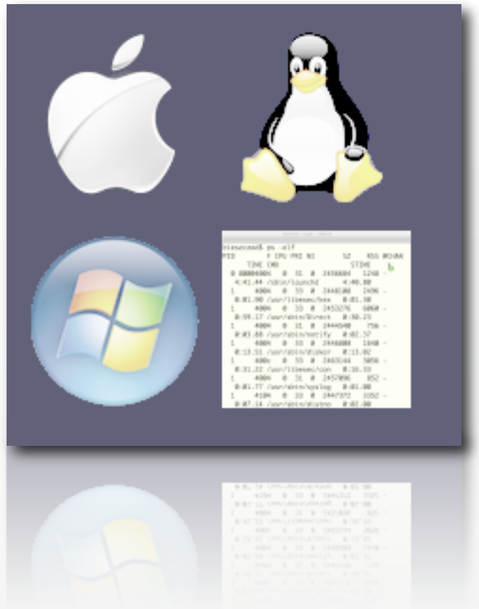


Lecture 2: Operating-System Structures

COMP362 Operating Systems
Prof. AJ Bieszczad

Outline

- System Startup
- Operating System Services
  - For users
  - For resource sharing
- User Operating System Interface
- System Calls
- Types of System Calls
- System Programs
- Operating System Design and Implementation
- Operating System Structure

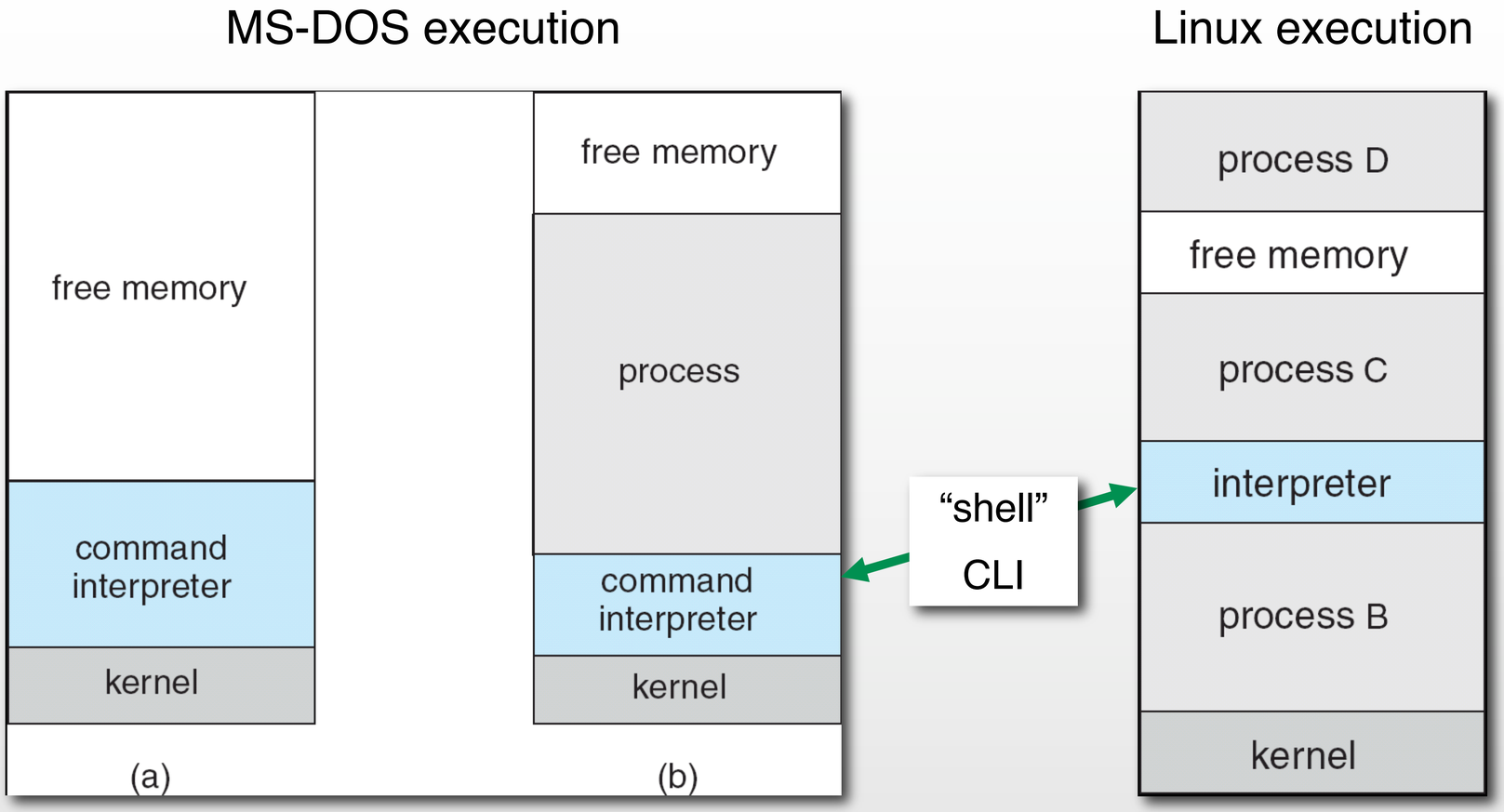
OS Startup

- Operating systems are designed to run on a variety of machines
  - Therefore, the system must be configured for each specific computer
  - During installation of the OS, a “system configuration” program obtains information concerning the specific configuration of the hardware system
- Before any operating system can be loaded and started, the computer needs to go through a **bootstrapping** process; **boot** for short.
  - When system is powered up, execution starts at a fixed memory location in NVRAM (non-volatile RAM) that holds code provided by the computer vendor called **firmware:**
    - **BIOS** (**Basic Input-Output System**) on older computers, or
    - **UEFI** (**Unified Extensible Firmware Interface**) intended to replace BIOS on modern computers.
  - After power-on self-tests (**POST**), the code that is called **bootstrap** (or **boot**) **loader** tests and initializes the hardware and then following a **boot sequence** loads an operating system
    - boot sequence is configured in BIOS or UEFI
    - all devices in the boot sequence are examined in sequence until a valid operating system is found and loaded.
  - On some computers, boot sequence can be overridden during the boot by invoking a **boot menu**, and then selecting a device to boot from disregarding the boot sequence
    - for example, it can be done on Dell computers by pressing the F12 key (as used in the lab)

OS Startup *(cont.)*

- The firmware on UEFI-based system supports loading an operating systems from any location on any boot device.
- In contrast, on older BIOS-based systems, the operating system is always loaded from the pre-assigned **master boot sector** (**MBR**) of the selected boot device
  - however, UEFI is sometimes configured to behave like BIOS for compatibility reasons.
- Instead of loading an operating system directly, some computers may be configured to load an intermediary software, so-called **OS loader**, that may allow users to choose from several operating systems present on the boot device
  - for example, on **dual-boot computers** you can choose between two operating systems; say between Windows and Ubuntu, or between macOS and Windows.
- LILO, GRUB, Bootcamp, are examples of such OS loaders
  - options may be set by the administrator through a configuration file that the loader reads before asking the user for selection
  - the configuration file has specifics such as the default OS that is loaded after some timeout, order of the OSes, location of specific loaders on the disk, etc.
  - For more details on Linux default OS Loader GRUB see: https://help.ubuntu.com/community/Grub2

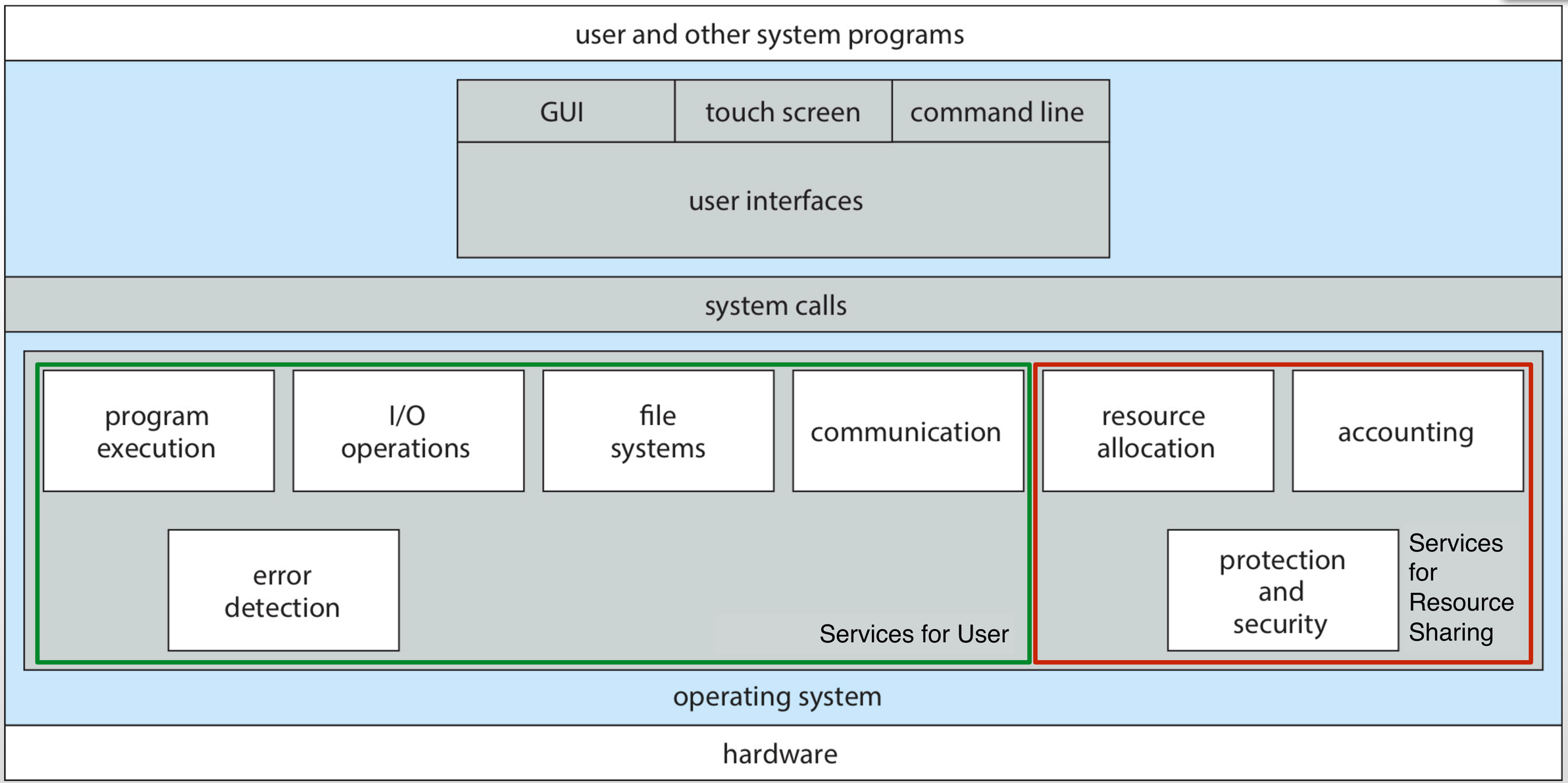
OS Memory Layout



(a) At system startup (b) running a program

Modern OSes have flexible organization of memory

OS Services


user and other system programs
GUI
touch screen
command line
user interfaces
system calls
program execution
I/O operations
file systems
communication
resource allocation
accounting
error detection
protection and security
Services for User
Services for Resource Sharing
operating system
hardware

OS Services for the User

- **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. Obviously, programs need to read and write files and directories, create and delete them, search them, list file information, permission management.
- **Communication** – 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

OS Services for Resource Sharing

- **Resource allocation**
  - When multiple users or multiple jobs run concurrently, resources must be allocated to each of them
    - Some resources such as CPU cycles, main memory, and file storage need customized allocation algorithms (e.g., dynamically depending on the state)
    - Others such as I/O devices may use generalized request and release algorithms
- **Accounting/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** is a **mechanism**
      - i.e., involves ensuring that all access to system resources is controlled; for example, from erroneous code like runaway (“floating”) pointers
    - **Security** is a **policy**
      - i.e., access requires user authentication to defending resources from invalid access attempts; for example, from unauthorized access

OS User Interfaces

- All operating systems come with a **user interface** (**UI**)
  - Varies between Command-Line (**CLI**), Graphics User Interface (**GUI**), Batch
  - A UI is not necessarily considered part of the operating system
- **CLI (command-line interface)** — character-based command entry
  - Can be implemented in kernel or by systems program
  - Often multiple flavors implemented – **shells**
    - We use **bash** shell, but there are many others; e.g., sh, csh, zsh, ksh, tcsh
  - Primarily fetches a command from user and executes it
    - Some commands are built-in (i.e., compiled with other shell parts); others may be just names of programs to execute
      - The latter - **utilities** - are convenient: adding new features doesn't require shell modification
- **GUI (graphical user interface)** - user-friendly desktop metaphor interface
  - Used to be mouse, keyboard, and monitor, but touch screen interface is increasingly popular
  - Icons represent files, programs, actions, etc
  - Mouse button clicks, gestures, over objects in the interface cause various actions like providing information, options, execution functionality, opening a file, directory, etc.
  - Invented at Xerox PARC (Palo Alto Research Center) — just like Smalltalk, one of Java precursors
- Many systems now include both CLI and GUI interfaces
  - MS Windows is a GUI with CLI "command" shell (also "powershell" starting with Win7)
  - Apple macOS has "Cocoa" GUI interface with UNIX kernel underneath (Darwin) and shells available in Terminal application (bash is the default)
  - Linux has Gnome or KDE (or any of many, many other) GUIs; it also has CLI in Terminal application (with bash as the default)
  - Some OSes start to be exclusively gesture-based (like iOS); others implement multiple paradigms (like Windows 10, for example)

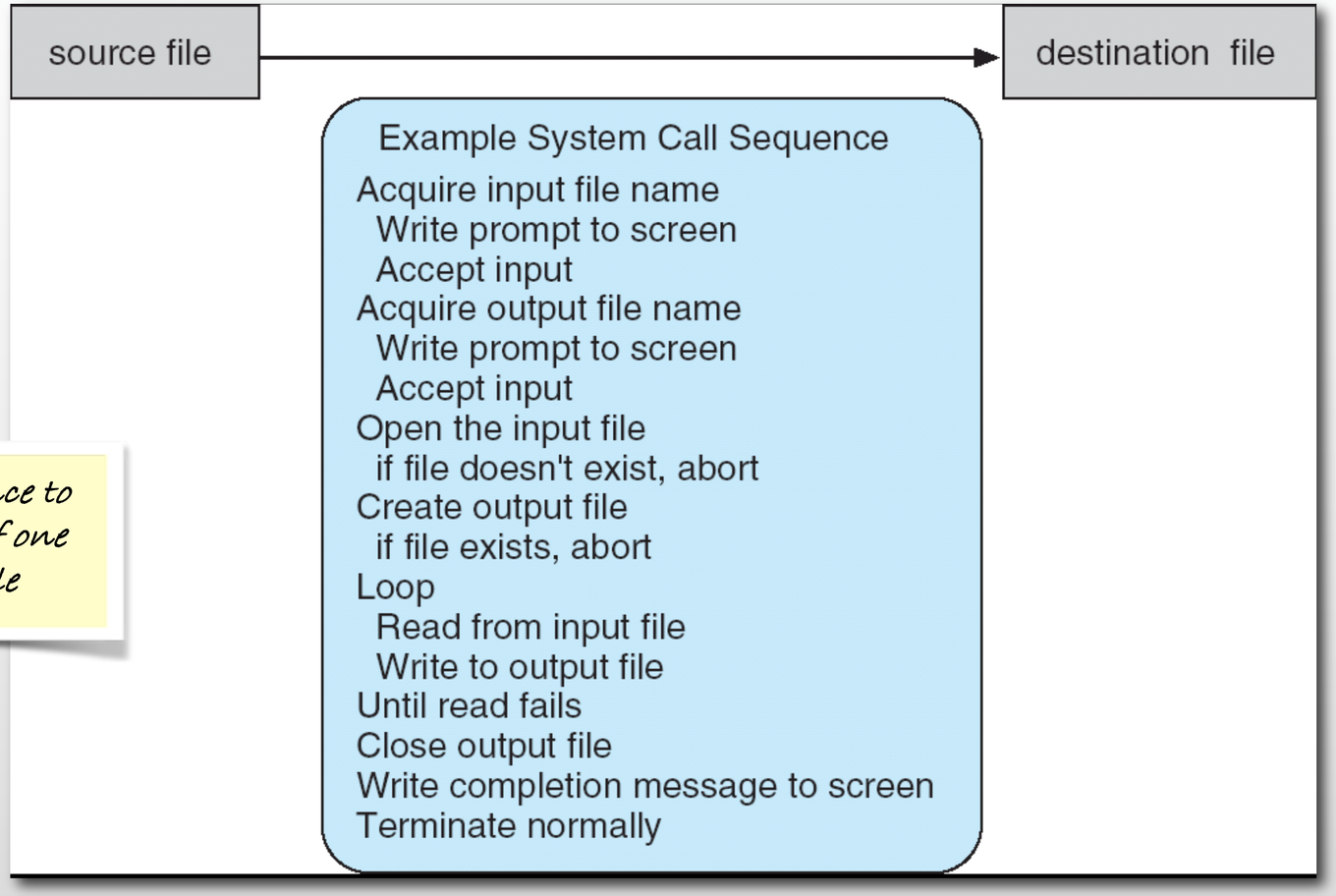
System Calls

- Programming interface to the services provided by the OS
  - Usually very topical in functionality
    - for example, copying a file may involve many system calls:

```
See:
$ man 2 syscall
$ man 2 syscalls
```

```
$ mycopy
< in.txt
> out.txt
$
```

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



System Call APIs

- Usually accessed by programs via a high-level **Application Program Interface** (**API**) rather than direct system call

```
$ man 2 syscall
```

- Often, but not always, the name of the wrapper function is the same as the name of the system call that it invokes.
- A number of system calls may be combined as needed to implement a complex function
- Error handling added through perror()

```
$ man perror
$ sudo apt install moreutils
$ man errno
$ errno -l
```

- Most known OS APIs are:
  - POSIX API for POSIX-based systems including virtually all versions of UNIX, Linux, and macOS

```
$ sudo apt install manpages-posix-dev
```

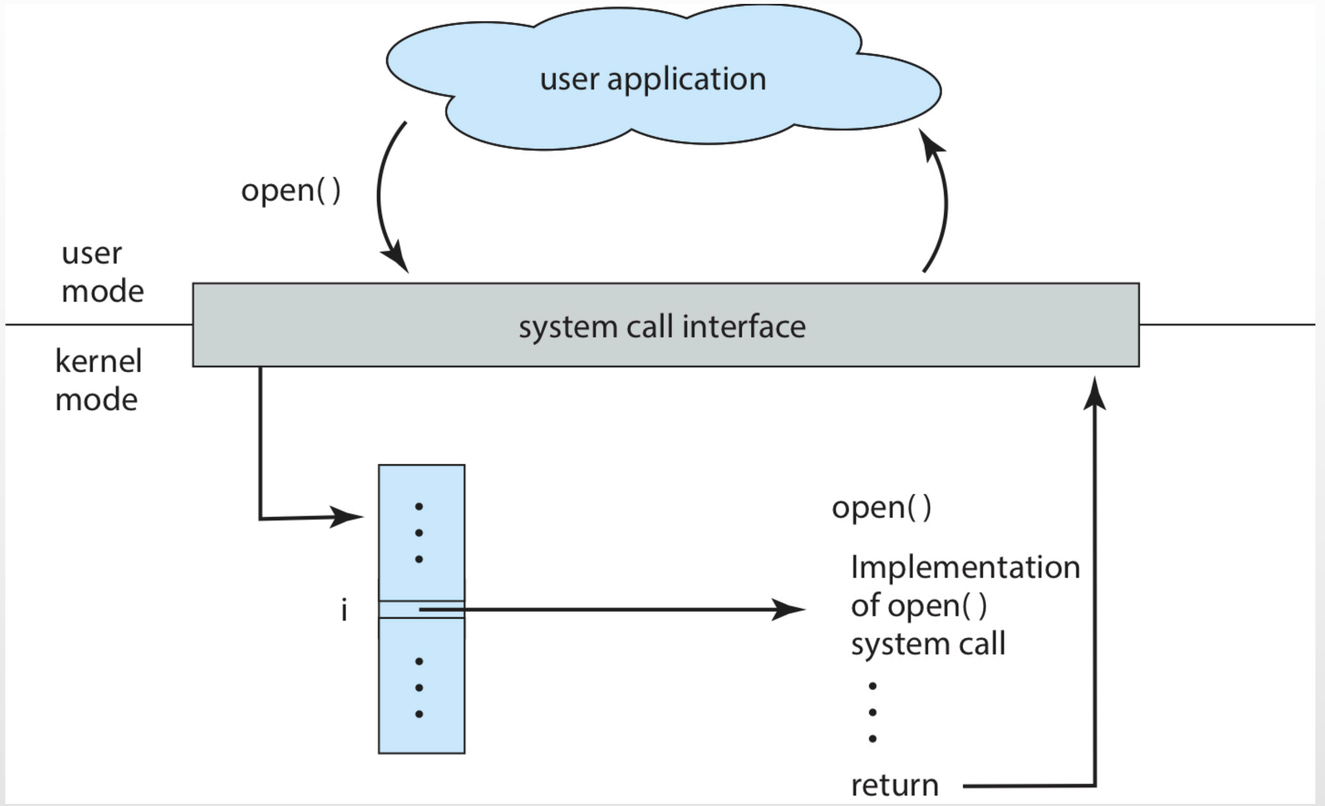
  - Win32 API for Windows
    - "Win32" also applies to 64-bit

- System calls can also be invoked directly through a low-level interface
  - implemented in assembler
  - architecture dependent

```c
#define _GNU_SOURCE
#include <unistd.h>
#include <sys/syscall.h>
#include <sys/types.h>
#include <signal.h>
#include <stdio.h>
#include <pthread.h>

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

  // NOTE: gettid() wrapper IS NOT IMPLEMENTED IN C LIBRARY

  // get thread id using a direct system call

  tid = syscall(SYS_gettid); // same as tid = syscall(224);

  printf("TID from a direct system call: %ld\n", (long int) tid);

  // send a hungup (SIGHUP) signal to the thread
  tid = syscall(SYS_tgkill, getpid(), tid, SIGHUP);
}
```

System Call Implementation

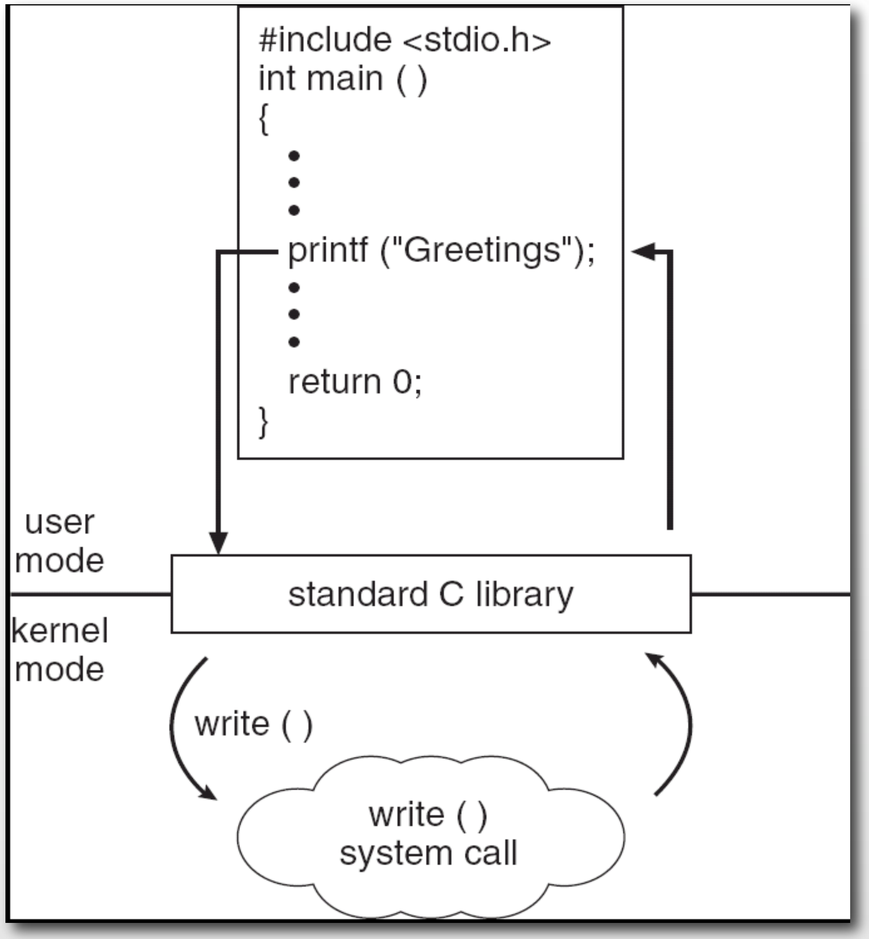
- Typically, there is a number associated with each system call
  - system-call interface maintains a table of references to system call handlers indexed according to these numbers
- The system call interface invokes intended system call in OS kernel and returns status of the system call and any return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



Standard C Library Example

- Most details of OS interface hidden from programmer by the API
  - managed by run-time support library
    - set of functions built into libraries included with the compiler and embedded into every program

The library that is called `libc` is added automatically to every C program

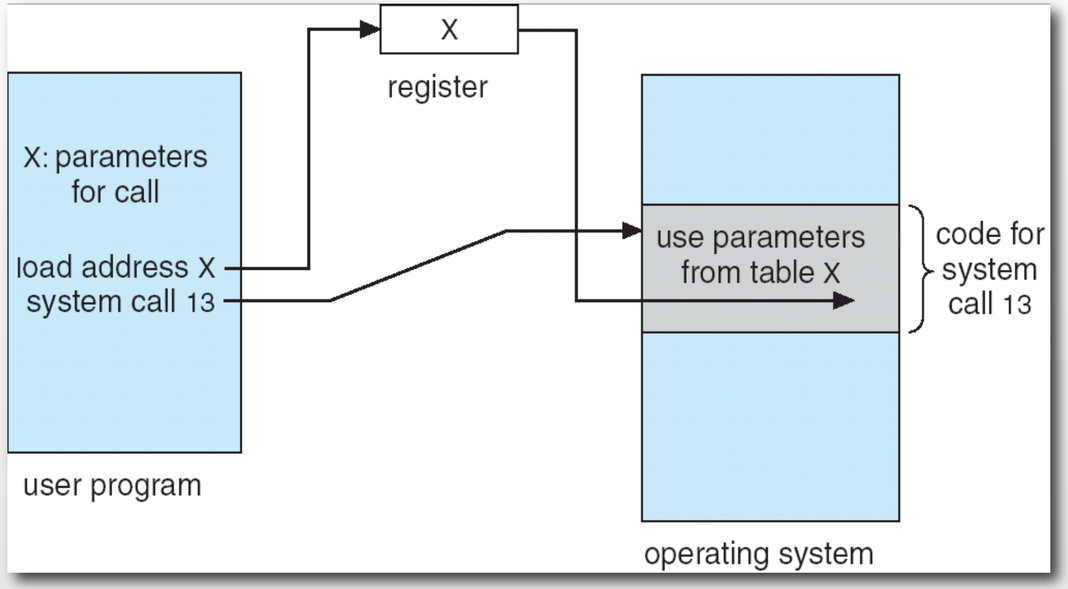


- C program invoking printf() library call, which calls write() system call

System Call Parameter Passing

- As in other type of programming, usually more information is required than simply the identity of desired system call
  - Exact type and amount of information vary according to OS and call
- There are three general methods used to pass parameters to the OS
    - NOTE: This is a choice that the OS designers make, and not the users
      - The developers of the APIs have to stick to the OS convention
  - Simplest: pass the **parameters in registers** (e.g., MS-DOS)
    - In some cases, may be more parameters than registers
  - **Parameters placed, or pushed, onto the stack** by the program and popped off the stack by the operating system (e.g., NetBSD)
    - This is just like a regular function call

- **Parameters stored** in a block, or table, **in memory**, and address of the block passed as a parameter in a register
(e.g., Linux, Solaris)



- Block and stack methods do not limit the number or length of parameters being passed

Types of System Calls

- Process control
- File management
- Device management
- Information maintenance
- Communication

- As we continue with the course, we will be discovering some of the APIs to these system calls at two levels:
  - What they are and how we are using them?
  - How are they (or rather: should be) implemented?

System Programs

- Most users' view of the operating system is defined by UI functions (e.g., GUI clicks, CLI commands) and by **OS utilities**, and not by the actual system calls
  - In fact, most OS utilities are **system programs** rather than shell commands
    - open a man page for bash to see which commands are listed there; anything else called from the shell is a utility

```
$ man bash
```

- System programs provide a convenient environment for program development and execution.
  - They can be divided into utilities for:
    - File manipulation
    - Status information
    - Programming language support
    - Program loading and execution
    - Communication
    - Common application programs (e.g., date, time, text processing, pattern matching, searching, calculators, formatters, etc.)
      - Some applications may become recognized as system programs as they gain popularity and are included with the distribution of the OS
        - e.g., many GNU utility programs

System Programs *(cont.)*

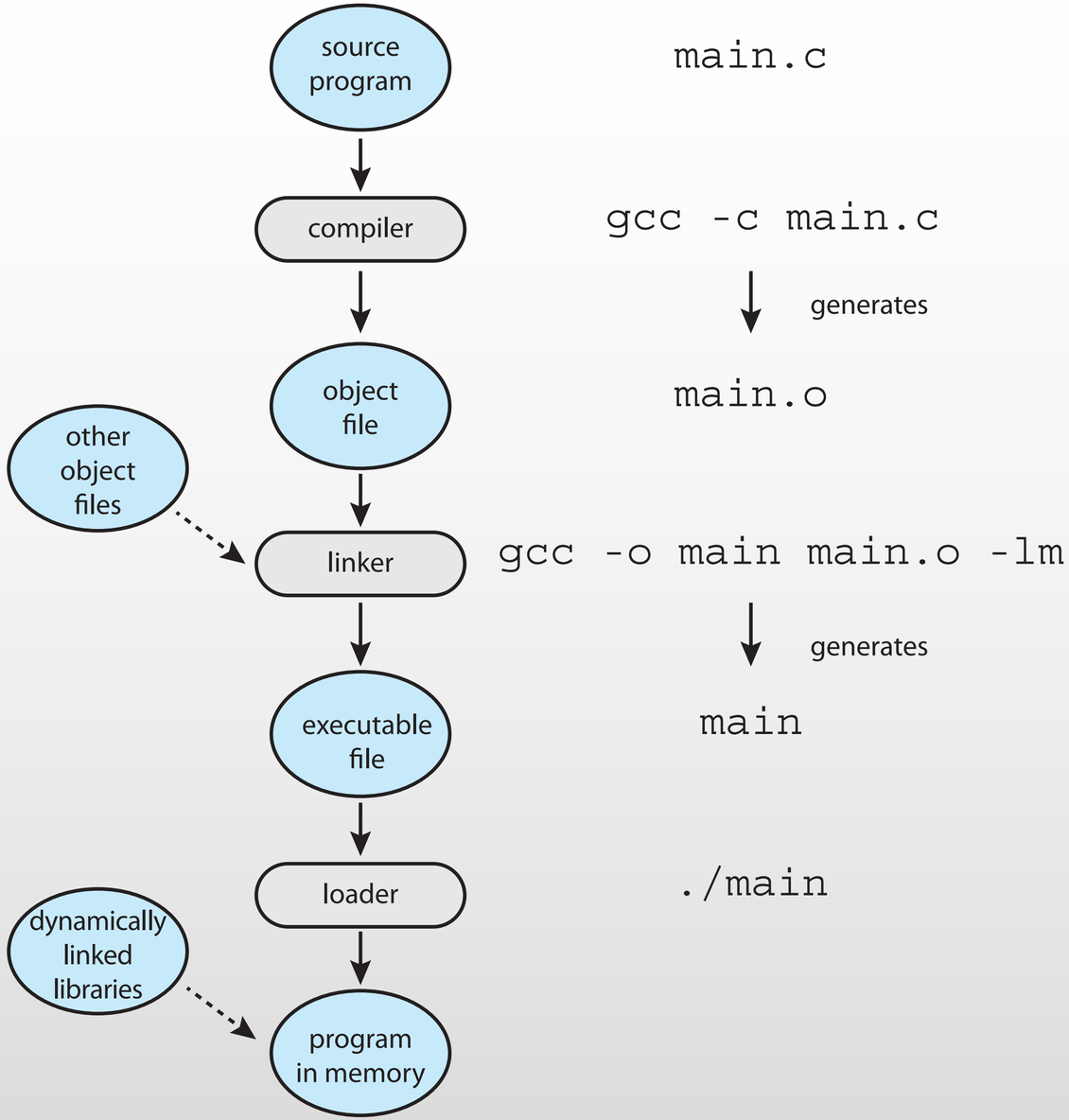
- Some are simply user interfaces to system calls; others are considerably more complex
- **File management**
  - Create, delete, copy, rename, print, dump, list, and generally manipulate files and directories
- **Status information**
  - Some ask the system for info - date, time, amount of available 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
    - the text can be piped into another program rather than sent to the terminal
  - Some systems implement a system database used to store and retrieve configuration information; for example, Registry in Windows
  - Others, provide utilities to manipulate per-program configuration files; for example, plists in macOS

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
- **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
- **Communication**
  - 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

Program Building, Loading, and Execution

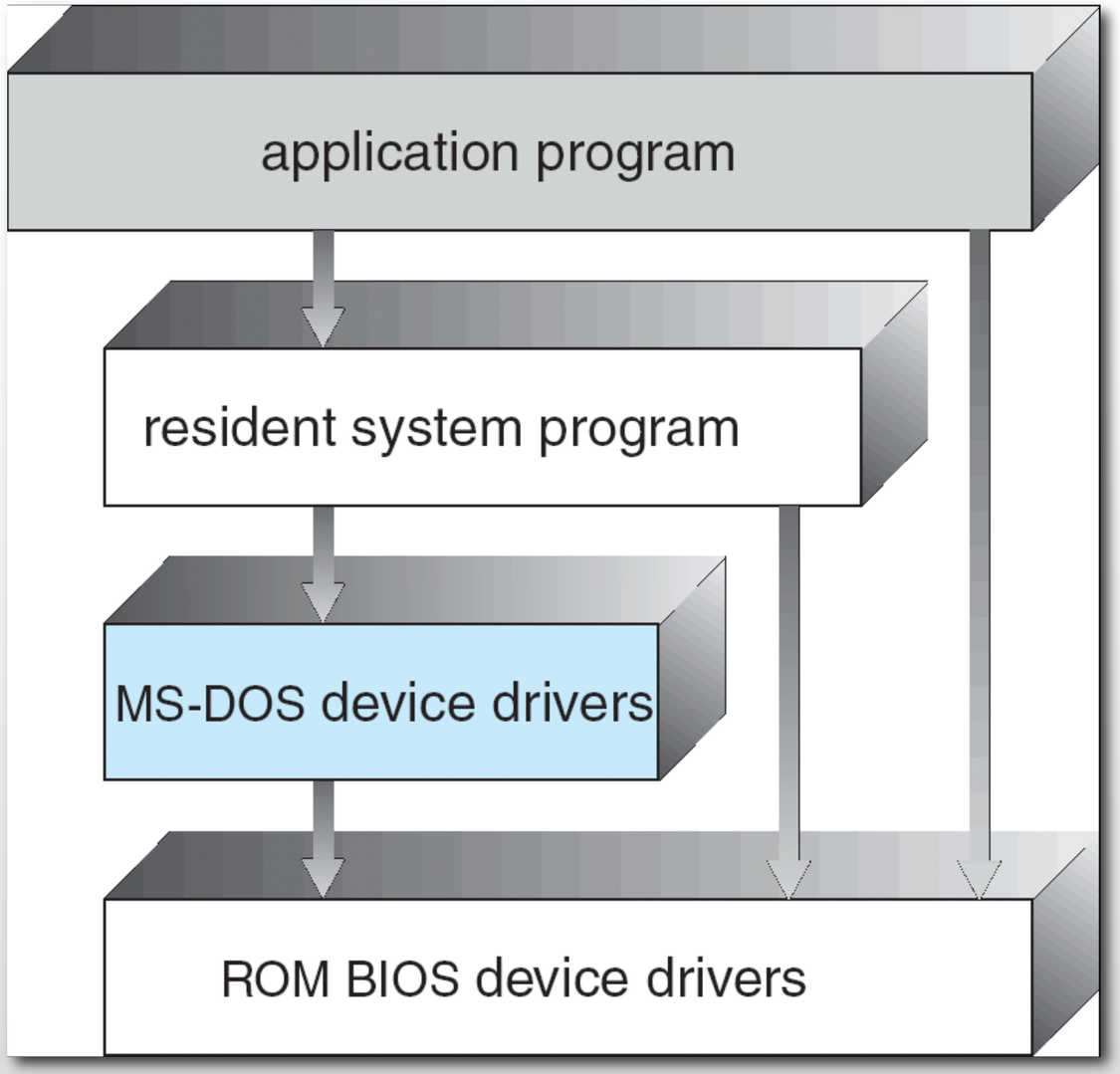
- Recall from COMP232



OS Structures: Simple Structure

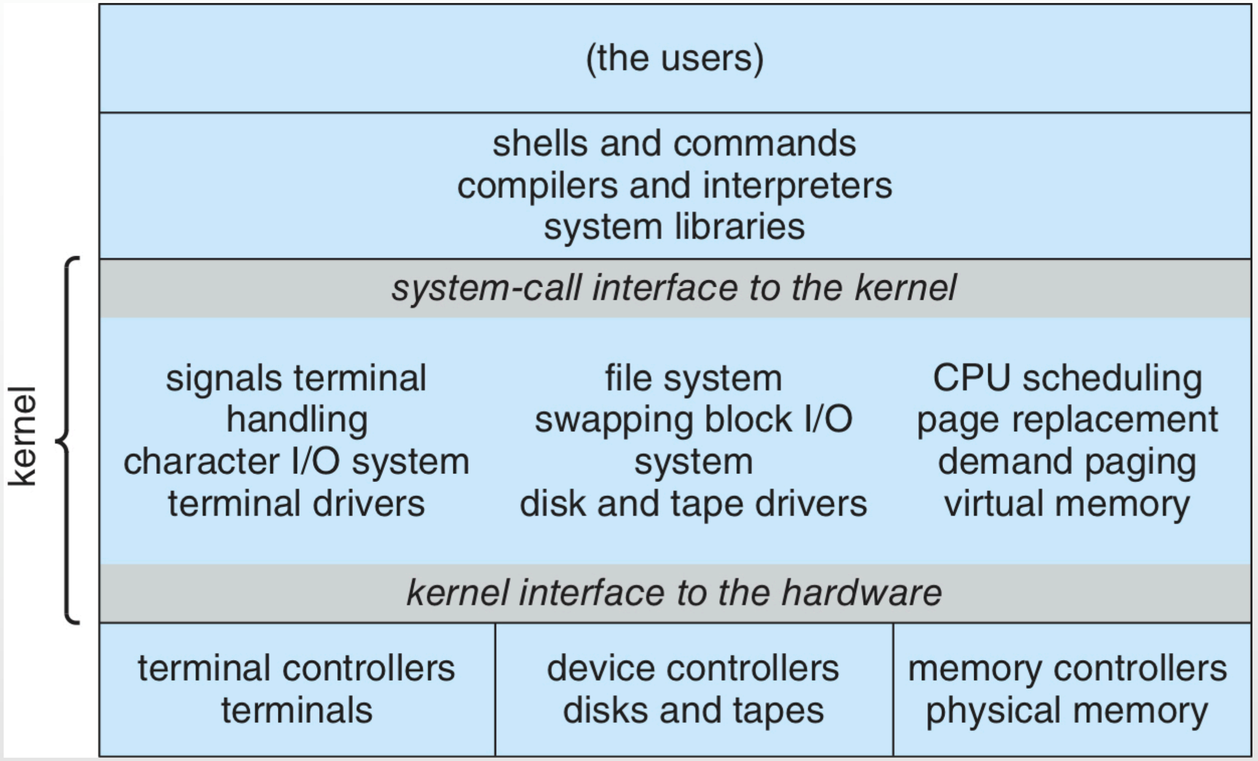
- Early OSes (e.g., MS-DOS) written to provide the most functionality in the least space
  - Monolithic (i.e., not divided into modules)
  - Although they had some structure, the interfaces and levels of functionality were not well separated
    - A lot of problems!

- Famous funny quotes:
  - Mr. Gates (former CEO of Microsoft): *“Nobody ever will need more memory!”*
    - (about the 640 kBytes allocation to programs; that is **kilo**!)
  - A bit better than Mr. Watson’s (at the time, the CEO if IBM): *"I think there is a world market for maybe five computers."*



Structure of Original UNIX

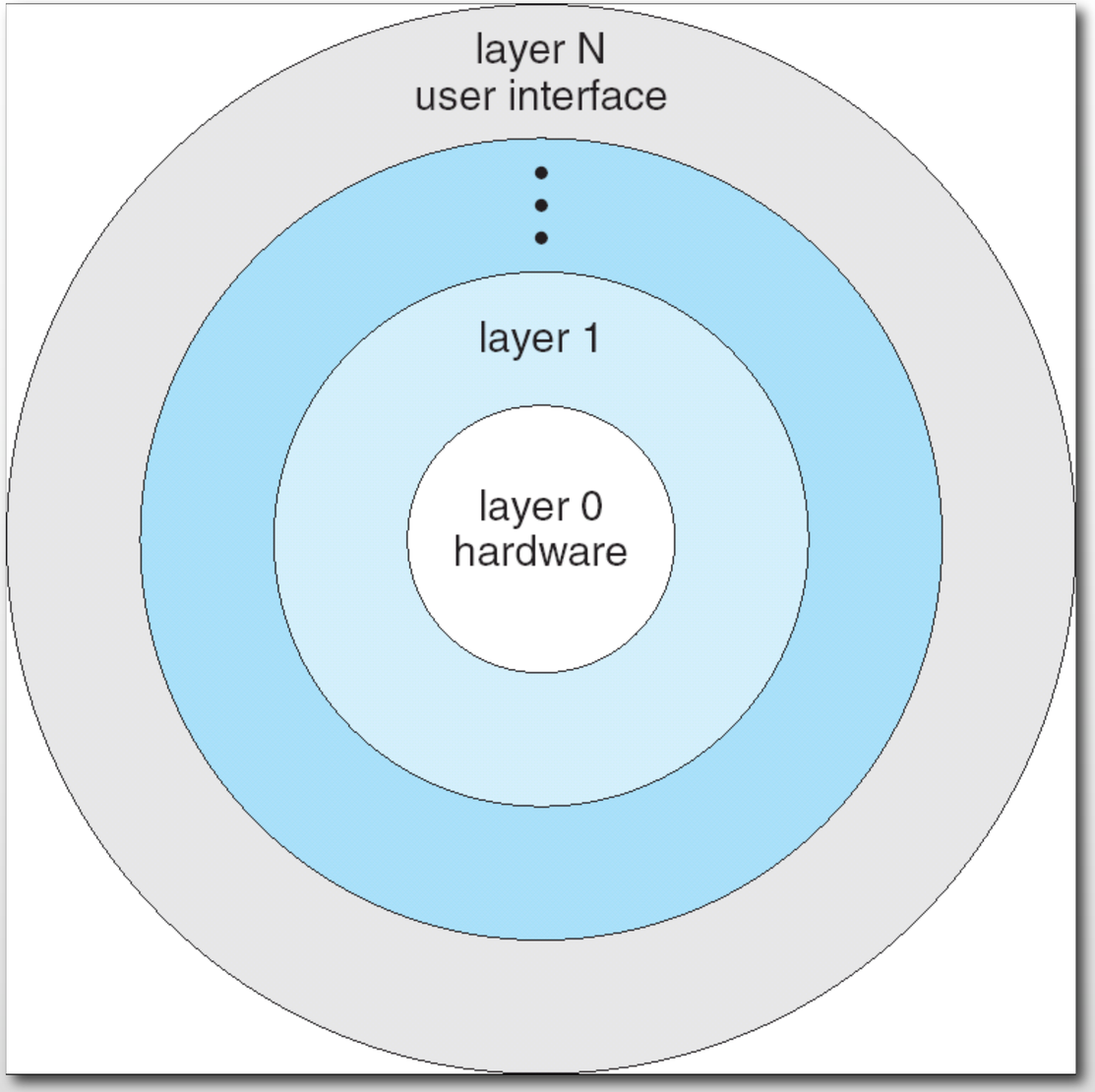
- UNIX – limited by hardware functionality, the original UNIX operating system had limited structuring
  - a lot of this original design is still well visible in modern implementations
- The UNIX OS is layered
  - it consists of two separable parts:
    - systems programs
    - the kernel
      - consists of everything below the system-call interface and above the physical hardware
- Problem
  - kernel provides the file system, CPU scheduling, memory management, and other operating-system functions
  - many functions for one level



OS Structures: Layered Approach

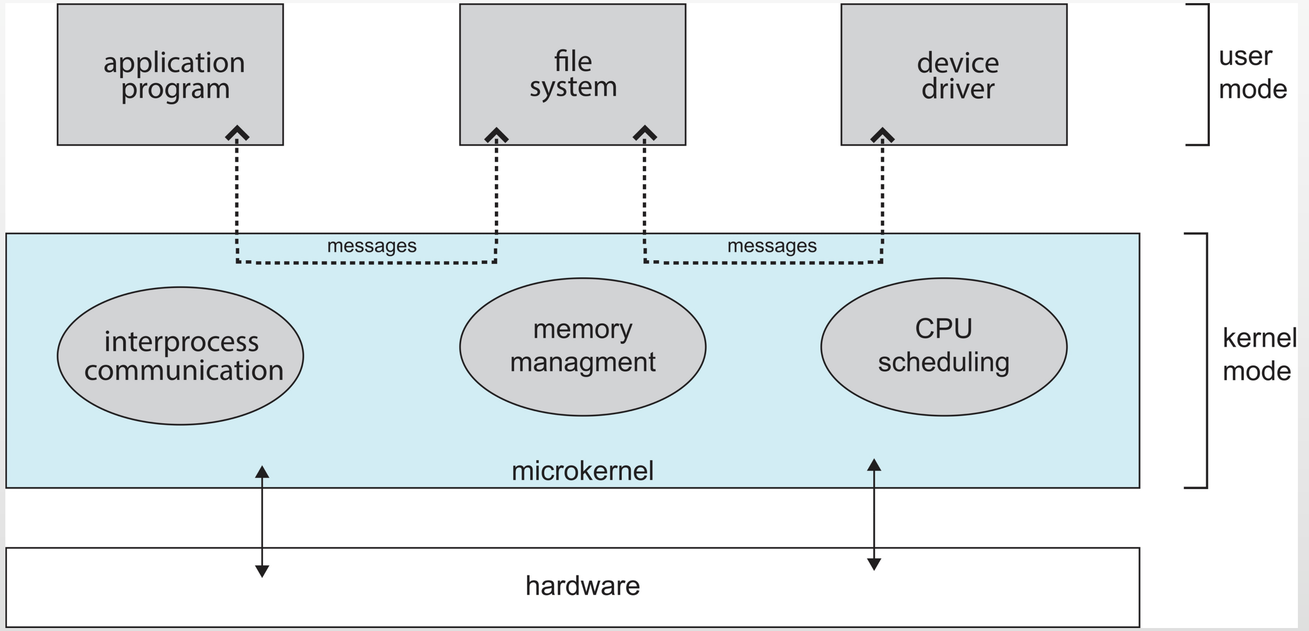
- Inspired by the extremely successful multi-layer network communication stack
- The operating system is divided into several layers (levels), each built on top of lower layers. The bottom layer (layer 0), is the hardware; the highest (layer N) is the user interface.
- With modularity, layers are selected such that each uses functions (operations) and services of only the lower-level layers

- Nice idea, but serious pragmatic problems
  - tricky to decide what goes into which layer
  - efficiency



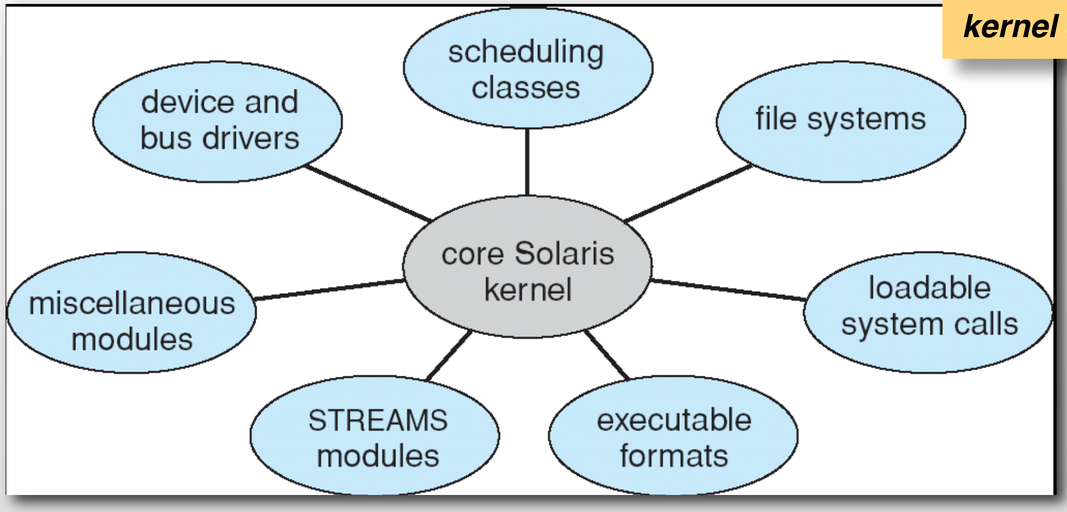
Microkernel System Structure

- Moves as much from the kernel into "user" space as possible
  - What's left: **microkernel**
- Communication takes place between user modules using message passing
- Benefits:
  - Easier to extend a microkernel
  - Easier to port the operating system to new architectures
  - More reliable (less code is running in kernel mode)
  - More secure
- Detriments:
  - Performance overhead of user space to kernel space communication
- Often, pragmatic compromises are used to improve efficiency



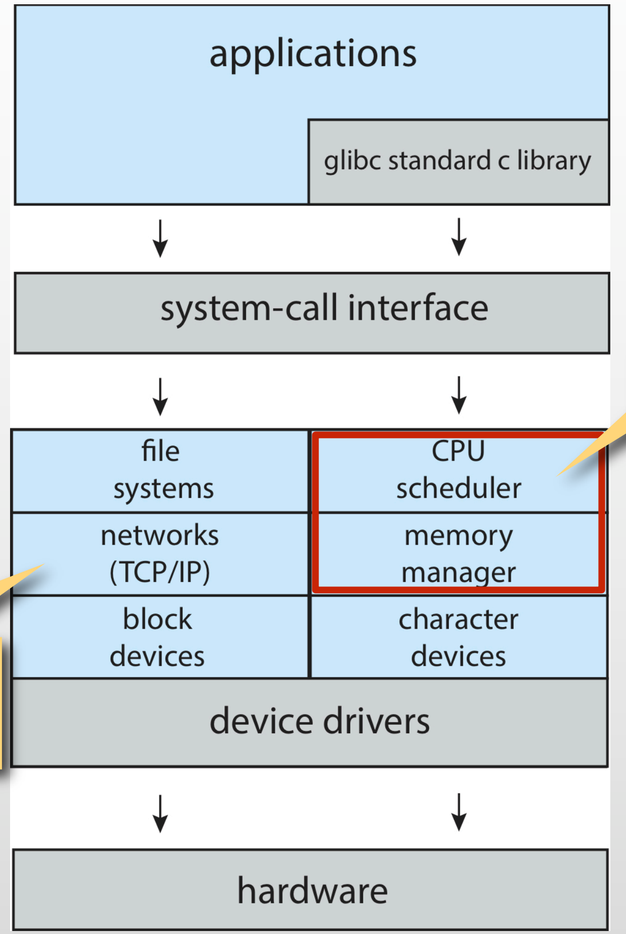
Modular OS Architecture

- Most modern operating systems implement kernel modules
  - Object-oriented approach
  - Each core component is separate
  - Each talks to the others over known interfaces
  - Each is loadable as needed within the kernel
- Overall, like layers but more flexible



also *modules* in Linux
***kernel extensions* (*kext*) in macOS**

Linux
UNIX-like monolith with modules

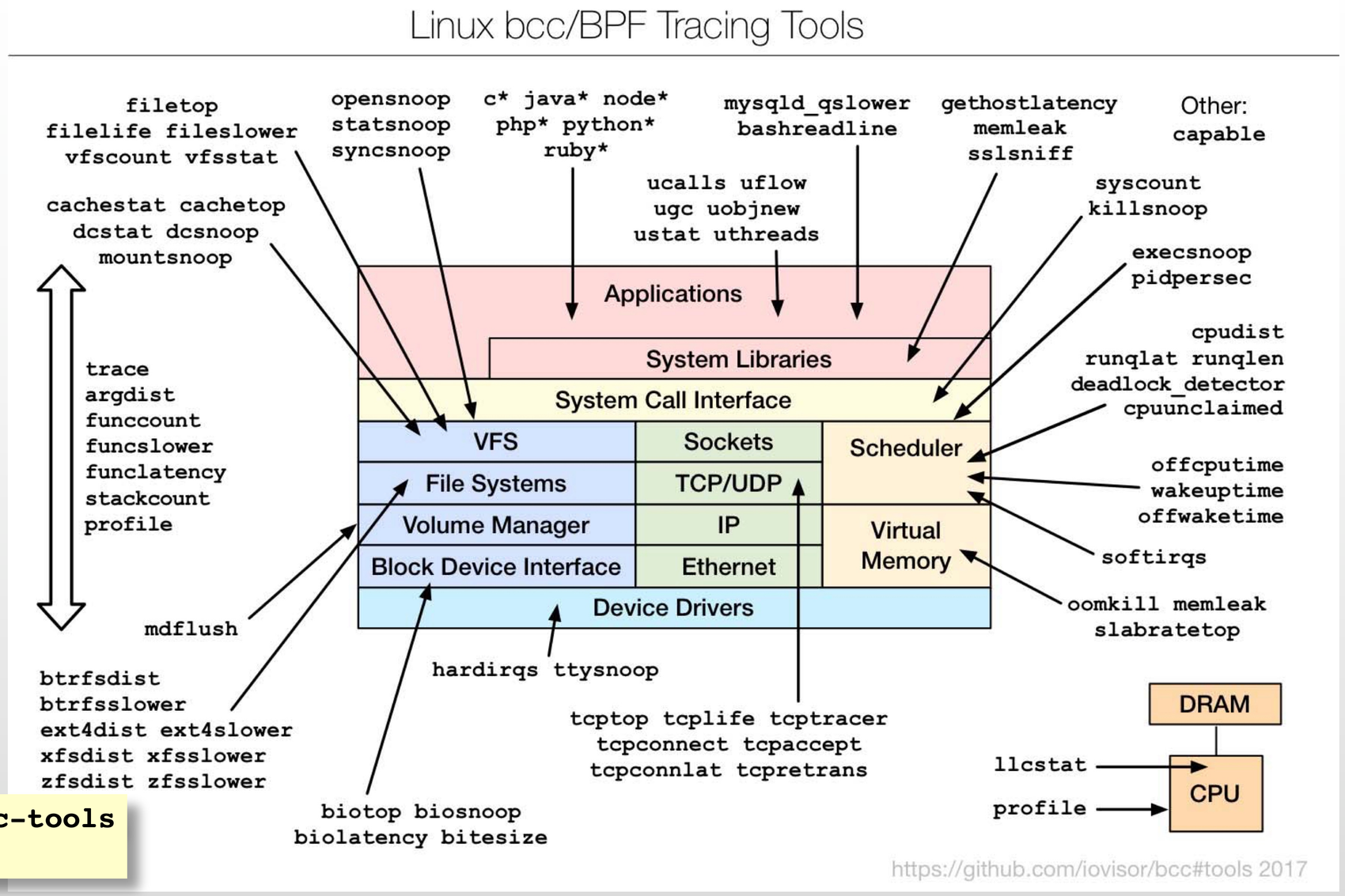


core Linux kernel

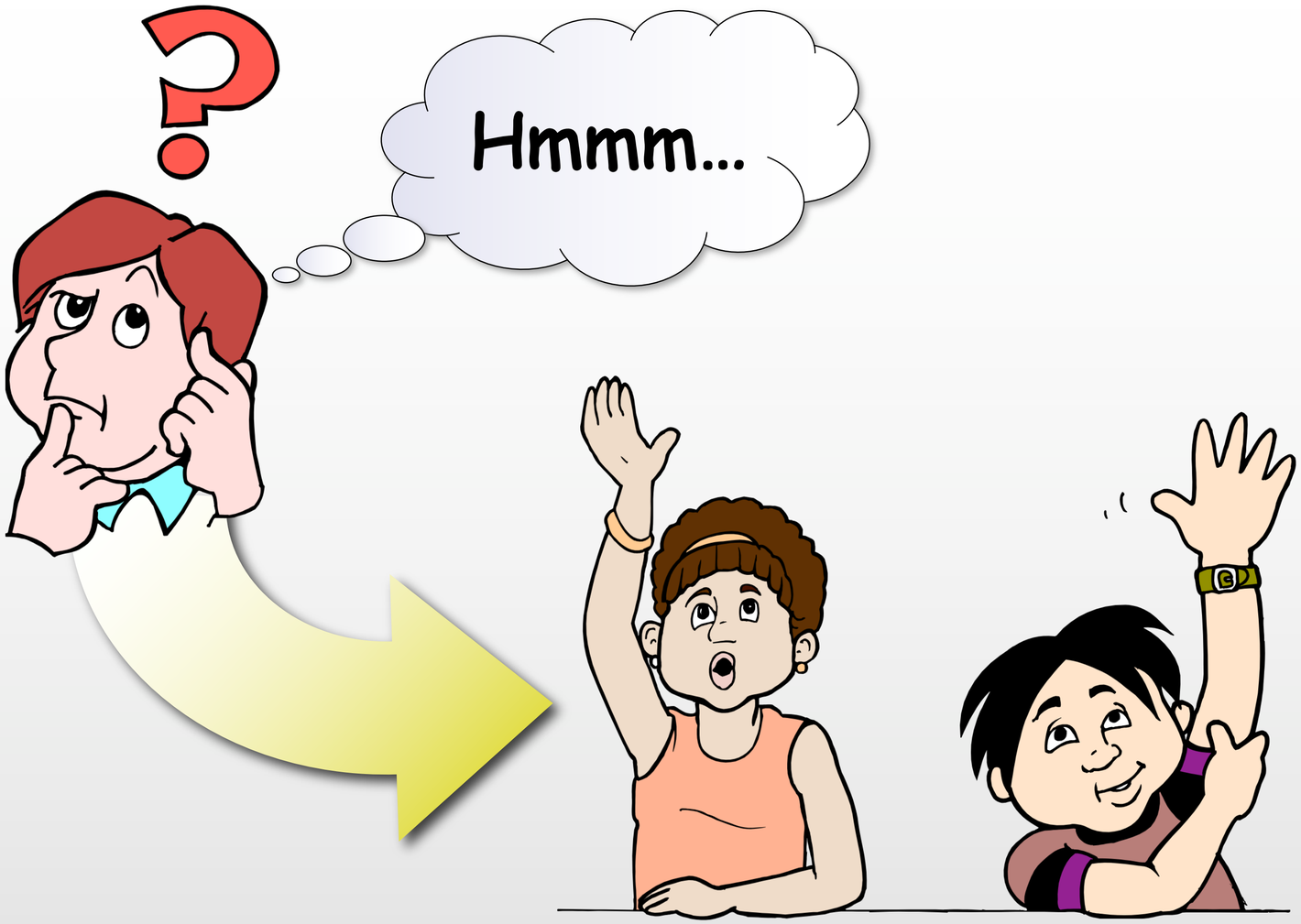
additional kernel modules *compiled* or *loadable*

OS Debugging

```
$ man gdb

$ man ps
$ man top

$ man vmstat

$ sudo apt install net-tools
$ man netstat

$ sudo apt install sysstat
$ man iostat

$ man strace

$ sudo apt install linux-tools-common
$ sudo apt install linux-tools-generic
$ man perf

$ man tcpdump
```




```
$ sudo apt install bpfcc-tools
$ sudo syscount-bpfcc
```



COMP362 Operating Systems

Prof. AJ Bieszczad