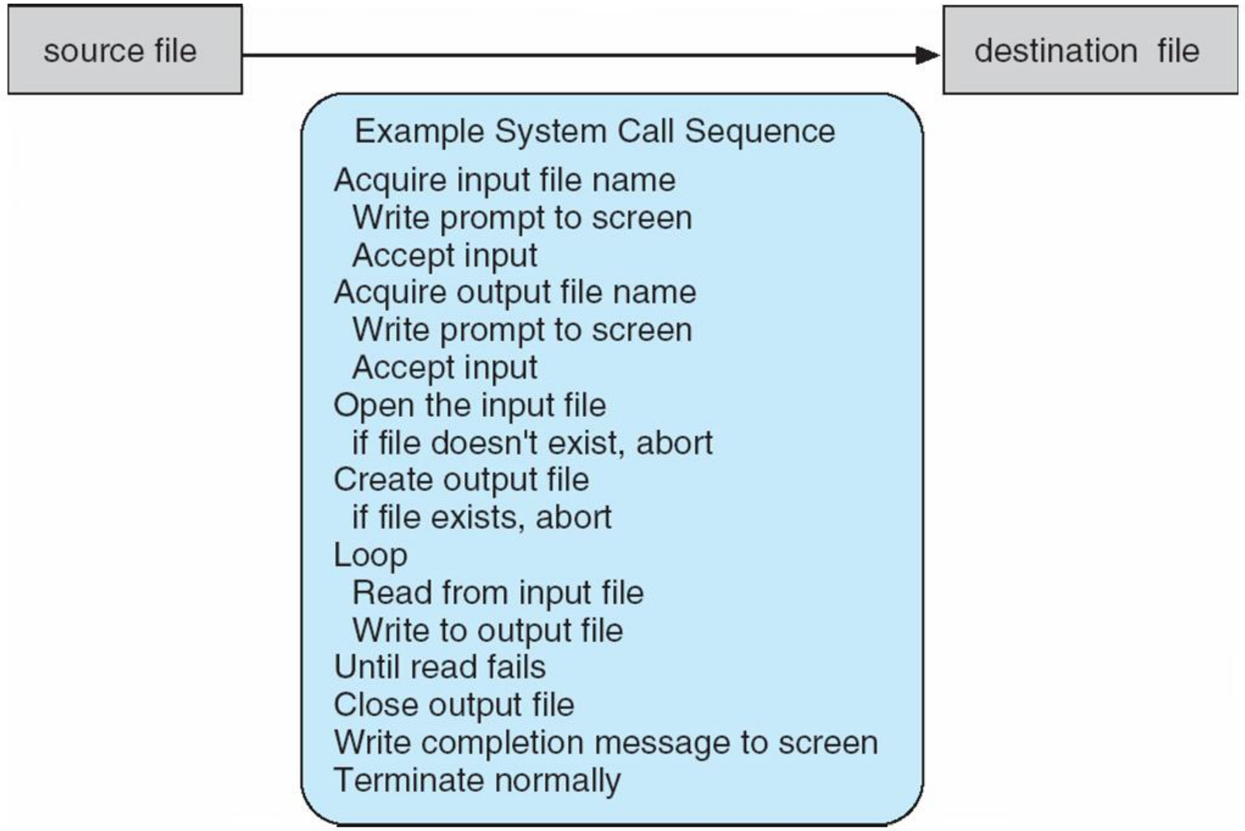
Example of System Calls

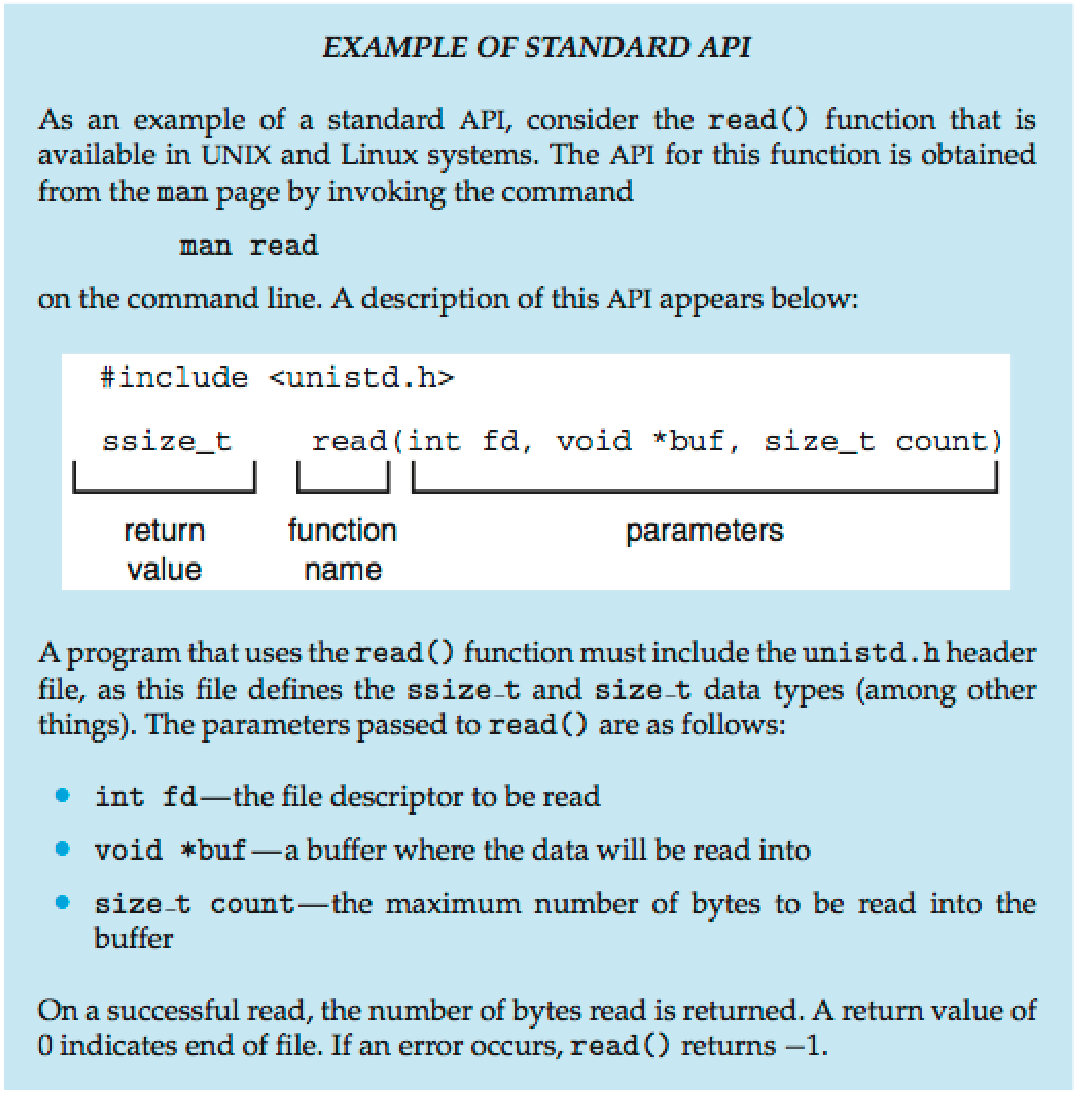
- The example below demonstrates the steps needed to copy the contents of one file to another file → A sequence of system calls results



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
```

Example of POSIX 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>
```

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

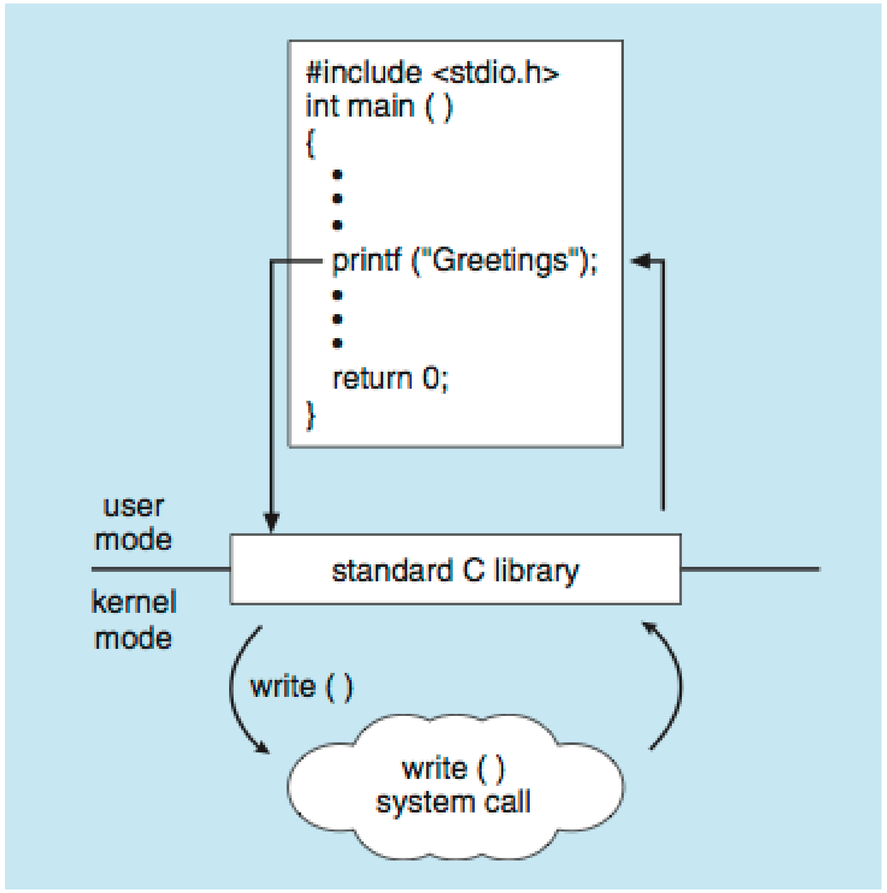
A program that uses the `read()` function must include the `unistd.h` header file, as this file defines the `ssize_t` and `size_t` data types (among other things). The parameters passed to `read()` are as follows:

- `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.

Example using standard C library

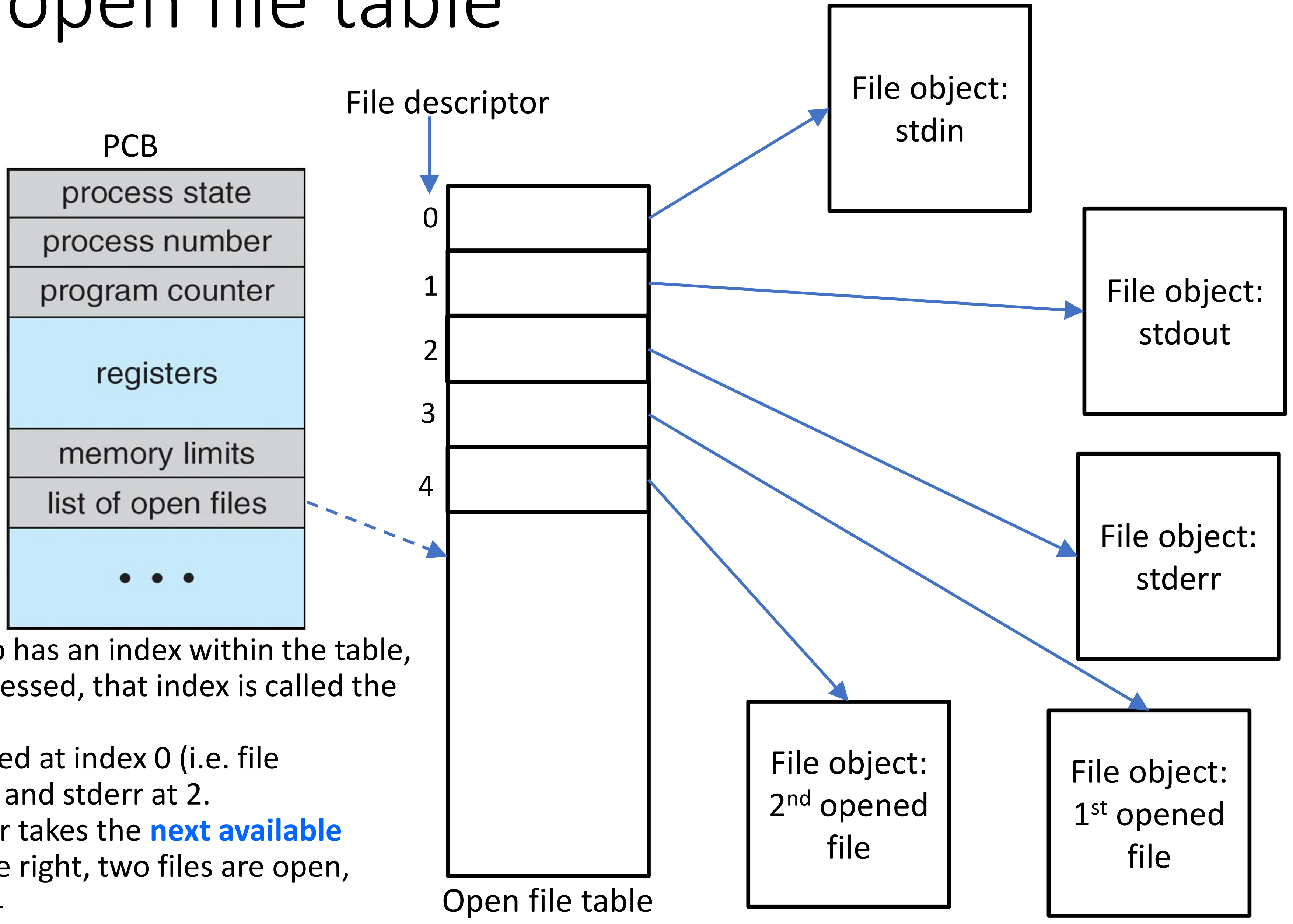
- C program invoking printf() library call, which calls write() system call



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

Per-process open file table

- In Linux, every process is represented by an object, called the process control block (PCB)
- The PCB contains a **pointer that eventually points to the open file table**, which contains info about open files.
- Each entry within the open file table has some info about the open file + a pointer that eventually points to an object representing that file (**File object**)



- Each entry in that table also has an index within the table, through which it can be accessed, that index is called the **file descriptor**.
- The pointer to stdin is located at index 0 (i.e. file descriptor = 0), stdout at 1, and stderr at 2.
- Any file you open thereafter takes the **next available index** (in the example to the right, two files are open, with file descriptors 3 and 4

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)

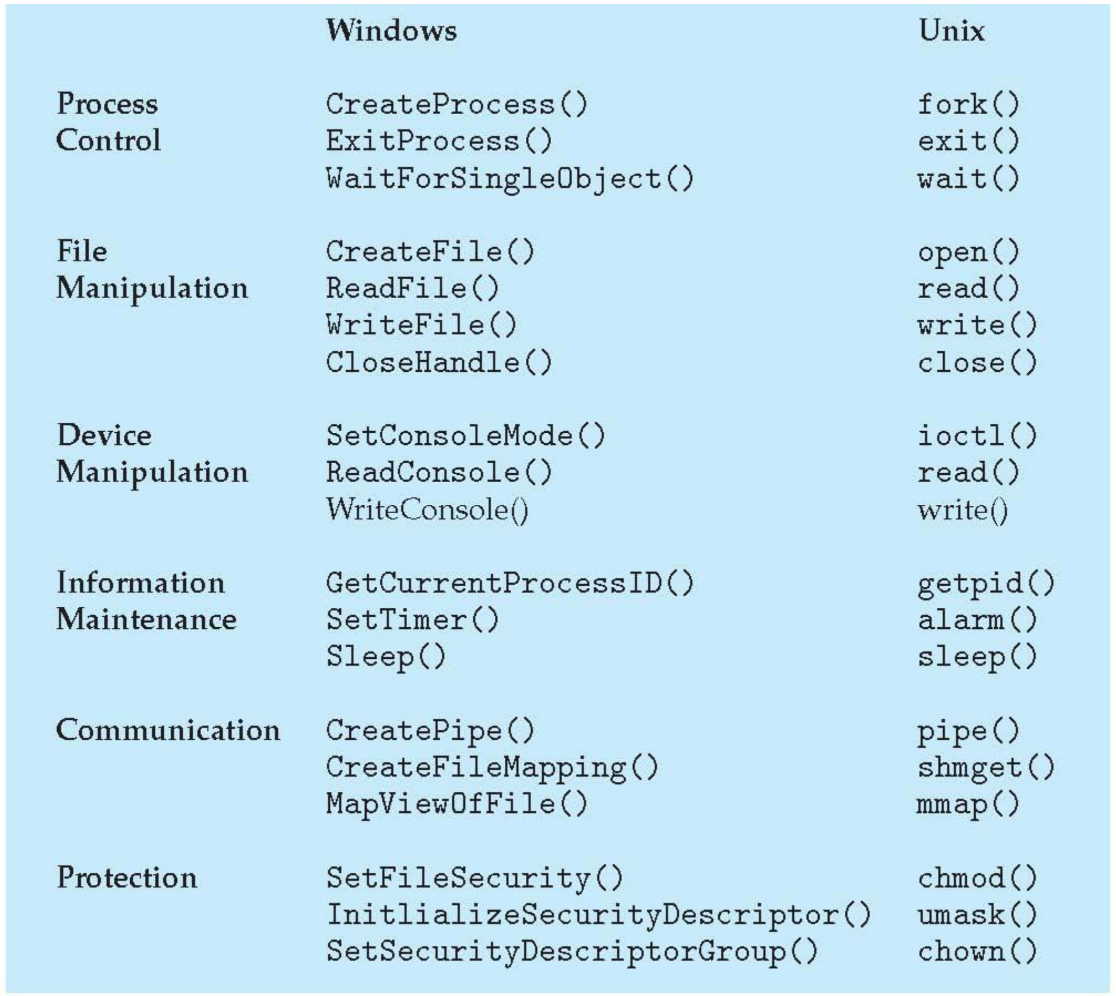
Types of System Calls (Cont.)

- 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.

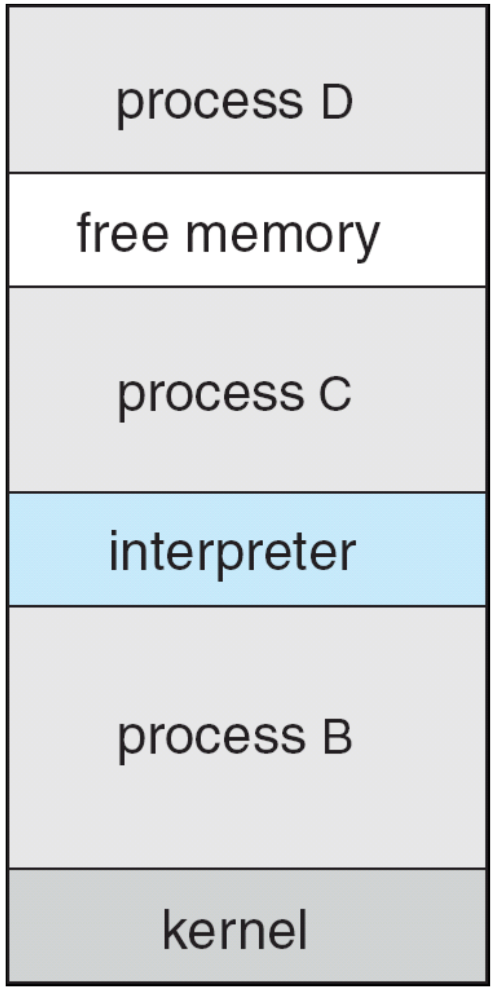
Examples of Windows and Unix System Calls



	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()

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

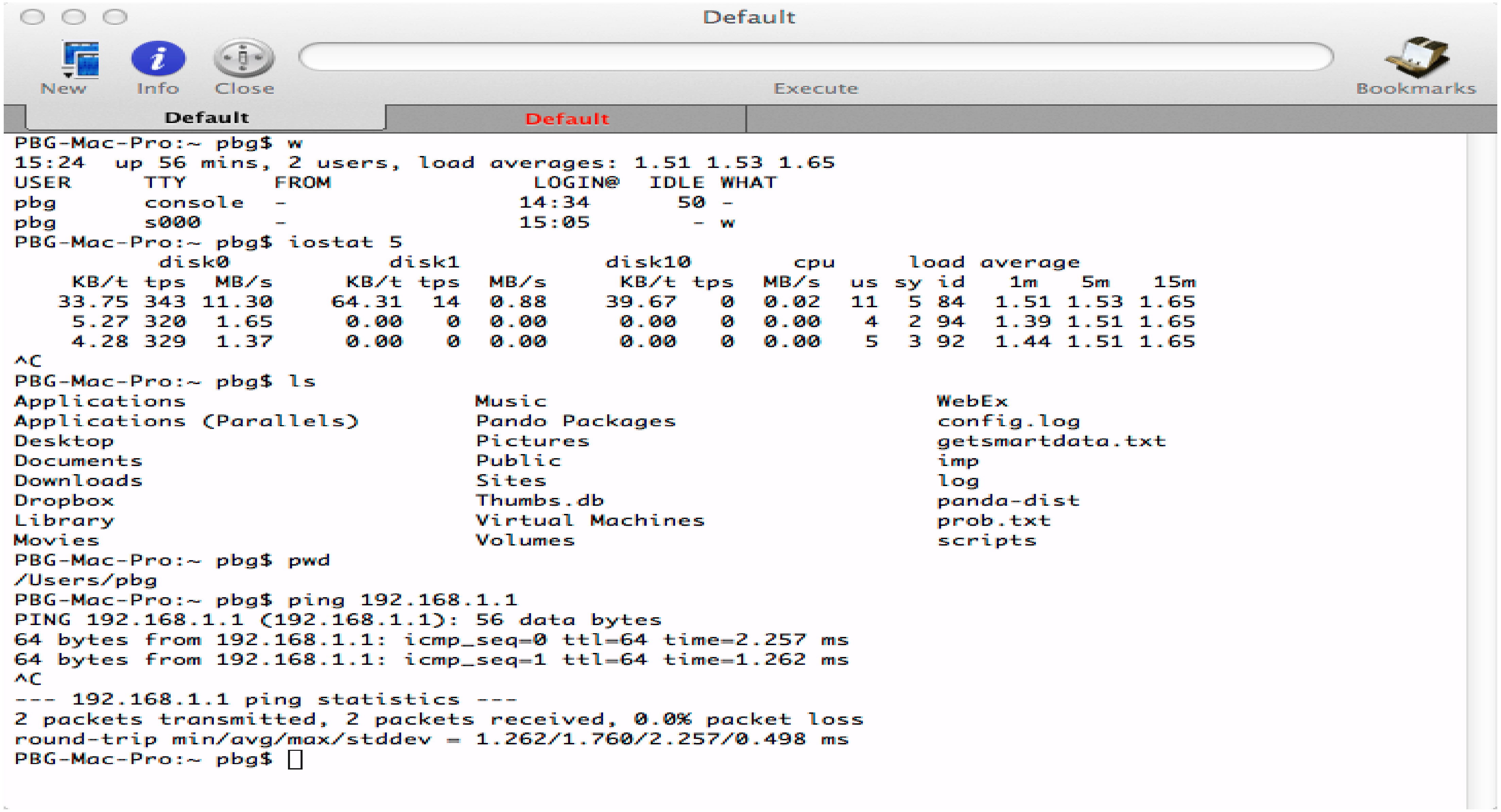


2.2 Operating System’s User Interface - CLI

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)

2.5 System Programs and utilites

- Most users' view of the operating system is defined by system programs and system utilities, 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 (e.g. **vi** in Unix-like systems)
  - Special commands to search contents of files or perform transformations of the text (e.g. **grep**)
- **Programming-language support**
  - Compilers, assemblers, and interpreters sometimes provided.
  - Also a debugging system (for higher level as well as machine language).
  - An API library, e.g. **libc**
- **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.
    - For windows, known as; **services** (e.g. TCP/IP services, print service), **subsystems** (e.g. WSL)
    - For Unix-like systems, known as; **daemons** (e.g. TCP/IP daemon)
  - 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.

2.6 Operating System Design

- There is no unique process for designing an OS, but some approaches have proven successful. Design starts by defining goals and specifications

- At highest level, design is affected by choice of **hardware** and **type of system**:
    - batch, single user, or multi user
    - Distributed?
    - real time?

- Next level are requirements for **user** goals and **system** goals
    - User requirements – operating system should be
        - Easy to use
        - Responsive
    - System requirements – operating system should be
        - Easy to design, implement, and <u>maintain</u>
        - Flexible (i.e. easy to modify and supports different choices/configurations)
        - Efficient
        - Reliable and error-free

Operating System Design

- An important design choice may be to separate

  Mechanism: ***How*** to do it?

  Policy: ***What*** specifically will be done?

  - e.g.: The OS timer construct for preempting processes is a mechanism, but the exact tick time is a policy.
- The separation of policy from mechanism is a very important principle, it allows maximum **flexibility** if policy decisions are to be changed later
  - e.g.: A scheduling mechanism may be made general-purpose to allow various policy setups such as time-sharing, batch, real-time etc.
- In some systems, both mechanisms and policy are encoded to enforce a **global look** such as in Windows and Mac OS X, but not in Unix (e.g. different window managers such as KDE, GNOME, etc.)

Operating System Implementation

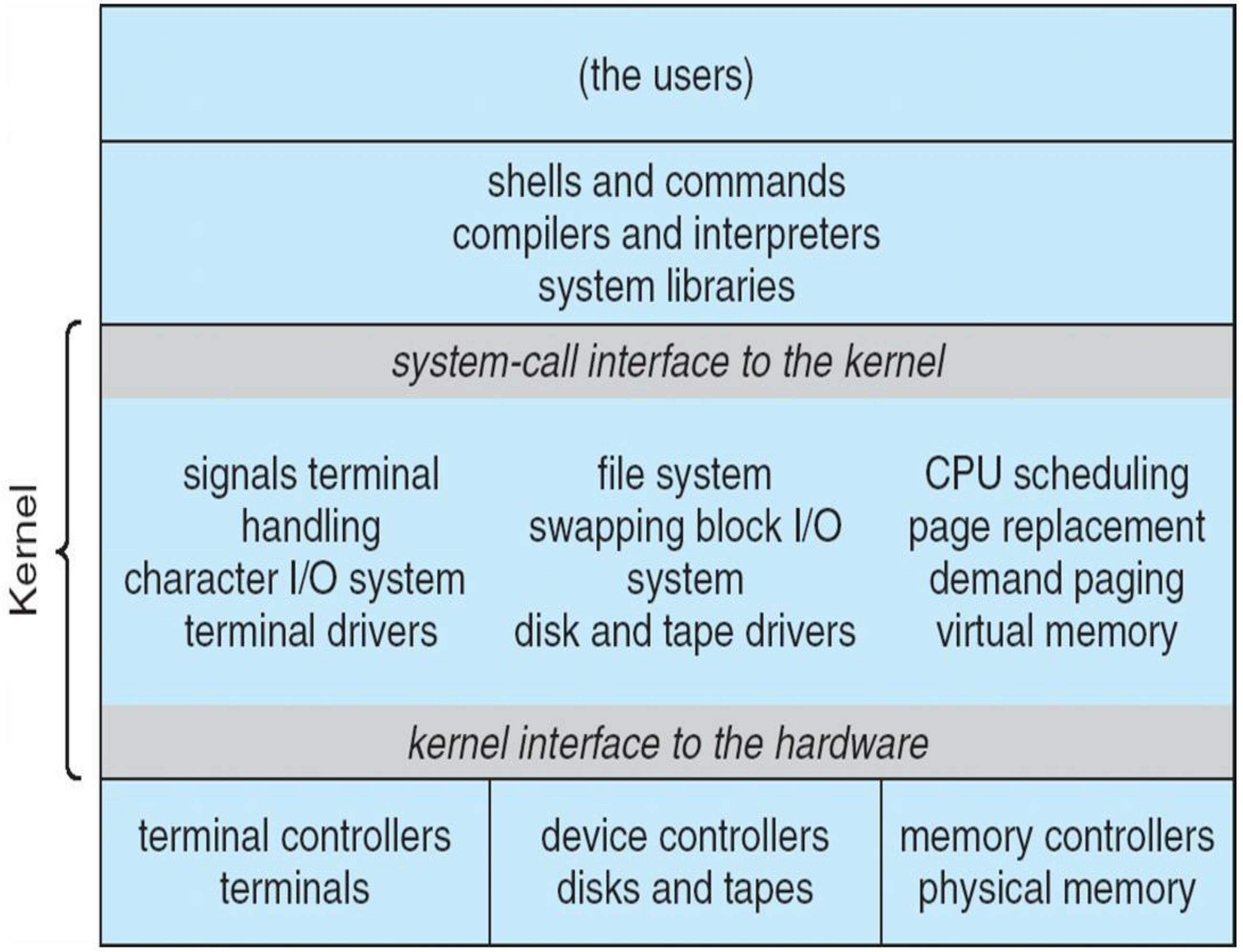
- Early OSes in assembly language (efficient but difficult to code/maintain)
- C, C++ (less memory-efficient, less performance but easier to code/maintain)
- Now a mix of languages
  - **Lowest levels** in assembly,
    - For hardware/architecture-dependent code and
    - For time critical areas, e.g. **within** interrupt handlers, device and memory managers, and schedulers
  - Main body of **kernel** is written in C. Improving the data structures and algorithms has more impact than coding in a lower-level language.
  - **Systems programs** in C, C++, scripting languages like PERL, Python, shell scripts

2.7 Operating System Structure

- A general-purpose OS is very large and complex program
- Various ways to structure an OS:
  - Monolithic structure
    - Simple – e.g. MS-DOS
    - More complex – e.g. original UNIX
  - Layered – provides abstraction levels
  - Microkernel – e.g. Mach and Minix
  - Modular: Subsystems and modules

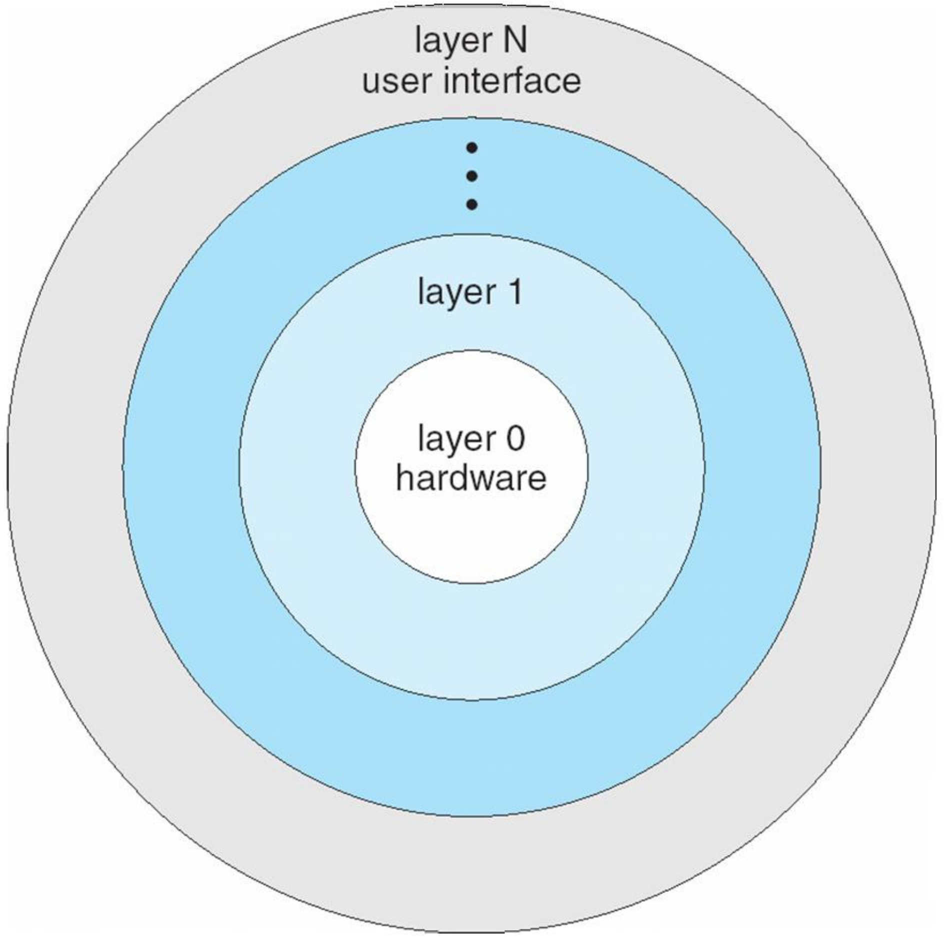
Monolithic Structure -- The original Unix

- UNIX – limited by hardware functionality, the original UNIX operating system had limited structuring. The UNIX OS consists of two separable parts
  - Systems programs and the **shell**
  - 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
    - That's large number of functions for a monolithic one-level system.
  - Beyond simple, but not fully layered.



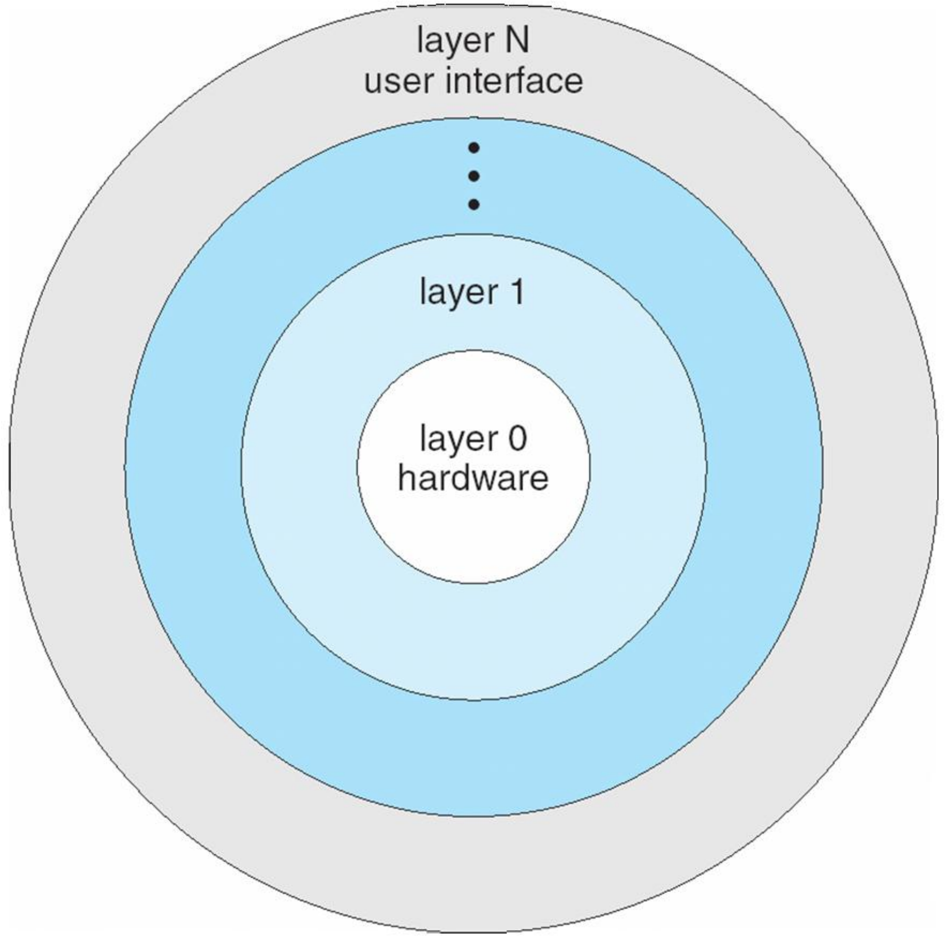
Layered (hierarchial) Approach

- The operating system is divided into a number of 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 the layer underneath it.



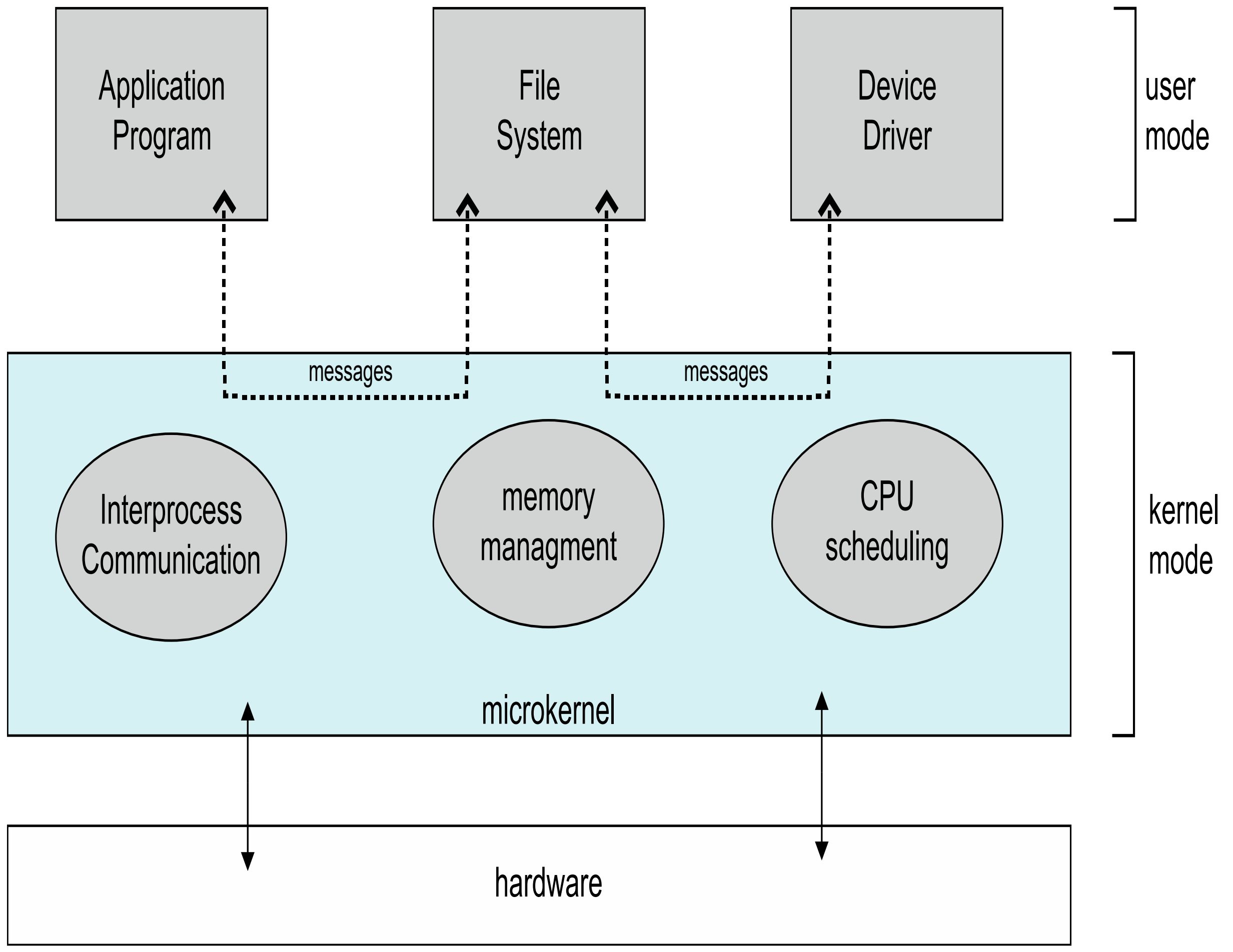
Layered Approach (cont.)

- **Advantages:** Ease of implementation and debugging.
  - The lowest layer may be debugged first (without debugging the higher layers), and once that’s done, the next layer can then be debugged, and so on.
- **Disadvantages:** (caused the layered approach to fall out of favor)
  - Difficulty in realizing layered levels
    - e.g. the memory manager and disk manager may want use each other’s services,
    - same for scheduler and disk manger
  - Less efficient – a call from upper layer propagates through many lower layers.



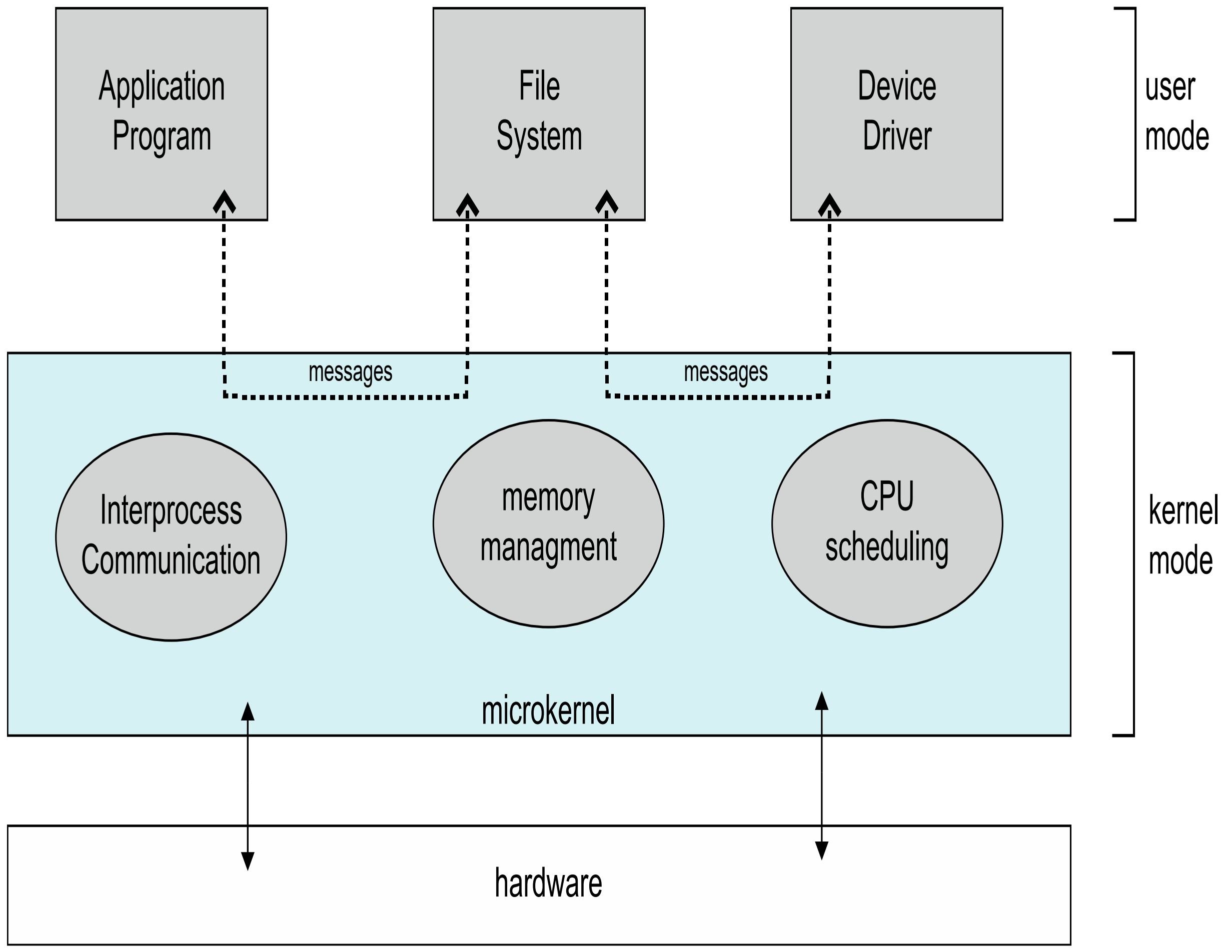
Microkernels

- Moves as much as possible from the kernel into user space
- **Mach** example of **microkernel**
  - Developed in mid 1980's @ Carnegie Mellon university.
  - Maps Unix system calls to messages sent to appropriate user level services
  - Mac OS X kernel (**Darwin**) partly based on Mach
- Micro-kernel provides minimal process and memory management, in addition to a communication facility.
- Communication takes place between user modules using **message passing** via the microkernel.



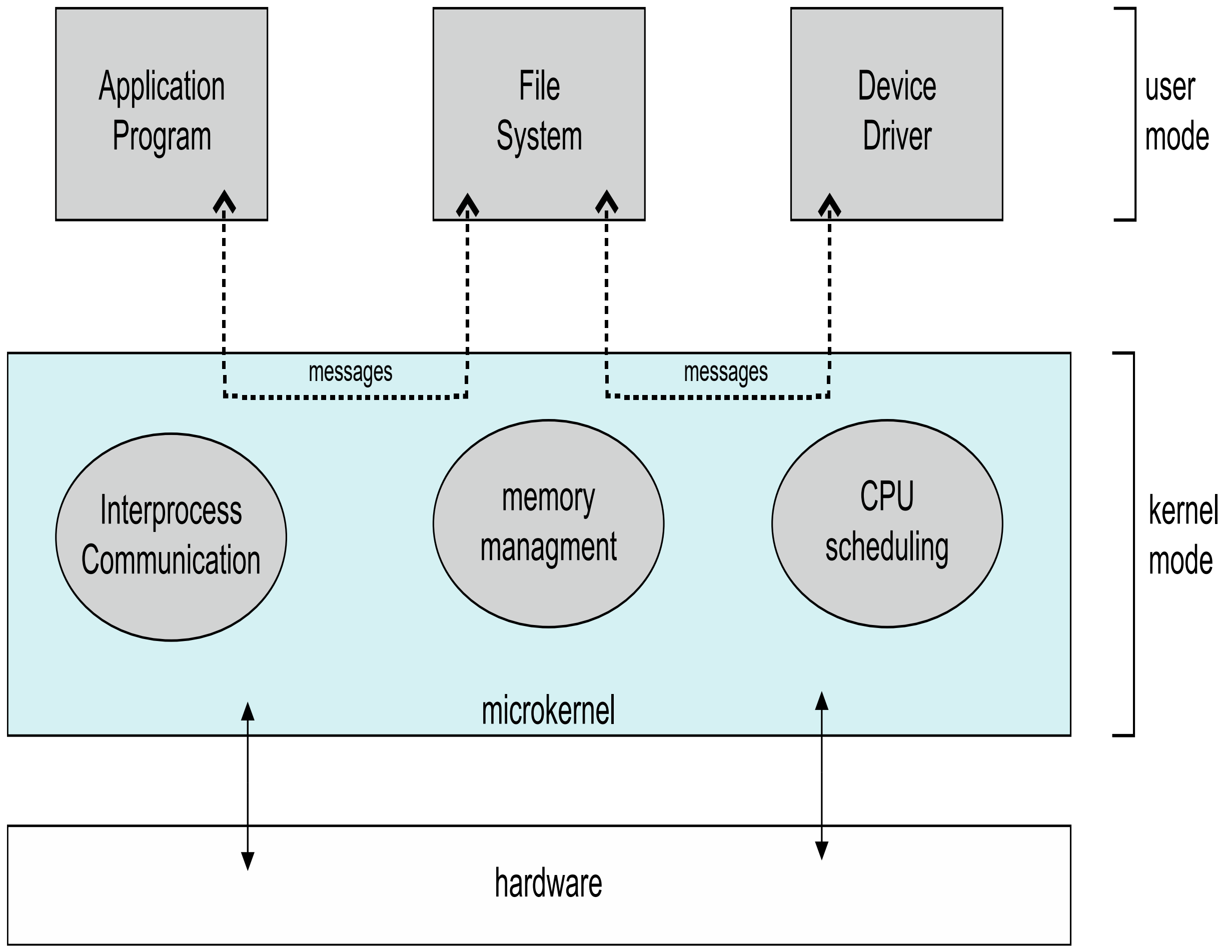
Microkernels – cont.

- Advantages:
  - 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
- Disadvantages:
  - Performance overhead of user space to kernel space, as well as user space to user space communications
- Note: A very good read is the Tanenbaum vs Torvalds debate; https://en.wikipedia.org/wiki/Tanenbaum%E2%80%93Torvalds_debate



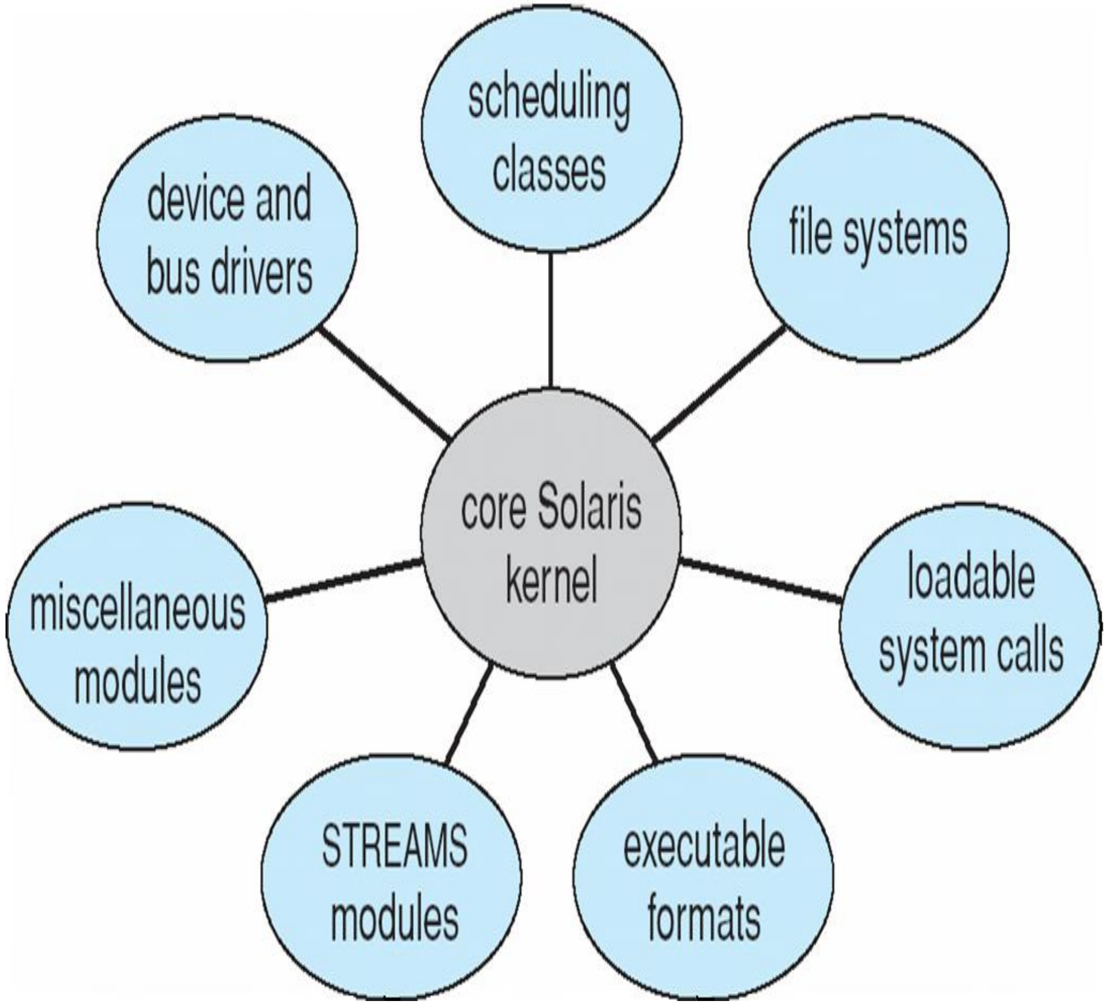
Microkernels – cont.

- Other examples:
  - Tru64 (aka Digital Unix)
  - Minix (Tanenbaum)
  - **QNX Neutrino**,
    - An embedded Unix-like real-time OS.
    - Kernel only handles process scheduling, memory management, IPC, low level network comm. and H/W interrupts.
    - Tight coupling between the IPC and scheduler.
      - The Inter-process communication mechanism allows a process waiting for a message to be immediately invoked when a message is sent to it, without waiting for the scheduler.
      - IPC messages are sent according the priority of the receiving process



Modular Subsystems and Loadable Modules

- Many modern operating systems implement **subsystems** and **loadable kernel modules**
  - The kernel has a set of core components, known as subsystems, that are linked (at compile time or load time) to additional services via modules.
    - This is not to be confused with Windows subsystems which are in user-mode, e.g. WSL, windows subsystem for linux
  - Subsystems and modules communicate via known interfaces
  - Modules **may be** loaded as needed within the kernel (preferred, as opposed to compile-time linking)
    - → loadable modules
- Unlike layered architectures, they provide more flexibility in invoking services from other modules.
  - Linux, Solaris, Mac OS X, Windows, etc
- The Solaris system shown has 7 loadable modules.
- Linux has loadable modules.
  - Used for device drivers, file systems, special devices, and many other things.



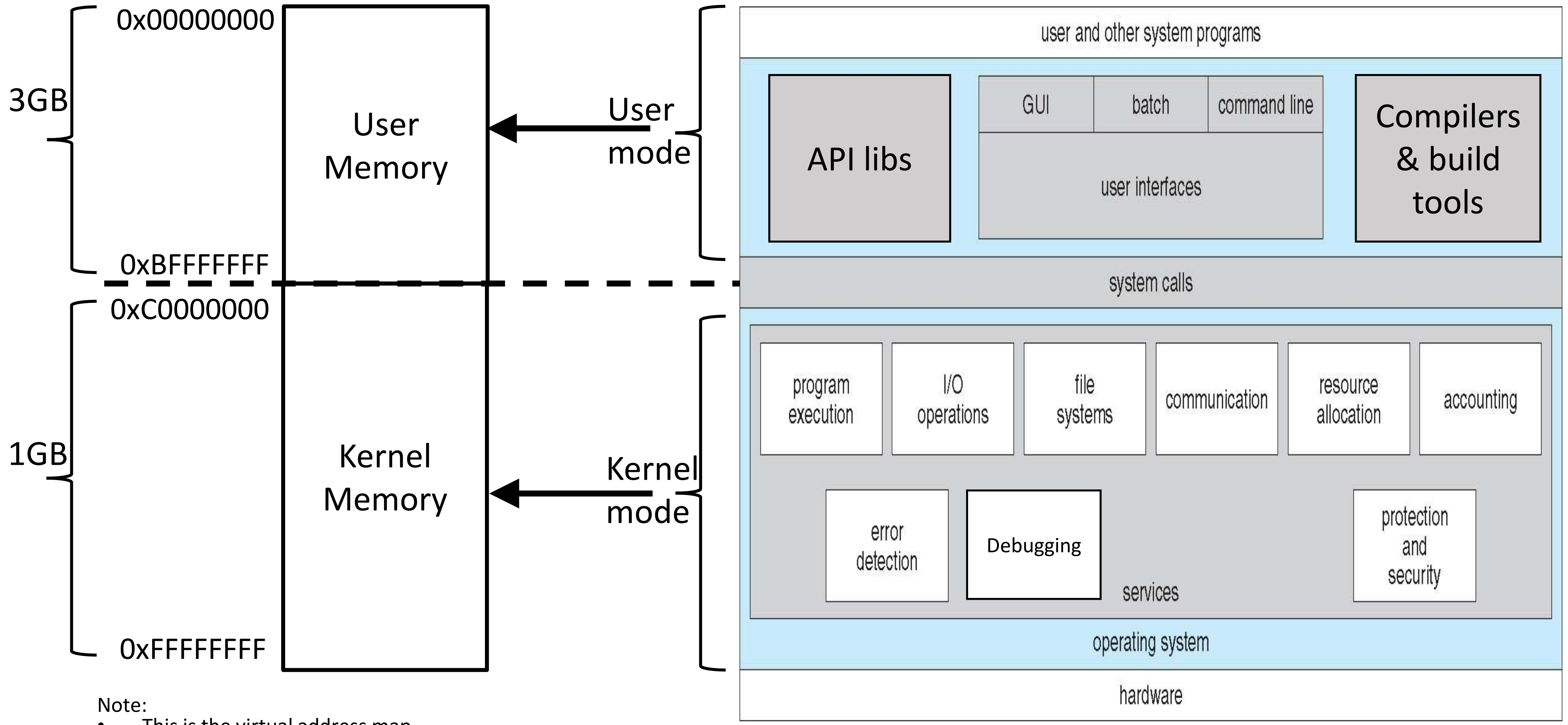
Hybrid Systems

- Most modern operating systems are actually not one pure model
  - Hybrid combines multiple approaches to address performance, security, usability needs
  - The **Linux** kernel is divided into components, known as subsystems.
    - They run in the same address space.
    - Within a subsystem**, modules may be loaded** or unloaded dynamically while the system is running, e.g.

```
sudo insmod ./lab3.ko
```

  - **Windows** has subsystems and modules, but it retains some **microkernel-like** behavior since it has different subsystems running as user-mode processes (e.g. Windows Subsystem for Linux , WSL). At the same time it provides dynamically loadable kernel modules (e.g. device drivers).

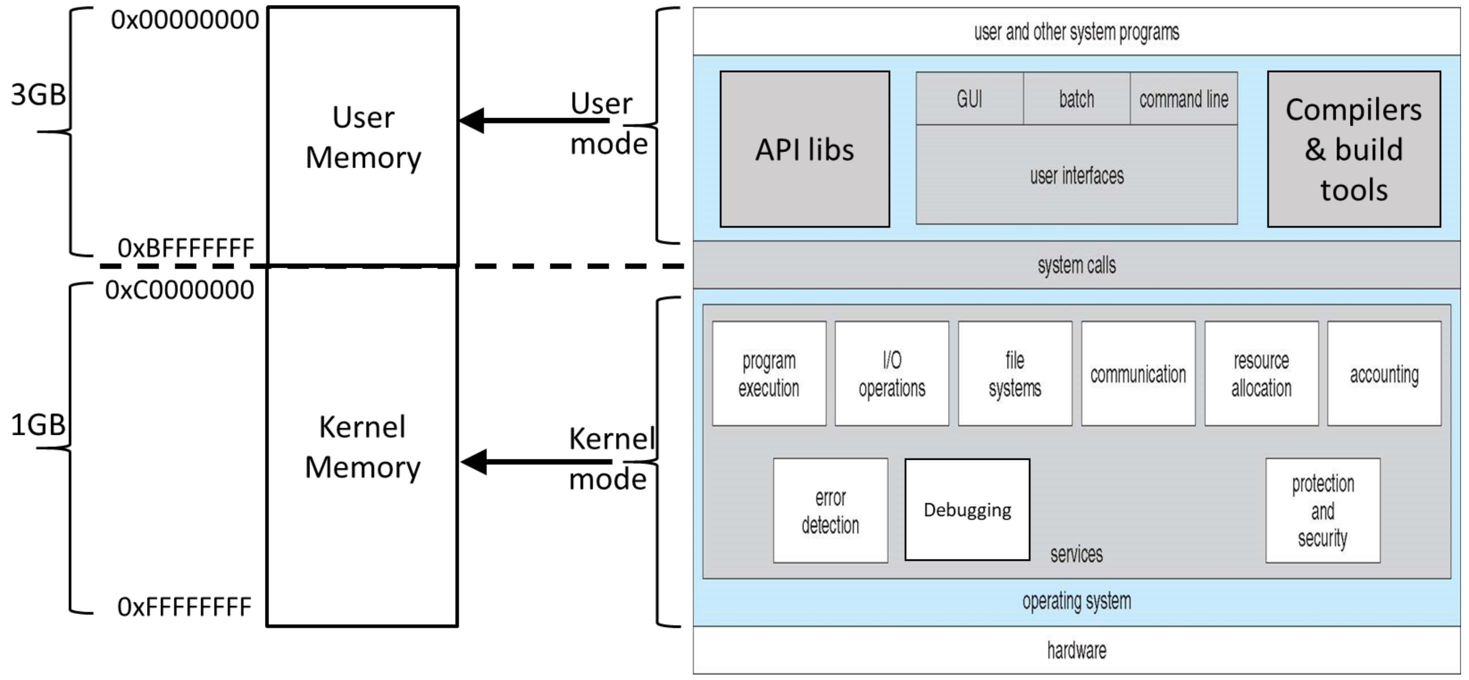
kernel space / user space – 32-bit Linux example



Note:
- This is the virtual address map
- The kernel's physical address map is the lower 1 GB

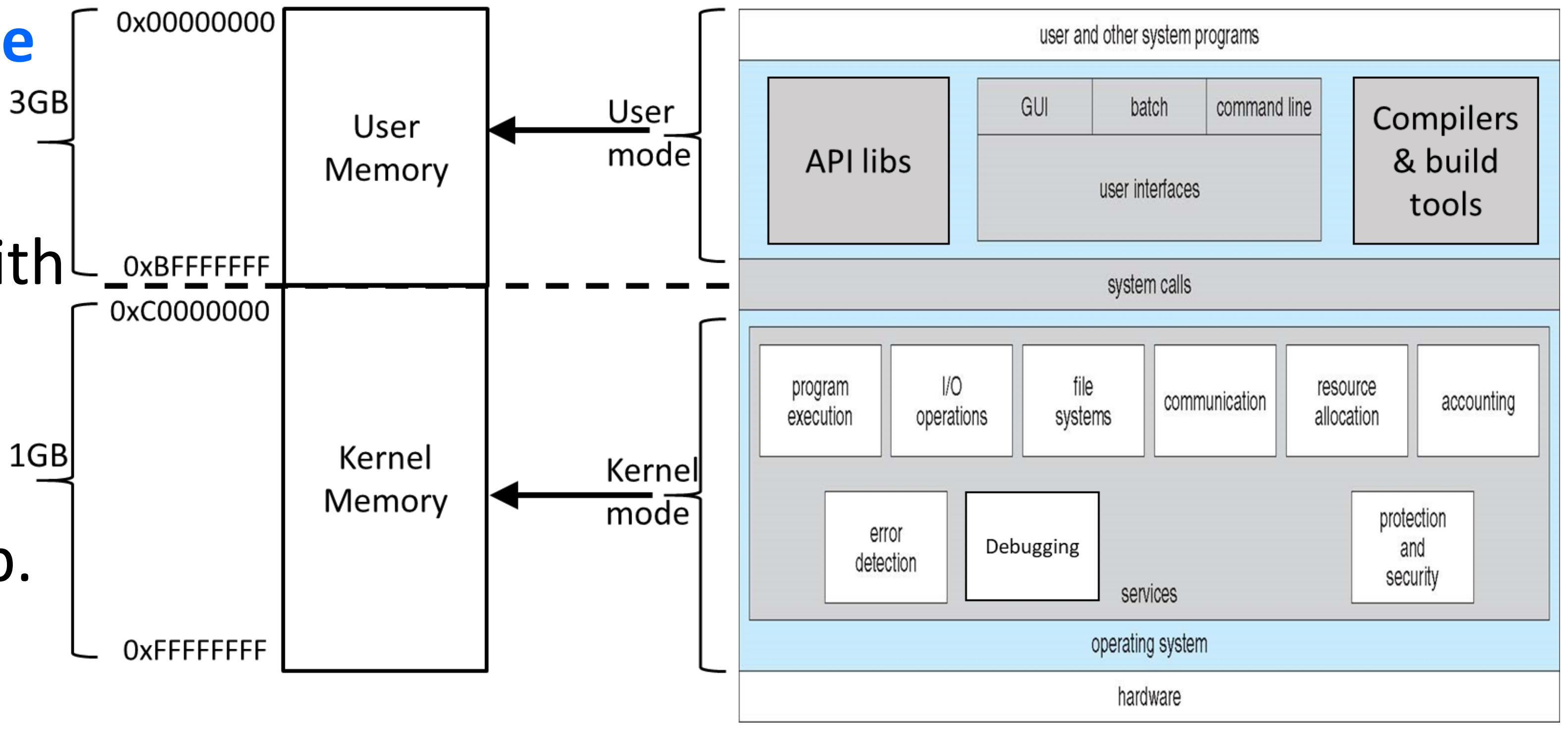
kernel space / user space – 32-bit linux example

- User-mode applications **cannot directly access** (i.e. read, write or execute) memory locations in the kernel memory. This is enforced by a piece of hardware (memory management unit) which we shall study in weeks 8 and 9.
  - If a user program attempts to call a function in kernel memory or read a variable in kernel memory → an **illegal operation interrupt** occurs (by MMU).

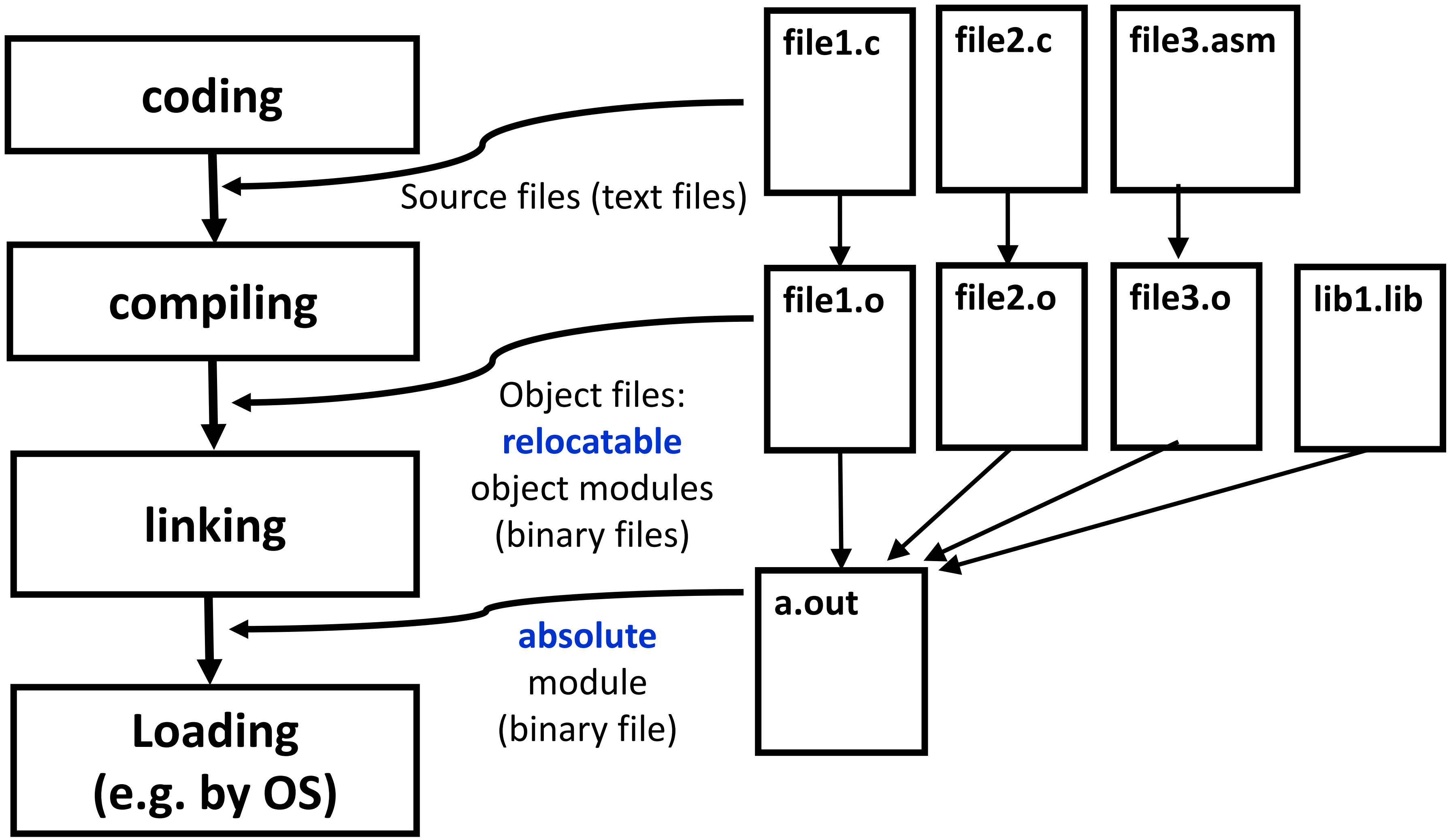


kernel space / user space – 32-bit linux example

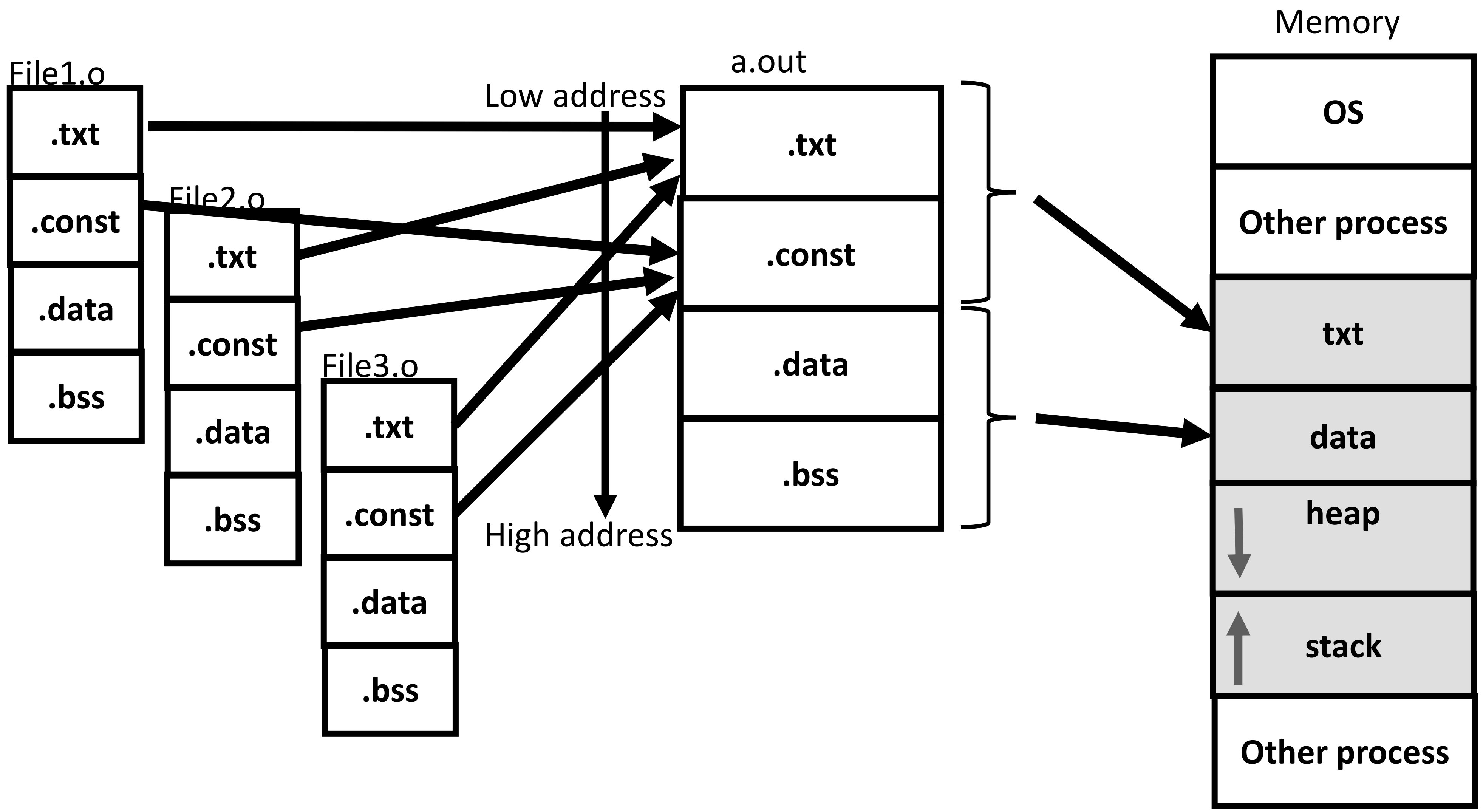
- Thus, the only way for a user program to invoke kernel functions **is via the system call interface**, in which a trap machine instruction is invoked, with the correct paramaters passed via registers/memory/stack prior to invoking the trap.

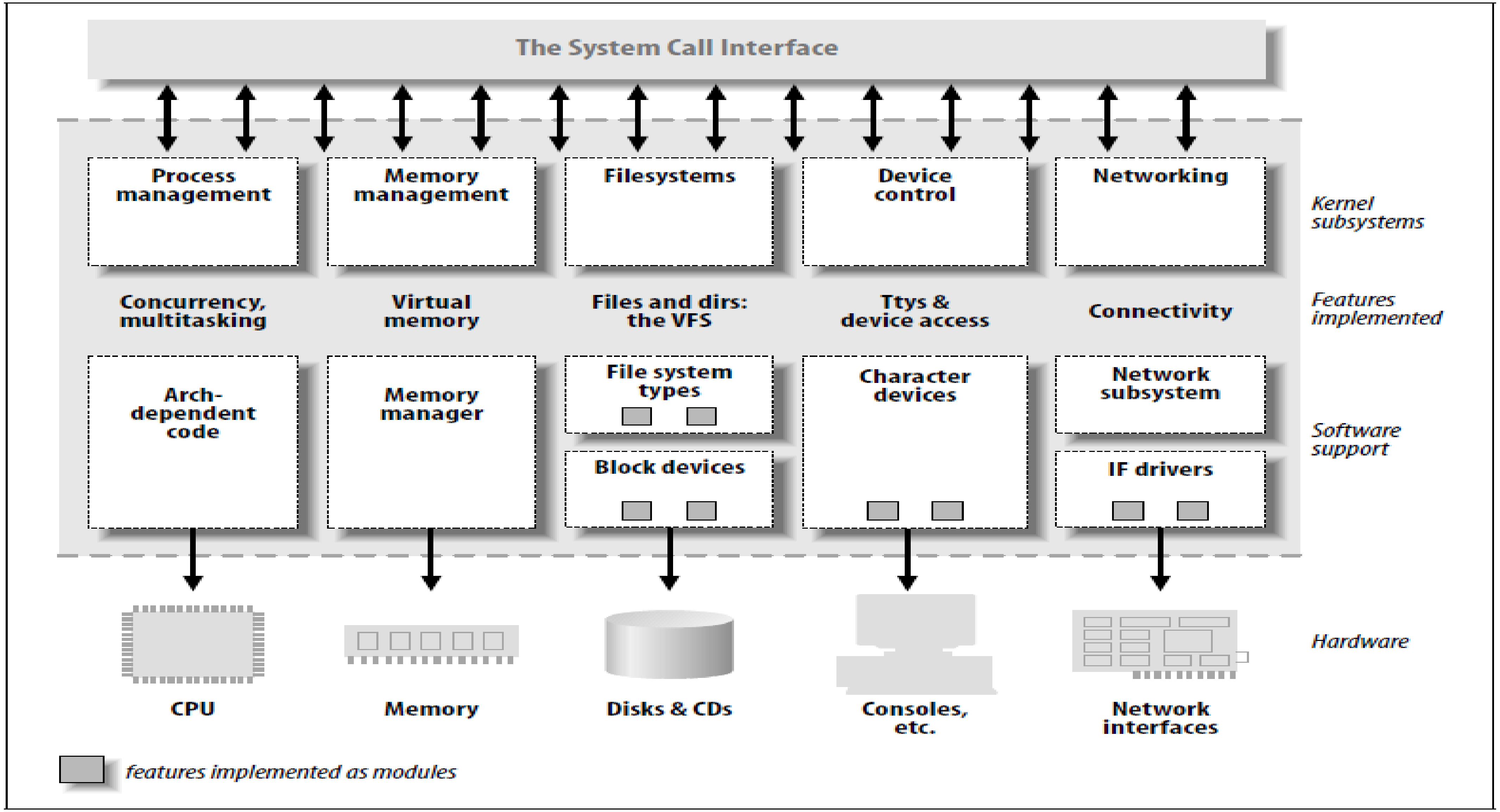


Program translation – (user-mode programs)

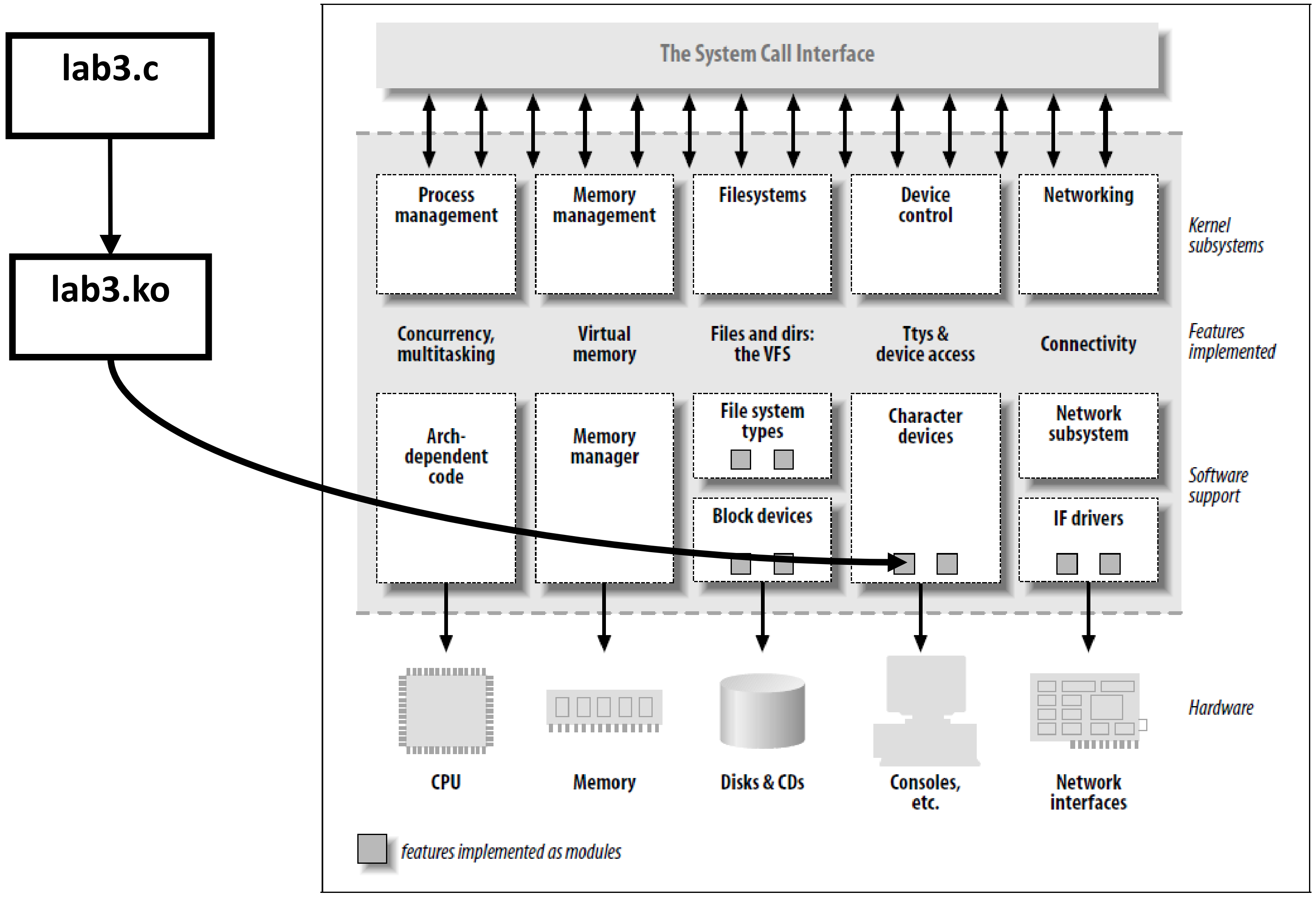


Program translation – linking + loading (user-mode programs)




The System Call Interface
Process management
Memory management
Filesystems
Device control
Networking
Kernel subsystems
Concurrency, multitasking
Virtual memory
Files and dirs: the VFS
Ttys & device access
Connectivity
Features implemented
Arch-dependent code
Memory manager
File system types
Block devices
Character devices
Network subsystem
IF drivers
Software support
CPU
Memory
Disks & CDs
Consoles, etc.
Network interfaces
Hardware
features implemented as modules

Program translation – (kernel modules)



Additional resource

https://www.kernel.org/doc/html/latest/

https://lwn.net/Kernel/

https://lwn.net/Kernel/LDD3/