**

Chapter 2: Operating-System Structures

- Operating System Services
- User and Operating System Interface
- System Calls
- Types of System Calls
- System Programs
- Operating-System Structure
- System Boot

Operating System Services

- Operating systems provide an environment for execution of programs and services to programs and users
- One set of operating-system services provides functions that are helpful to the user:
  - **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 end its execution, either normally or abnormally (indicating error)
  - **I/O operations** - A running program may require I/O. For efficiency and protection, users cannot control I/O devices directly. OS must provide a means to do I/O → it needs to be regulated
  - **File-system manipulation** - Programs need to read and write files and directories, create and delete them, search them, list file information, permission management. The OS must offer these functions

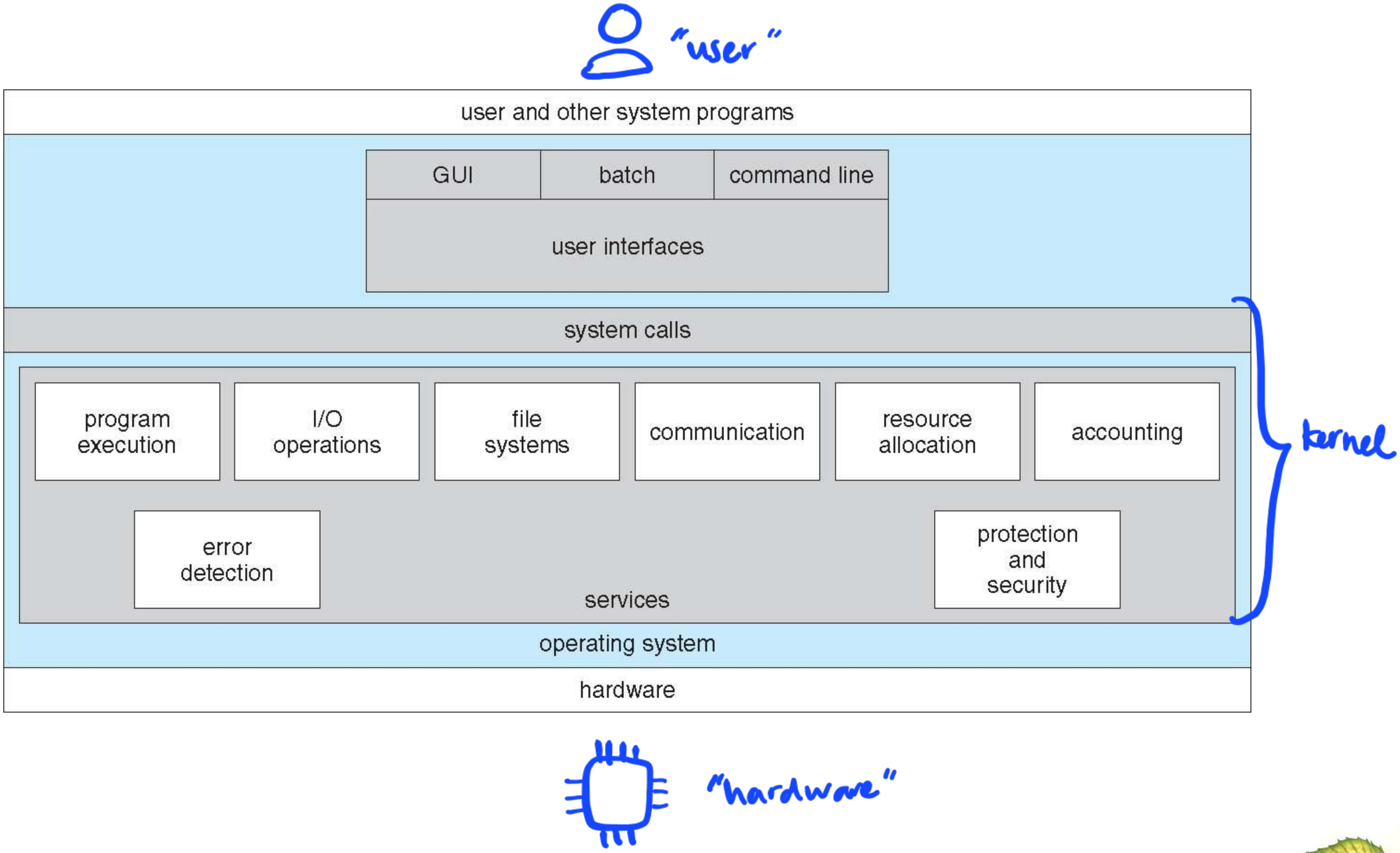
Operating System Services

- **Communications** – Processes may exchange information on the same computer or between computers over a network
  - Communications may be via either:
    - same "PID" – shared memory (in logical address space of the processes)
    - send/receive – message passing (in the kernel, packets moved by the OS which offers send/receive operations to the processes)
- **Error detection** – OS needs to be constantly aware of errors
  - May occur in the CPU and memory hardware, in I/O devices, in user program
  - For each type of error, OS should take the appropriate action to ensure correct and consistent computing
  - Debugging facilities: OS provides the environment for a controlled, "interruptable" execution for testing; or core/memory dump facilities for post-mortem analysis.

Operating System Services

- Another set of OS functions exists for ensuring the **efficient operation** of the system itself via resource sharing
  - **Resource allocation -** When multiple users or multiple jobs running concurrently, resources must be allocated to each of them
    - Many types of resources - Some (such as CPU cycles, main memory, and file storage) may have special allocation code, others (such as I/O devices) may have general request and release code
  - **Logging -** To keep track of which users use how much and what kinds of computer resources (for accounting or statistics)
  - **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
    - If a system is to be protected and secure, precautions must be instituted throughout it. A chain is only as strong as its weakest link.

A View of Operating System Services



user and other system programs

GUI | batch | command line

user interfaces

system calls

program execution | I/O operations | file systems | communication | resource allocation | accounting

error detection | protection and security

services

operating system

hardware

User Operating System Interface - CLI

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

  it makes the user easier to call system calls

  - Sometimes implemented in kernel, sometimes by systems program
  - Sometimes multiple flavors implemented – **shells**

    different ways to interact

  - Primarily fetches a command from user, interprets and executes it

    type command

    OS read

    - built-in commands
    - external executions: in foreground (default) or background (using symbol **&**)

      usually comes from the library

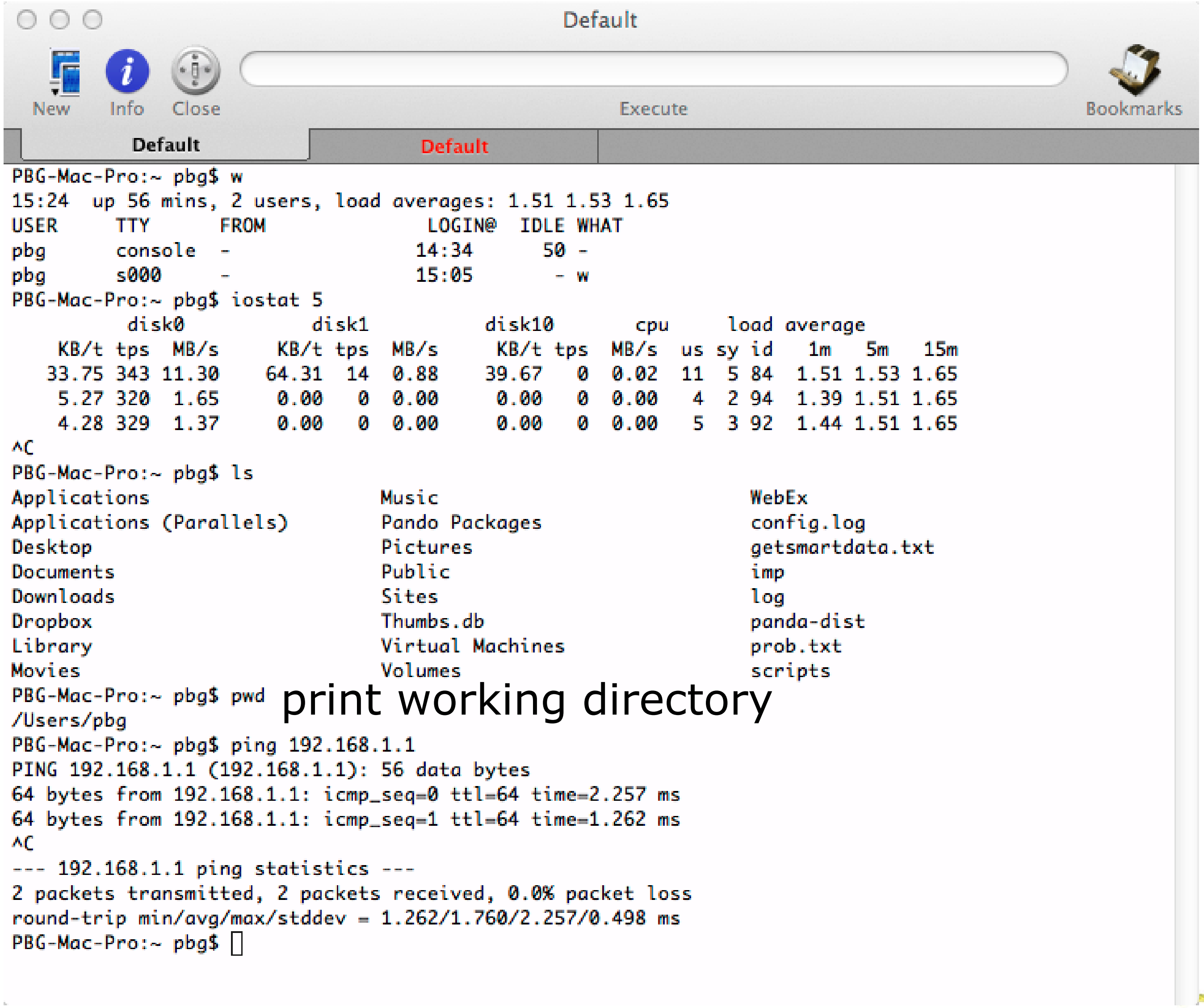
Bourne Shell Command Interpreter

who

i/o stats

Ctrl+c

list



```
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$ 
```

print working directory

Basic Linux CLI

how the user interacts that issues traps

print smth

process status

password

concatenation

MANUAL

- Sample commands: echo, ls, cd, ps, pwd, cat, less, **man**
- File system: /bin, /usr/bin, /mount, /usr, /home, …
- Commands pipeline: ls /usr/bin | less

  show less message
- File creation: > file1

  write smth
- Output redirect: ls /usr/bin > output (overwrite); ls /usr/bin >> output (append)
- Background execution: sleep 5 &

create an empty file

the command doesn't do anything until the process finished

read the output and overwrite the file

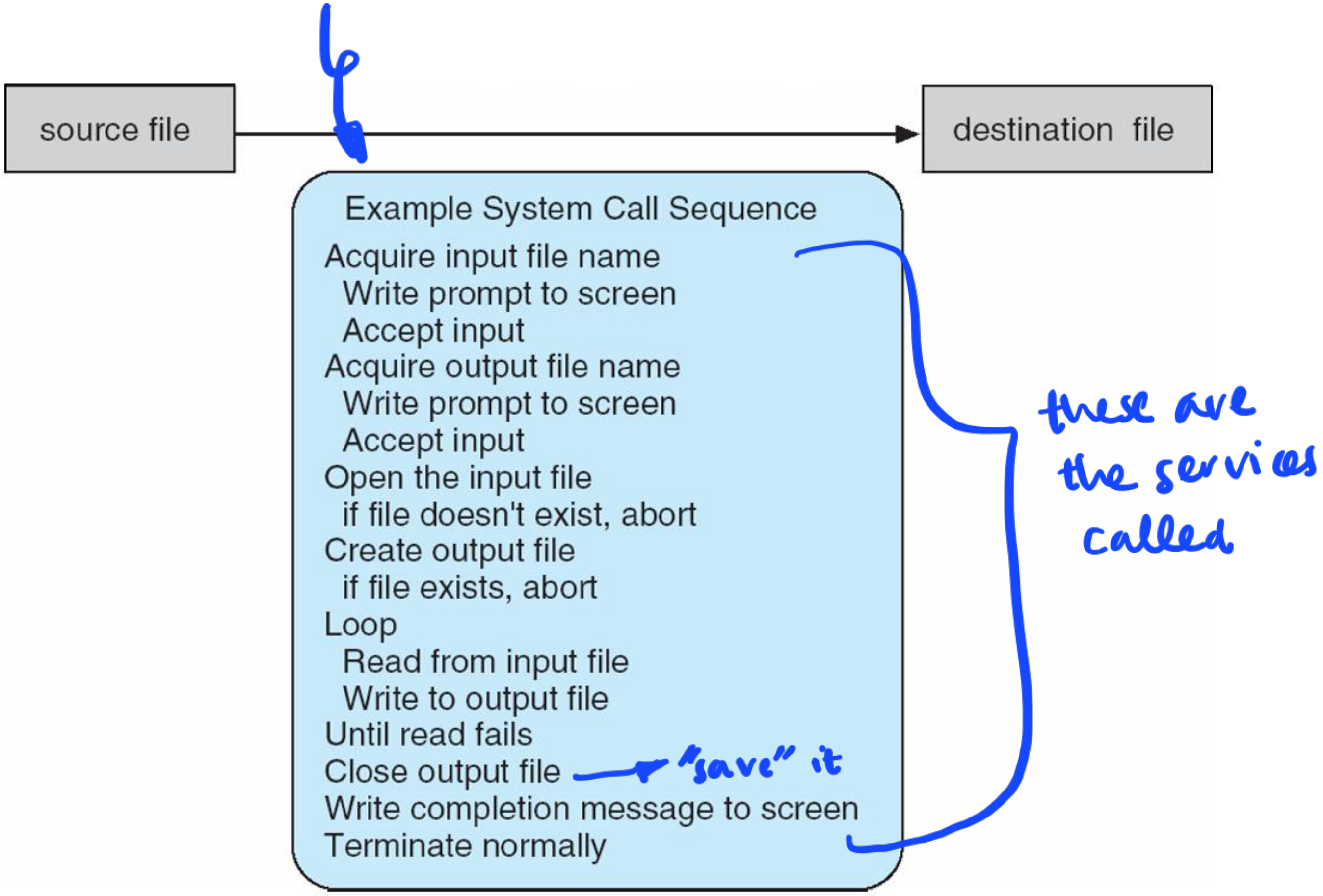
System Calls

→ instructions

- Programming interface to the services provided by the OS
- Example: *copy the contents of one file to another file*

Example of System Calls

- System call sequence to copy the contents of one file to another file



Example System Call Sequence

```
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
```

"save" it

these are the services called

System Calls

- Programming interface to the services provided by the OS

how the service is written

- Typically written in a high-level language (C or C++). Some low-level tasks in assembly-language

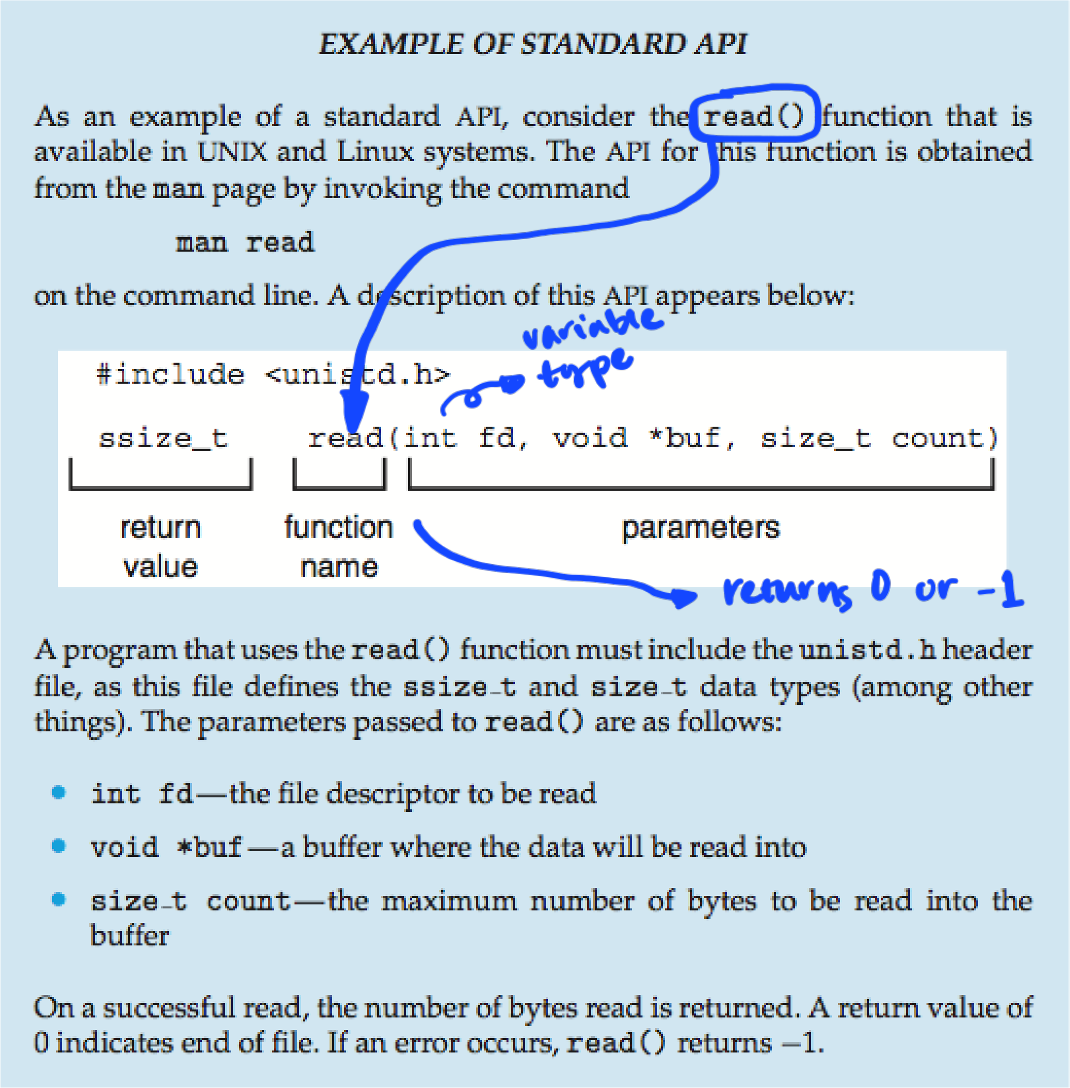
to avoid direct system calls use

- Mostly accessed by programs via a high-level **Application Program Interface** **(API)** rather than direct system call use
  - E.g., in (most) Linux distro, you call open(…), not sys_open(…)

- Three most 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)

- Why use APIs rather than system calls?
  - portability ✓
  - simplicity ✓

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:

variable type

```
#include <unistd.h>

ssize_t    read(int fd, void *buf, size_t count)
```

return value — function name — parameters

returns 0 or -1

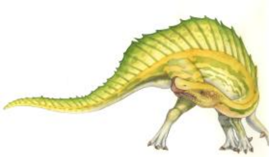
A program that uses the read() function must include the unistd.h header file, as this file defines the ssize_t and size_t data types (among other things). The parameters passed to read() are as follows:

- 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.

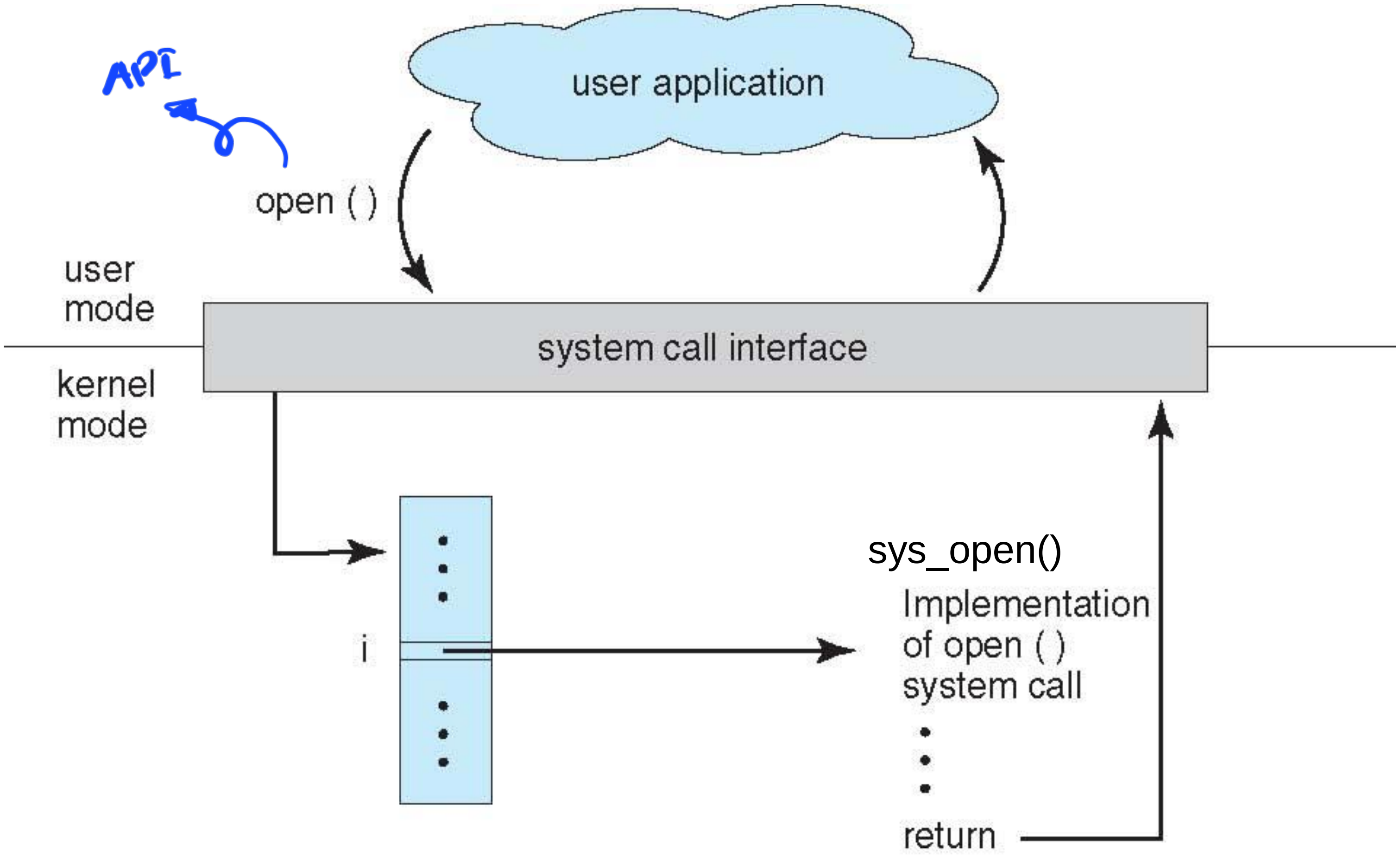
System Call Implementation

- Typically, a number associated with each system call
  - **System-call interface** maintains a table indexed according to these numbers
- The system call interface invokes intended system call in OS kernel and returns status of the system call and any return values
- The caller need know nothing about how the system call is implemented
  - Just needs to obey API and understand what OS will do as a result 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)



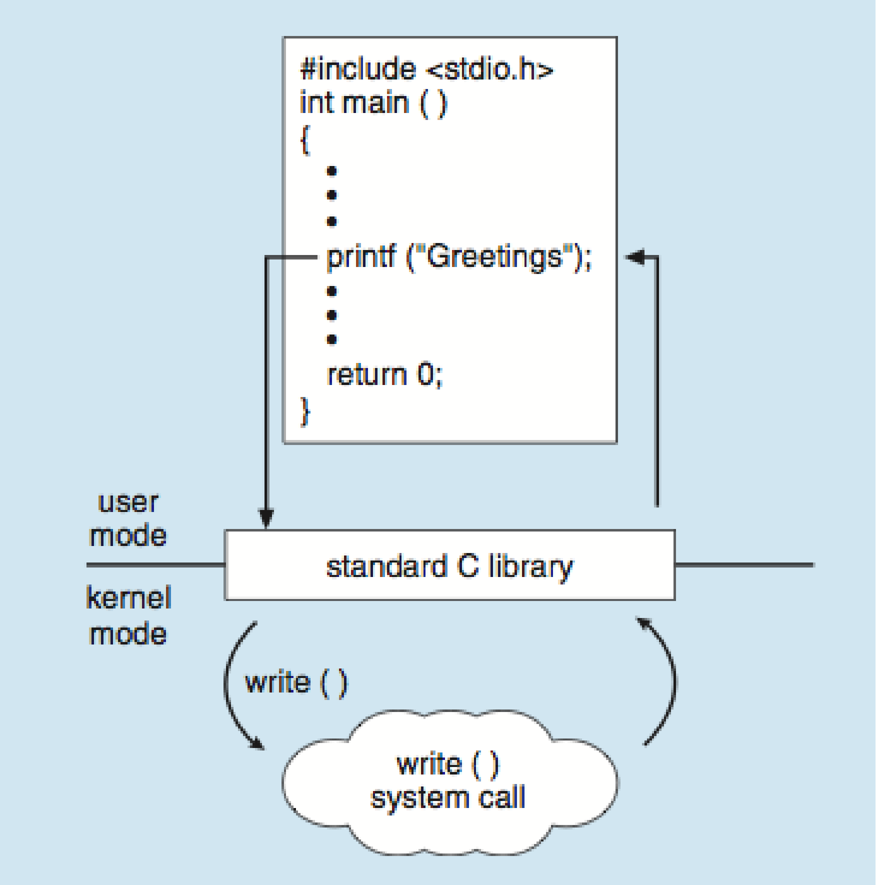
API – System Call – OS Relationship

use a trap



Standard C Library Example

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



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

Quizzes

- An operating system can have more than one command interpreter which a user can choose. ✓
- Every operating system has a graphical user interface. ✗
- In the Linux shell, it is possible to concatenate commands, so that the output of a command becomes the input of another command. ✓
- Reading user-input from the keyboard requires system calls. ✓
- Typically, every user process maintains a table of system calls. ✗ → instead, it's in the KERNEL which is in the OS
- To use system calls correctly in a program, one needs to know how the systems calls are implemented. ✗ → instead, use API

Types of System Calls

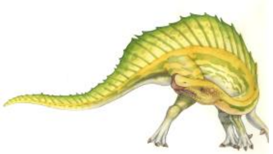
API : Fork

API : exit

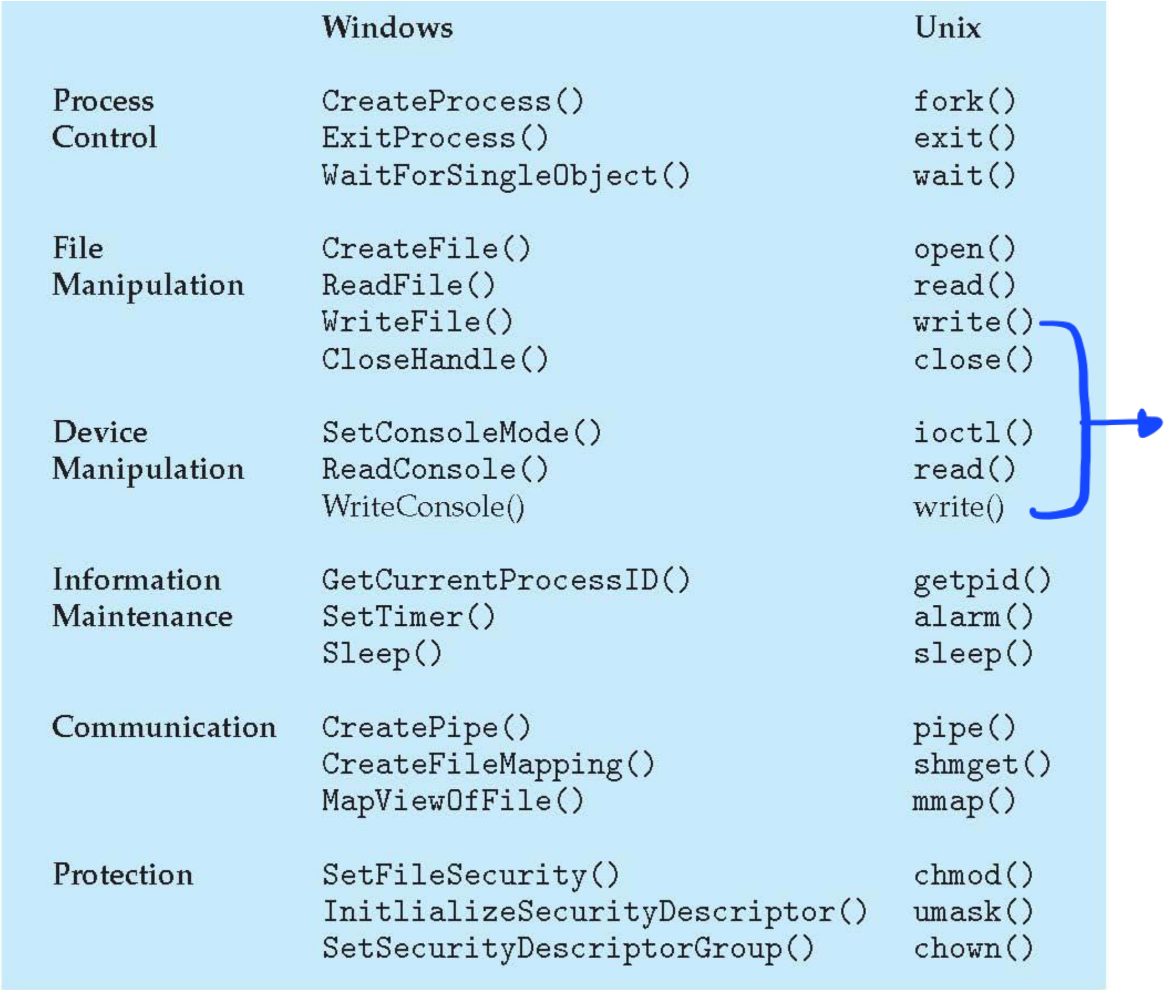
raises an error

- Process control
  - **create process, terminate process** (i.e., explicitly ending a specific process)
  - **end** (i.e., normal end of execution), **abort** (i.e., exceptions cause OS intervention)
  - **load** (“turn a program into a process”: create the memory image of the process), **execute** (CPU dispatching)
  - **get process attributes, set process attributes** (e.g., process identifier, priority)
  - **wait for time, wait event** (e.g., termination of a child process, data ready to be read in a pipe), **signal event**
  - **allocate** and **free** memory

  - Issues:
    - Dump memory if error
    - **Debugger** for determining **bugs, single step** execution
    - Background/foreground execution
    - **Locks** for managing access to shared data between processes

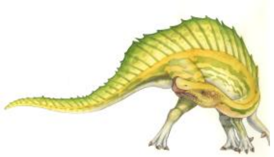


Examples of Windows and Unix System Calls



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

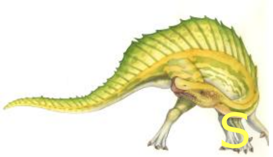
UNIX
treats
I/O devices
like files



Types of System Calls

- File management
  - create file, delete file
  - open, close file
  - read, write, reposition
  - get and set file attributes

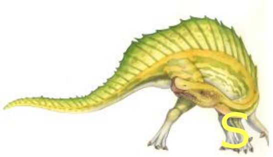
- Device management
  - request device, release device
    - Exclusive use. Deadlock → two processes want to access the same resource
  - read, write, reposition
    - Physical vs. abstract devices. Similarity between files and devices
  - get device attributes, set device attributes
  - logically attach or detach devices



Types of System Calls

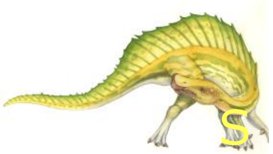
- Information maintenance
  - get time or date, set time or date
  - get system data, set system data
  - get and set process, file, or device attributes

- Communications
  - create, delete communication connection
    - open, close, accept, wait
  - send, receive messages if **message passing model** to **host name** or **process name**
    - From **client** to **server**
  - **Shared-memory model** create and gain access to memory regions
    - Create, attach



Types of System Calls

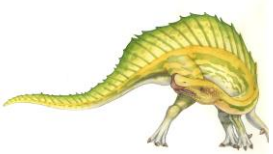
- Protection (control access to resources)
  - Get and set permissions
  - Allow and deny user access



System Programs (Utilities)

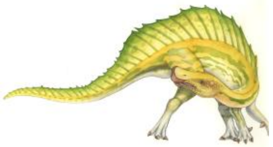
use kernel services as an intermediate

- System programs provide a convenient environment for program development and execution. They can be divided into:
  - File manipulation (e.g., file manager)
  - Status information (e.g., task manager)
  - File editing (graphical or command-line e.g., vi) & content search (e.g., find, which,…)
  - Programming-language support (JVM, gcc,…)
  - Program loading and execution
  - Communications (e.g., web browsers)
  - Background services (daemons)
  - Application programs
- Most users' view of the operation system is defined by system programs, not the actual system calls
- Utilities use kernel services but are not part of it



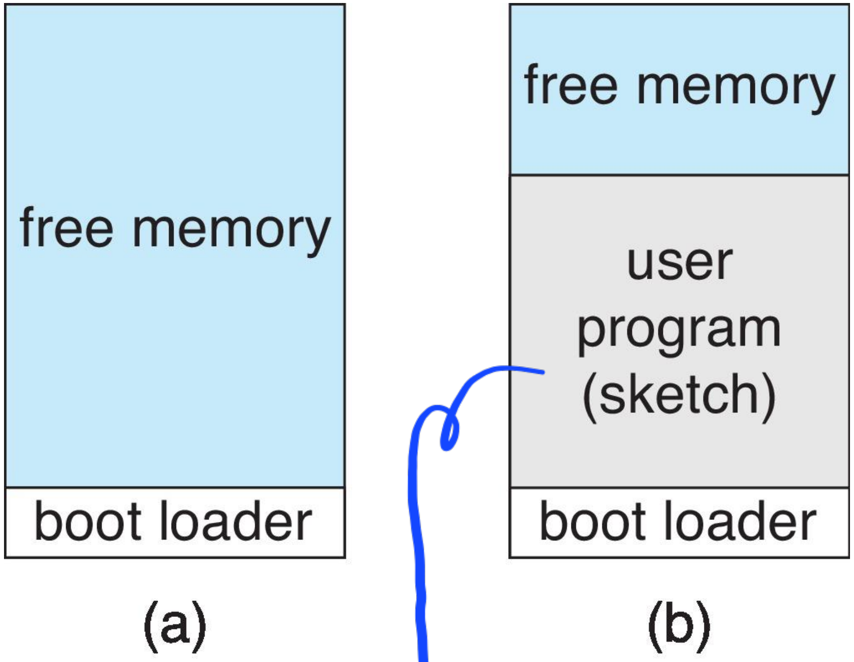
Operating System Structure

- General-purpose OS is very large program
- Various ways to structure one as follows
  - (knowing all details is less and less important nowadays)



Example: Arduino

- Single-tasking → only one thing at a time
- No operating system
- Programs (sketch) loaded via USB into flash memory
- Single memory space
- Boot loader loads program
- Program exit -> shell reloaded



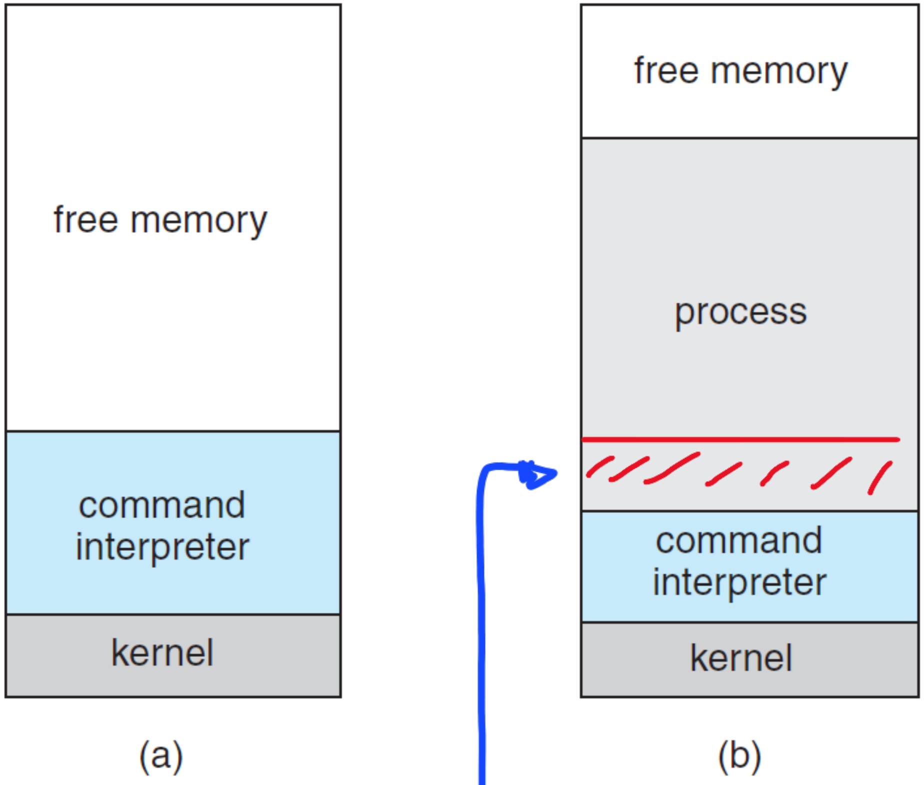
At system startup

running a program

uploaded by USB

Example: MS-DOS

- Single-tasking
- Interpreter launched at start-up
- Does not create new process.
  - Overwrites the interpreter in the memory
- Output of the program saved in memory
- At the end, the interpreter is reloaded from disk into memory
- Output code made availabe



At system startup

running a program

UNIX → layers

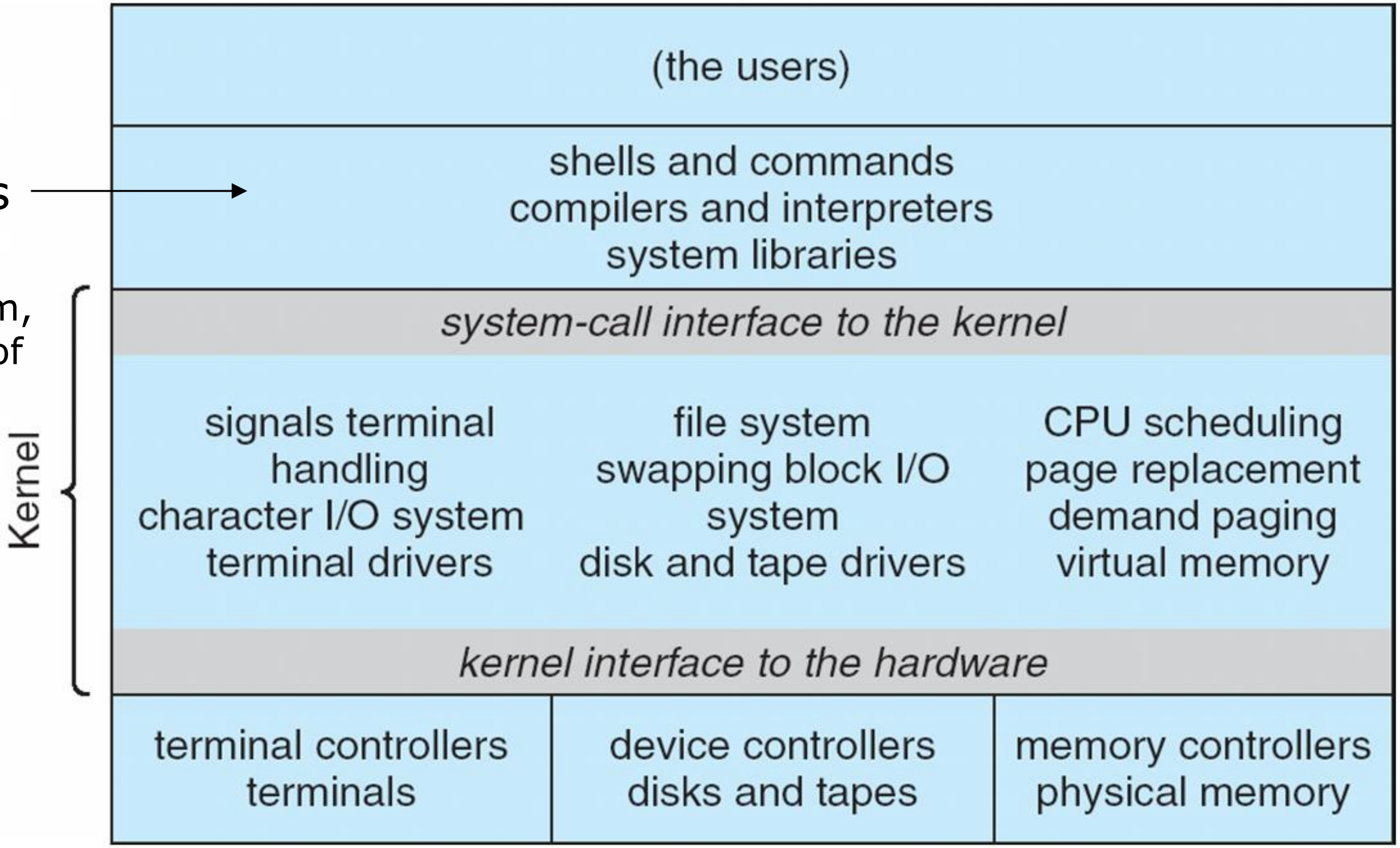
- UNIX – limited by hardware functionality, the original UNIX operating system had limited structuring. The UNIX OS consists of two separable parts
  - Systems programs
  - The kernel
    - Consists of everything below the system-call interface and above the physical hardware
    - Provides the file system, CPU scheduling, memory management, and other operating-system functions; a large number of functions for one level

Traditional UNIX System Structure

utilities

one big program, manages a lot of different things

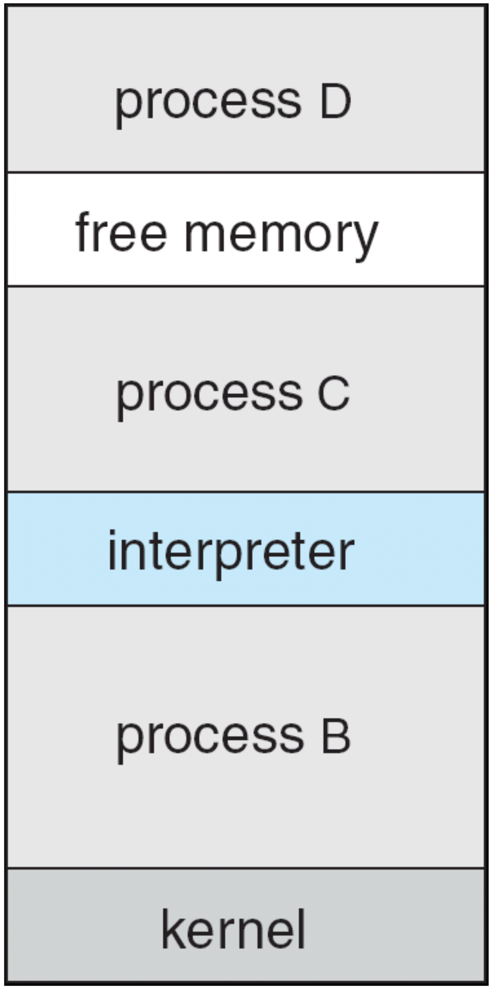
the more we need to add functionalities to the OS, the more this program gets complicated



Example: FreeBSD

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

remains in the memory


process D
free memory
process C
interpreter
process B
kernel

Modules

- Most modern operating systems (e.g., VxWorks) implement **loadable kernel modules**
  - Uses object-oriented approach
  - Each core component is separate
  - Each talks to the others over known interfaces
  - Each is loadable as needed within the kernel

→ more "efficient" kernel that only loads "necessary" modules to the kernel

Quizzes

- What system calls must be executed by a shell in order to start a new process? fork()
- [✓ [T]] [F] The operating system is always stored in the hard disk (storage device)
- [T] [✓ [F]] Utilities are loadable kernel modules

→ additional functions to interact w/ the kernel