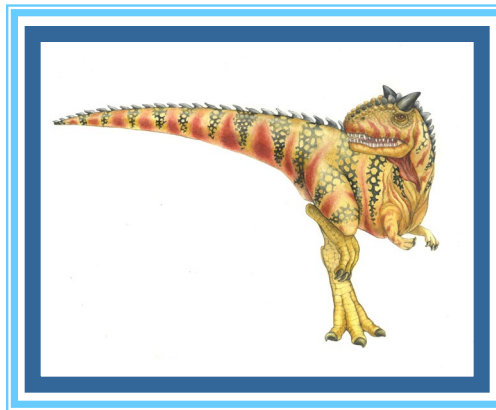


# Chapter 2 (1)

---





# Outline

---

- Operating System Services
- User and Operating System-Interface
- System Calls
- System Services
- Linkers and Loaders
- Why Applications are Operating System Specific
- Design and Implementation
- Operating System Structure
- Building and Booting an Operating System
- Operating System Debugging





# Objectives

---

- Identify services provided by an operating system
- Illustrate how system calls are used to provide operating system services
- Compare and contrast monolithic, layered, microkernel, modular, and hybrid strategies for designing operating systems
- Illustrate the process for booting an operating system
- Apply tools for monitoring operating system performance
- Design and implement kernel modules for interacting with a Linux kernel





# 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)**, **touch-screen**, **Batch**
  - **Program execution** - The system must be able to load a program into memory and to run that program, end execution, either 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
  - **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.





# Operating System Services (1)

- One set of operating-system services provides functions that are helpful to the user (Cont.):
  - **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 moved by the OS)
  - **Error detection** – OS needs to be constantly aware of possible 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 can greatly enhance the user's and programmer's abilities to efficiently use the system





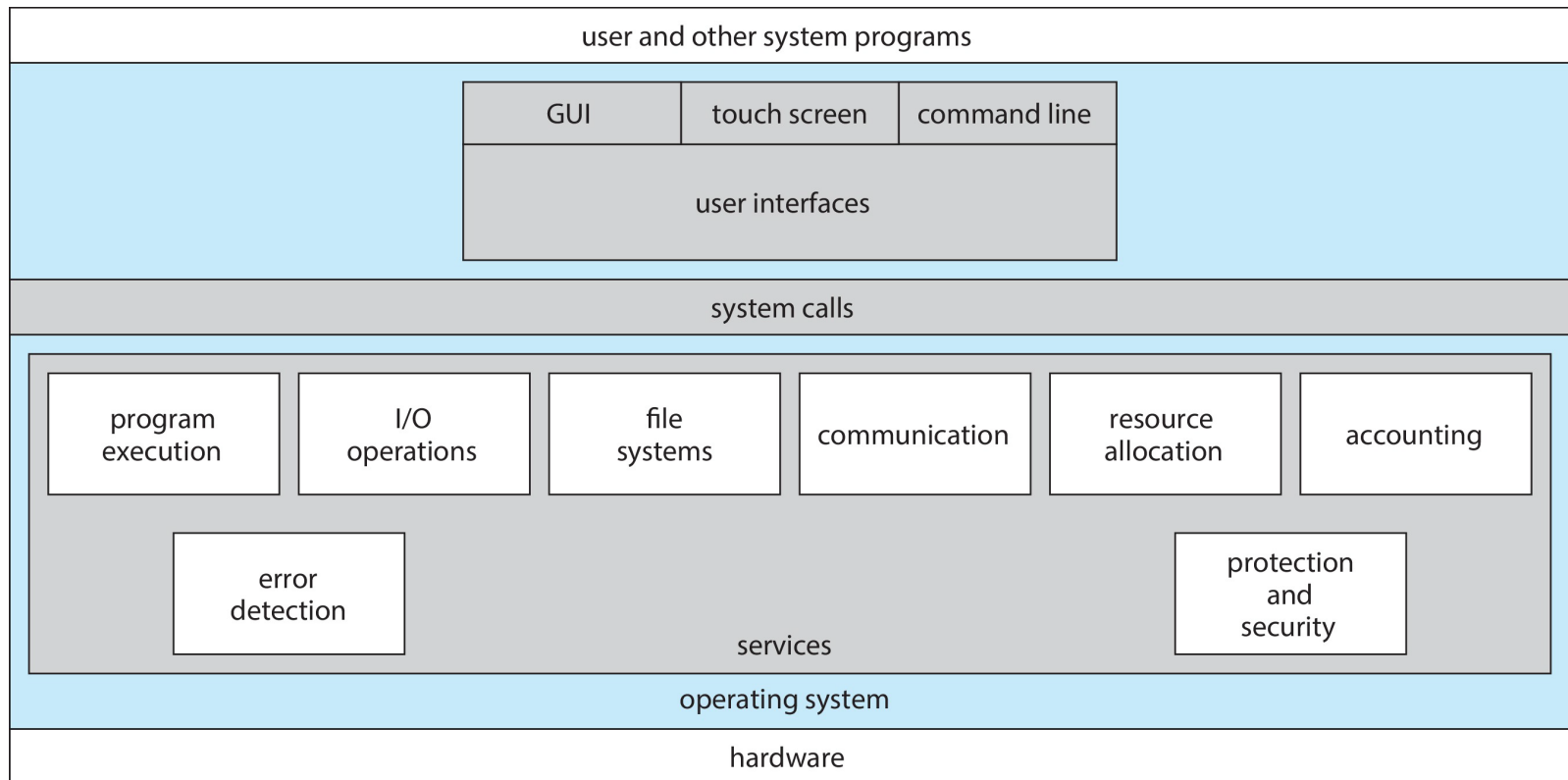
# Operating System Services (2)

- 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 - CPU cycles, main memory, file storage, I/O devices.
  - **Logging** - To keep track of which users use how much and what kinds of computer resources
  - **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





# A View of Operating System Services





# Command Line interpreter

---

- CLI allows direct command entry
- Sometimes implemented in kernel, sometimes by systems program
- Sometimes multiple flavors implemented – **shells**
- Primarily fetches a command from user and executes it
- Sometimes commands built-in, sometimes just names of programs
  - If the latter, adding new features doesn't require shell modification





# Bourne Shell Command Interpreter

```
1. root@r6181-d5-us01:~ (ssh)
X root@r6181-d5-u...  %1 X ssh  %2 X root@r6181-d5-us01...  %3

Last login: Thu Jul 14 08:47:01 on ttys002
iMacPro:~ pbg$ ssh root@r6181-d5-us01
root@r6181-d5-us01's password:
Last login: Thu Jul 14 06:01:11 2016 from 172.16.16.162
[root@r6181-d5-us01 ~]# uptime
 06:57:48 up 16 days, 10:52,  3 users,  load average: 129.52, 80.33, 56.55
[root@r6181-d5-us01 ~]# df -kh
Filesystem                Size      Used Avail Use% Mounted on
/dev/mapper/vg_ks-lv_root
                        50G       19G   28G   41% /
tmpfs                      127G       520K   127G    1% /dev/shm
/dev/sda1                  477M       71M   381M   16% /boot
/dev/dssd0000              1.0T      480G   545G   47% /dssd_xfs
tcp://192.168.150.1:3334/orangefs
                        12T       5.7T   6.4T   47% /mnt/orangefs
/dev/gpfs-test             23T       1.1T   22T    5% /mnt/gpfs
[root@r6181-d5-us01 ~]#
[root@r6181-d5-us01 ~]# ps aux | sort -nrk 3,3 | head -n 5
root      97653 11.2   6.6 42665344 17520636 ?    S<Ll  Jul13 166:23 /usr/lpp/mmfs/bin/mmfstd
root      69849   6.6   0.0      0      0 ?        S    Jul12 181:54 [vpthread-1-1]
root      69850   6.4   0.0      0      0 ?        S    Jul12 177:42 [vpthread-1-2]
root       3829   3.0   0.0      0      0 ?        S    Jun27 730:04 [rp_thread 7:0]
root       3826   3.0   0.0      0      0 ?        S    Jun27 728:08 [rp_thread 6:0]
[root@r6181-d5-us01 ~]# ls -l /usr/lpp/mmfs/bin/mmfstd
-r-x----- 1 root root 20667161 Jun  3  2015 /usr/lpp/mmfs/bin/mmfstd
[root@r6181-d5-us01 ~]#
```





# User Operating System Interface - GUI

---

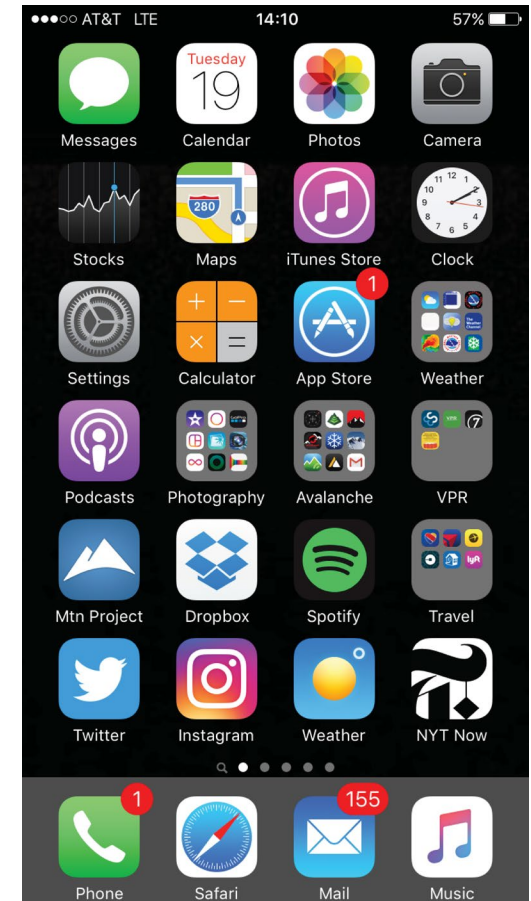
- User-friendly **desktop** metaphor interface
  - Usually mouse, keyboard, and monitor
  - **Icons** represent files, programs, actions, etc
  - Various mouse buttons over objects in the interface cause various actions (provide information, options, execute function, open directory (known as a **folder**))
  - Invented at Xerox PARC
- Many systems now 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)





# Touchscreen Interfaces

- Touchscreen devices require new interfaces
  - Mouse not possible or not desired
  - Actions and selection based on gestures
  - Virtual keyboard for text entry
- Voice commands







# System Calls

---

- Programming interface to the services provided by the OS
- Typically written in a high-level language (C or C++)
- Mostly accessed by programs via a high-level **Application Programming Interface (API)** rather than direct system call use
- 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), and Java API for the Java virtual machine (JVM)

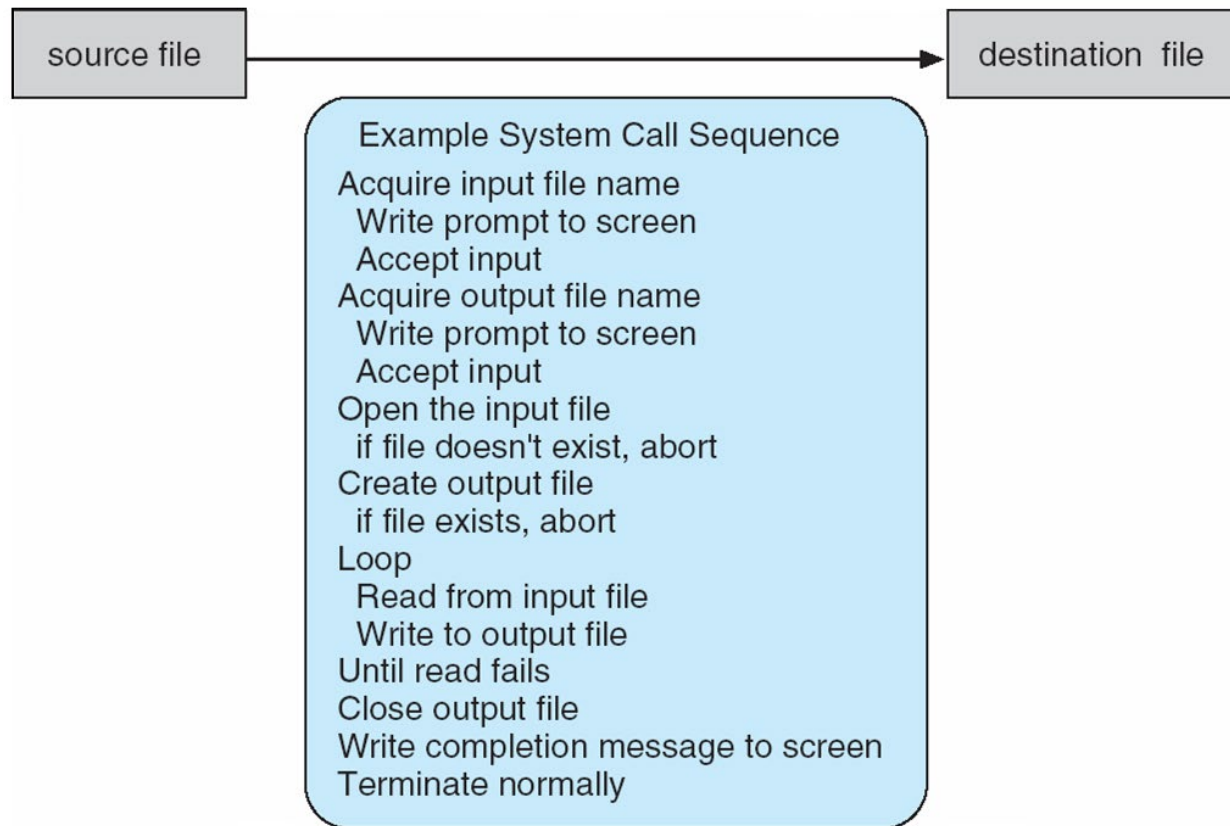
Note that the system-call names used throughout this text are generic





# Example of System Calls

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





# 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>

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

return value	function name	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 into which the data will be read
- `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

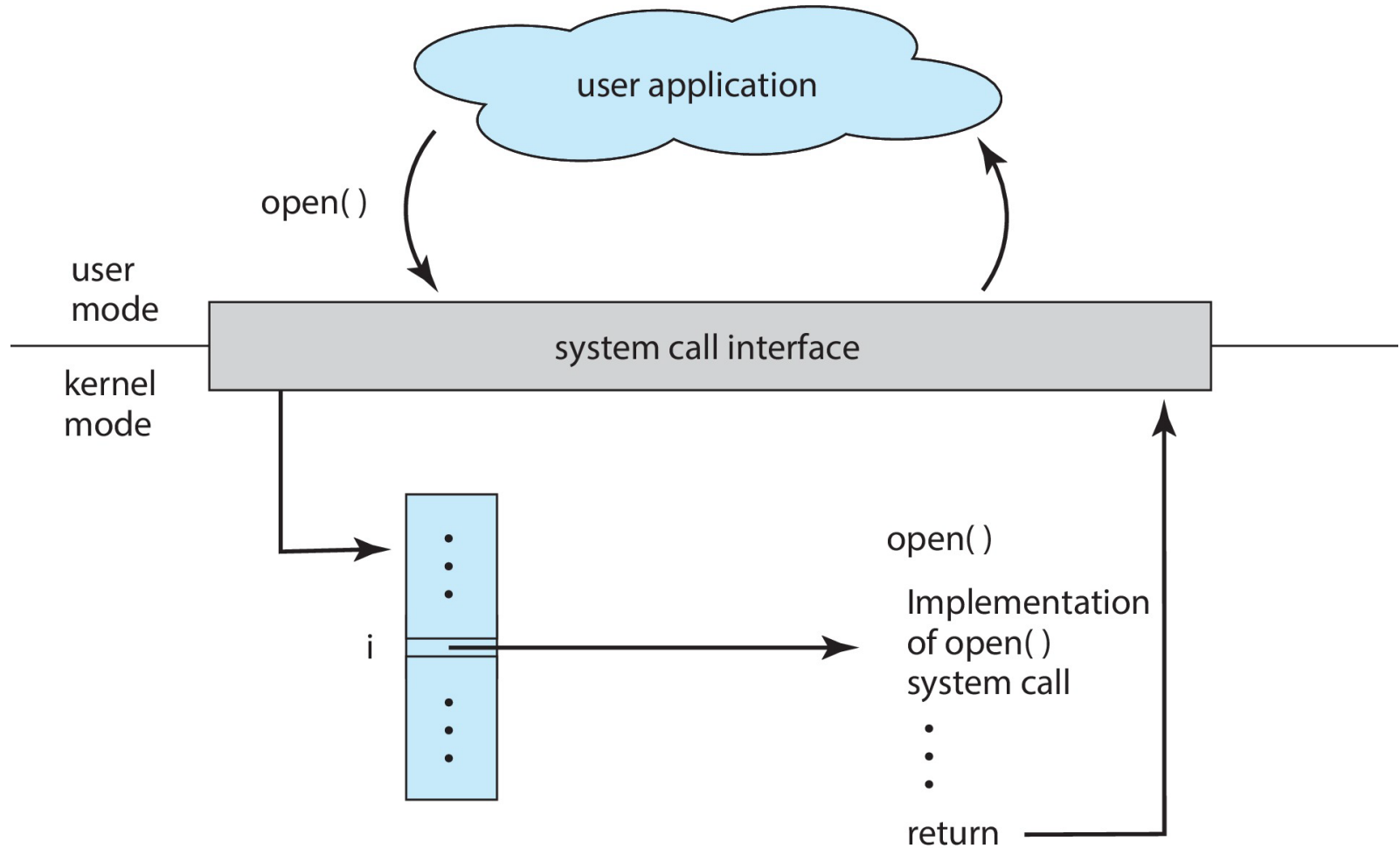
---

- Typically, a number is 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
- 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





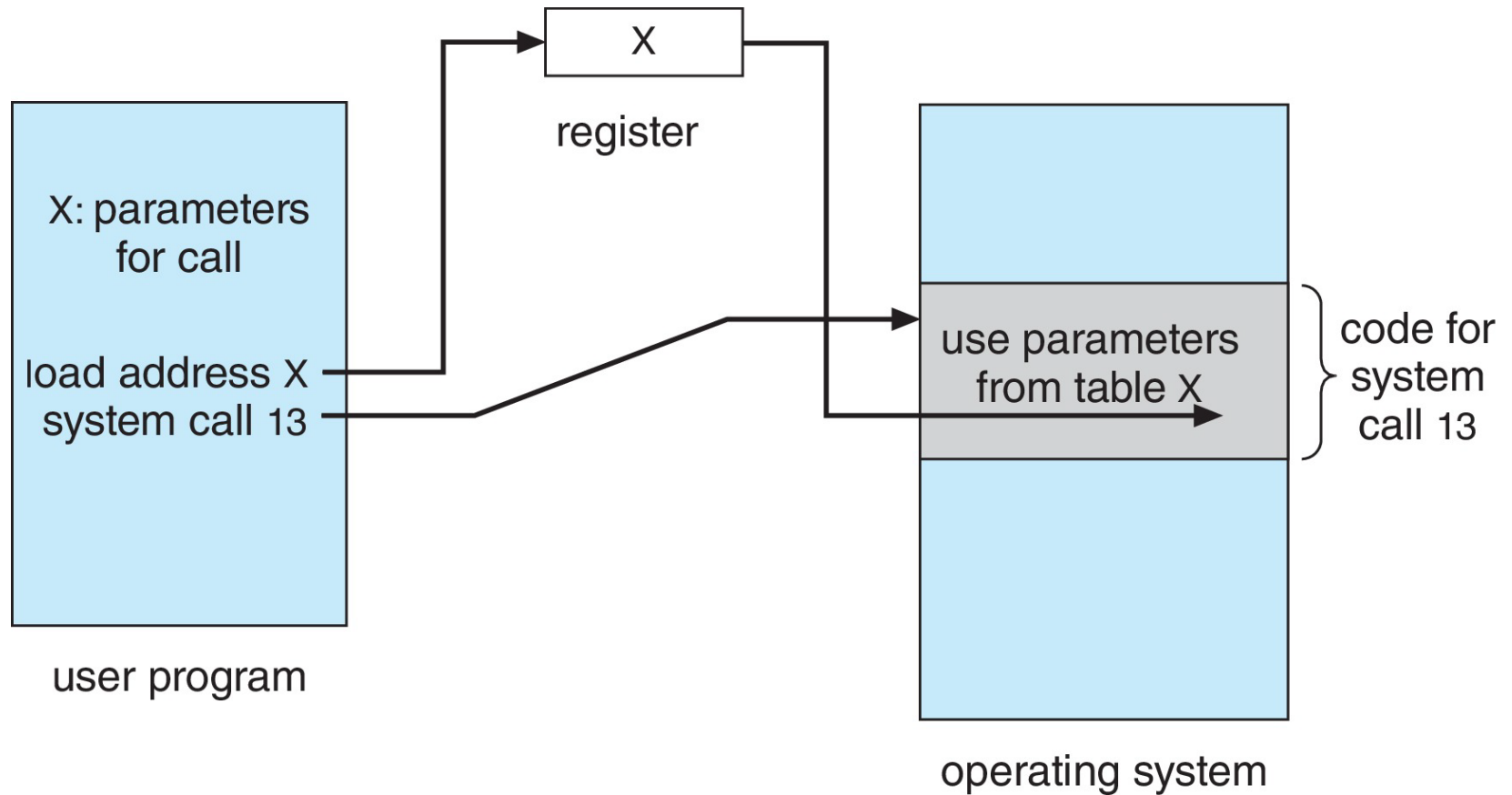
# System Call Parameter Passing

- Often, more information is required than simply identity of desired system call
  - Exact type and amount of information vary according to OS and call
- Three general methods used to pass parameters to the OS
  - Simplest: pass the parameters in registers
    - ▶ In some cases, may be more parameters than registers
  - Parameters stored in a block, or table, in memory, and address of block passed as a parameter in a register
    - ▶ This approach taken by Linux and Solaris
  - Parameters placed, or **pushed**, onto the **stack** by the program and **popped** off the stack by the operating system
  - Block and stack methods do not limit the number or length of parameters being passed





# Parameter Passing via Table





# Types of System Calls

---

- Process control
  - create process, terminate process
  - end, 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, single step** execution
  - **Locks** for managing access to shared data between processes





# Types of System Calls (1)

---

- File management
  - create file, delete file
  - open, close file
  - read, write, reposition
  - get and set file attributes
- Device management
  - request device, release device
  - read, write, reposition
  - get device attributes, set device attributes
  - logically attach or detach devices
- Information maintenance
  - get time or date, set time or date
  - get system data, set system data
  - get and set process, file, or device attributes





# Types of System Calls (2)

---

- Communications
  - create, delete communication connection
  - 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
  - transfer status information
  - attach and detach remote devices
- Protection
  - Control access to resources
  - Get and set permissions
  - Allow and deny user access





# Examples of Windows and Unix System Calls

## EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS

The following illustrates various equivalent system calls for Windows and UNIX operating systems.

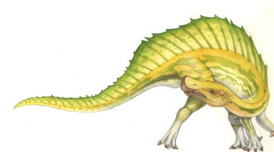
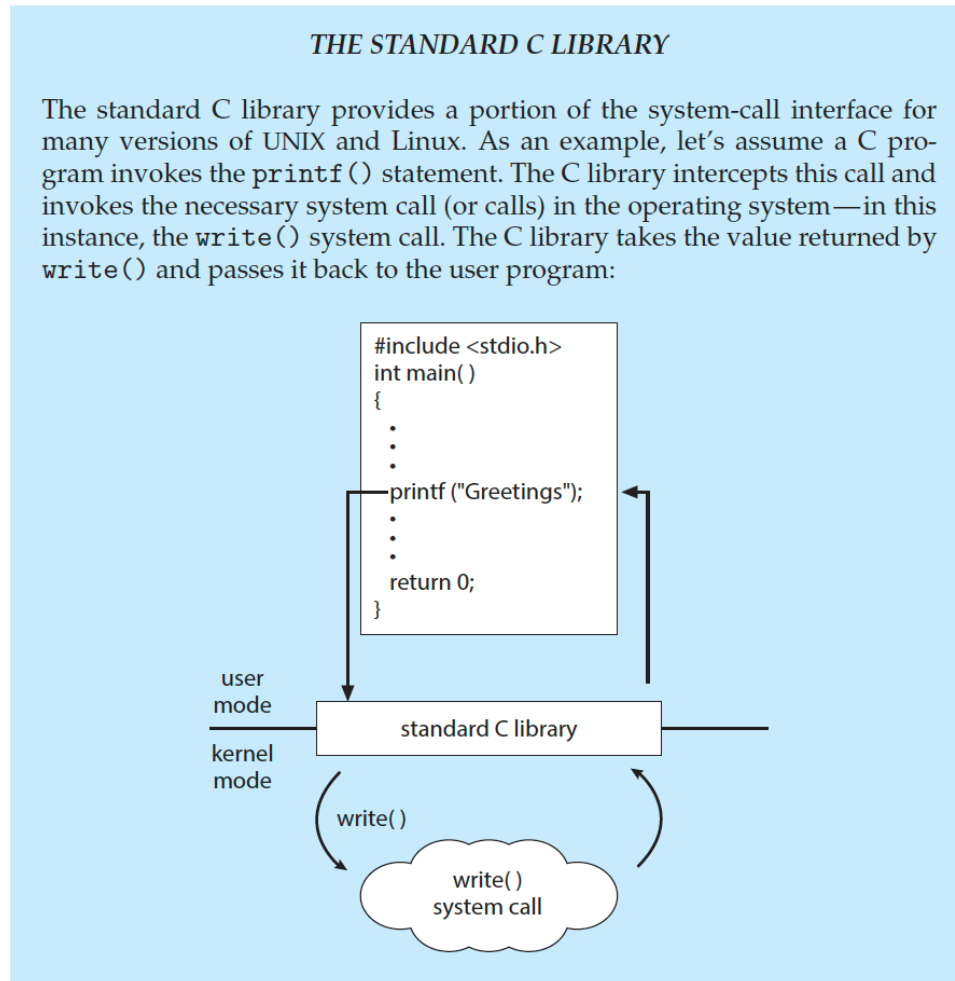
	Windows	Unix
<b>Process control</b>	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
<b>File management</b>	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
<b>Device management</b>	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
<b>Information maintenance</b>	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
<b>Communications</b>	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shm_open() mmap()
<b>Protection</b>	SetFileSecurity() InitializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()





# Standard C Library Example

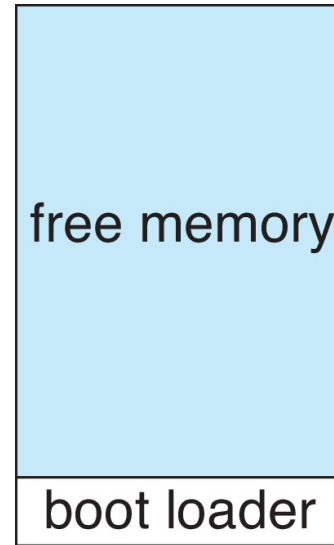
- C program invoking `printf()` library call, which calls `write()` system call





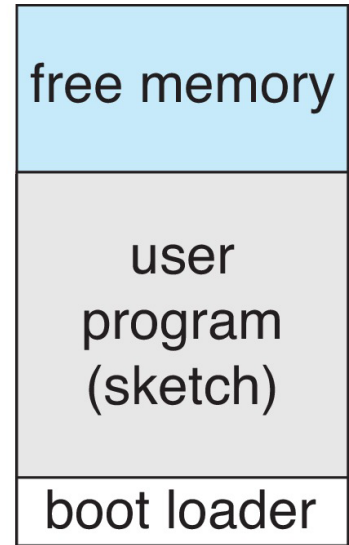
# Example: Arduino

- Single-tasking
- No operating system
- Programs (sketch) loaded via USB into flash memory
- Single memory space
- Boot loader loads program
- Program exit -> shell reloaded



(a)

At system startup



(b)

running a program

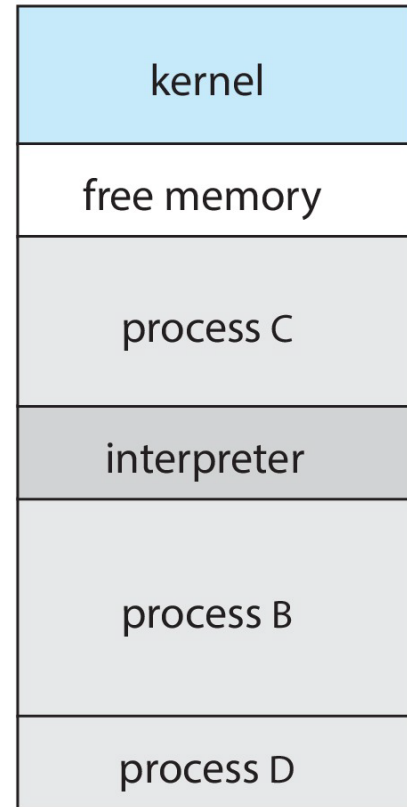




# Example: FreeBSD

- 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 = 0` – no error
  - `code > 0` – error code

high  
memory



low  
memory





# System Services (1)

---

- System programs provide a convenient environment for program development and execution. They can be divided into
  - File manipulation
  - Status information sometimes stored in a file
  - Programming language support
  - Program loading and execution
  - Communications
  - Background services
  - Application programs
- Most users' view of the operating system is defined by system programs, not the actual system calls





# System Services (2)

---

- Provide a convenient environment for program development and 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**
  - Some ask the system for info - date, time, amount of available memory, disk space, number of users
  - Others provide detailed performance, logging, and debugging information
  - Typically, these programs format and print the output to the terminal or other output devices
  - Some systems implement a **registry** - used to store and retrieve configuration information





# System Services (3)

---

- **File modification**

- Text editors to create and modify files
- Special commands to search contents of files or perform transformations of the text

- **Programming-language support** - Compilers, assemblers, debuggers and interpreters sometimes provided

- **Program loading and execution**- Absolute loaders, relocatable loaders, linkage editors, and overlay-loaders, debugging systems for higher-level and machine language

- **Communications** - Provide the mechanism for creating virtual connections among processes, users, and computer systems
  - Allow users to send messages to one another's screens, browse web pages, send electronic-mail messages, log in remotely, transfer files from one machine to another





# System Services (4)

---

## ■ Background Services

- Launch at boot time
  - ▶ Some for system startup, then terminate
  - ▶ Some from system boot to shutdown
- Provide facilities like disk checking, process scheduling, error logging, printing
- Run in user context not kernel context
- Known as **services**, **subsystems**, **daemons**

## ■ Application programs

- Don't pertain to system
- Run by users
- Not typically considered part of OS
- Launched by command line, mouse click, finger poke



# Take a Break

---

