OS Internals

Andrew H. Fagg

CS 3113

File Abstraction

What does a file represent?

File Abstraction

What does a file represent?

- Fundamental level: it is a sequence of bytes
- Each byte has an address (0, 1, 2, …)
- Sets of bytes might represent some value requiring more space

Representing an Open File

Representing an Open File

Need to know:

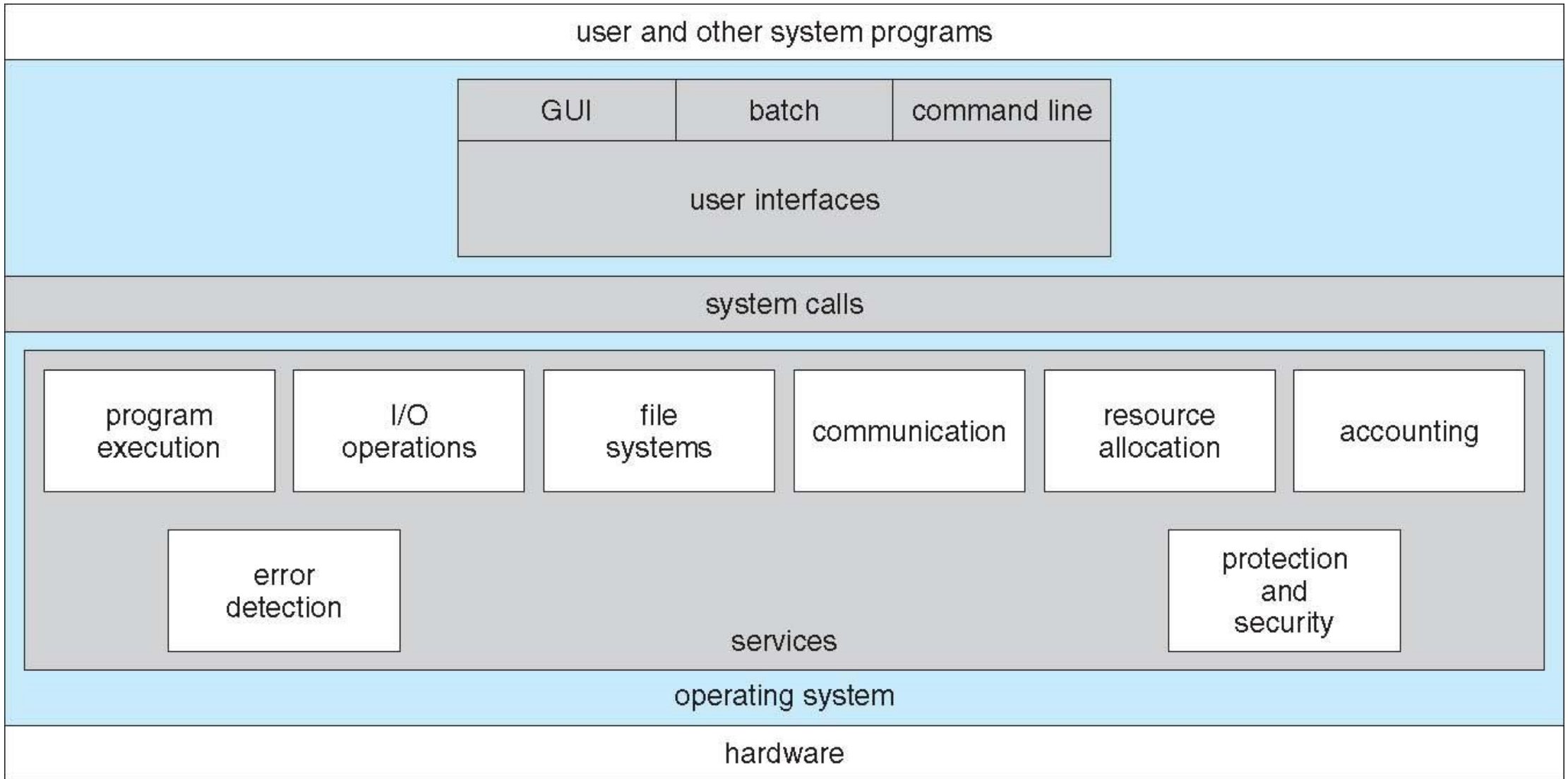
- Where the file is located on the disk
- Which byte is the current point of access
  - Byte number (address) of the next access
  - When we read a byte from the file, this value is returned
    - And the point of access advances by one (increments)
  - When we write a byte to the file, the new value is written to the point of access
    - And the point of access increments

Streams

Generalization of what we think of as a file: any sequence of bytes

- Communication between processes
- Communication between computers
- A model for some I/O modes (USB, audio, video, shells)
  - In Unix, all communication of byte sequences is done through streams
- Also adds buffering

OS 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

Command Line Interface

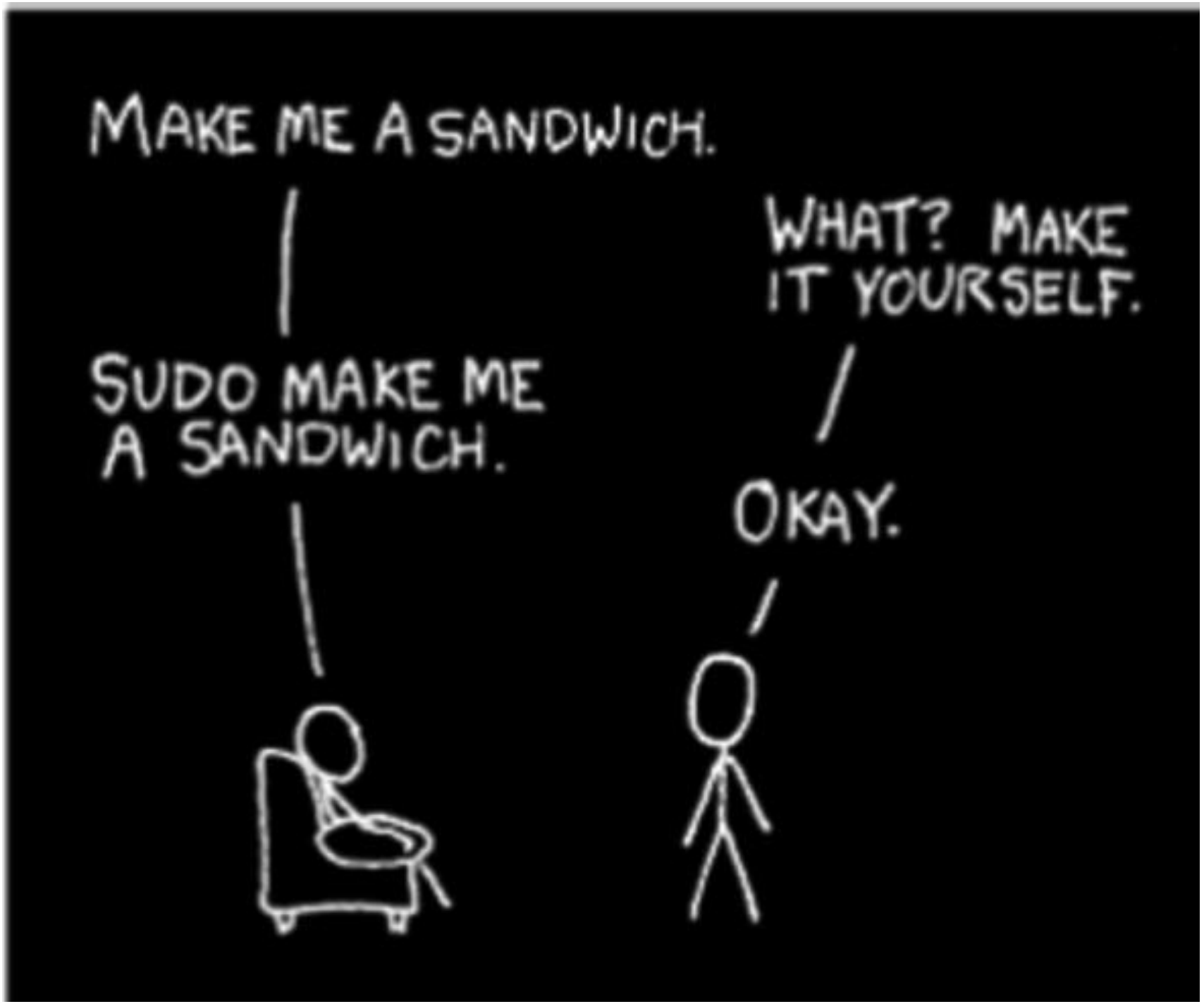
Textual interface for user to interact with the system

- In some OSes, it is a kernel component, in other OSes it is a system program
- Parses text input, which includes commands and parameters and executes the command
  - List a directory, create a file, etc.
  - Some commands are built in, others are independent executables

Command Line Interface

Most of you are used to graphical user interfaces (GUIs)

- A shell program is an implementation of a CLI
  - Common in Linux today: bash shell
- CLIs existed long before GUIs
- GUIs generally provide access to a shell program
- With practice, one can often be much more efficient working with a CLI than a GUI


MAKE ME A SANDWICH.
WHAT? MAKE IT YOURSELF.
SUDO MAKE ME A SANDWICH.
OKAY.

xkcd.com

Command Line Interface

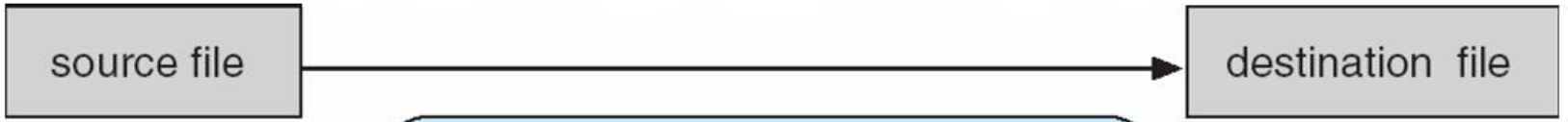
Demo

User-Space Programs: Accessing Resources

- A user-space program has a set of its own resources including the stack, heap and process state
- But, the program often needs to access resources that are shared in some way with other programs
- It is the job of the operating system to make this sharing as safe as possible

Example: Copying a File

Copy all of the bytes contained within a file to a file of a different name



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

System Calls

- Programming interface to the services provided by the OS
- Typically written in C or C++
- Mostly accessed by programs via a high-level ***Application Programming Interface (API)*** rather than directly through system calls
- 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)

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 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.

Function Calls

A function is just another block of code in program memory. The call process is:

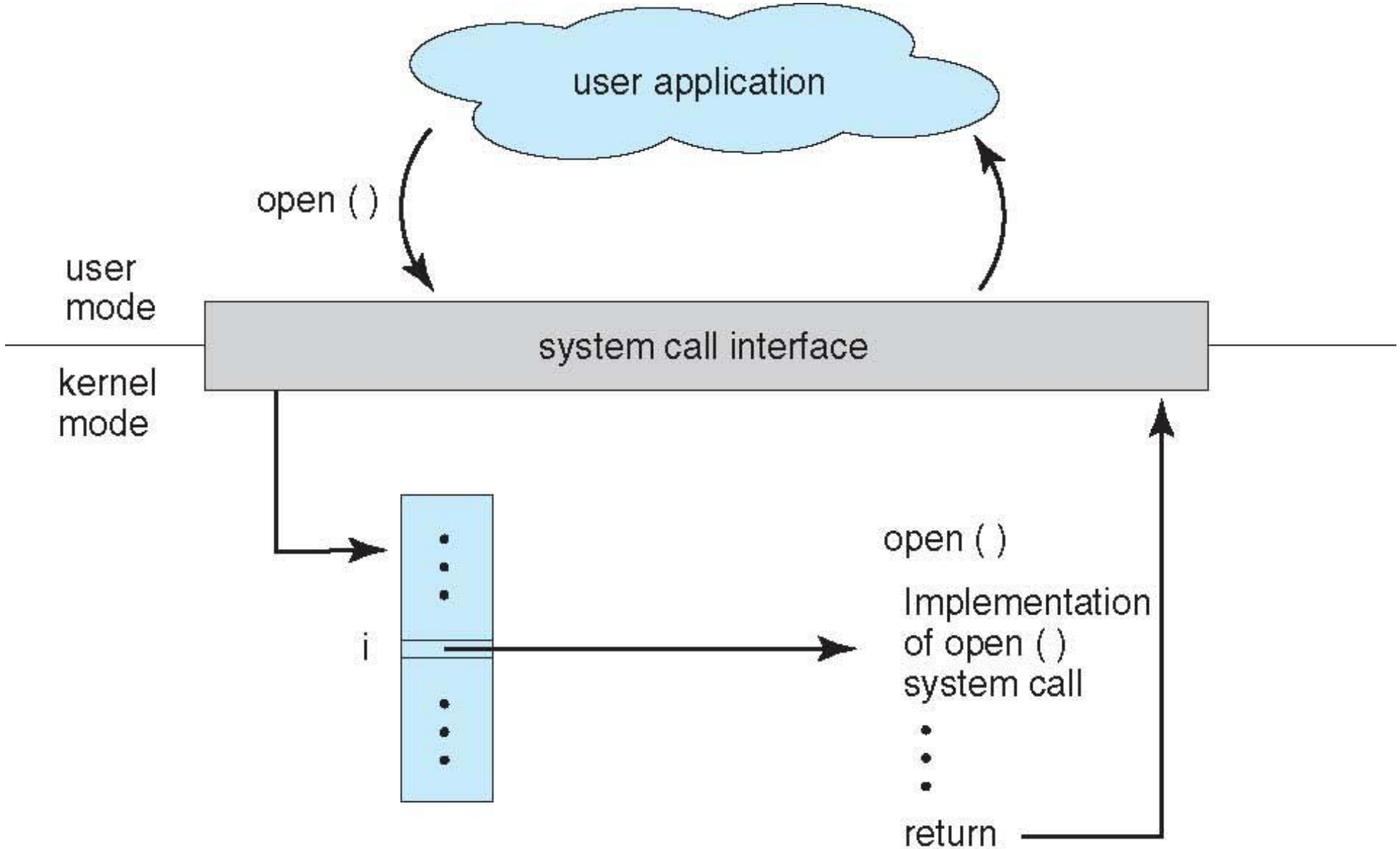
- Store current state of the processor (registers, including program counter) onto the stack
- Store function arguments (in registers or on stack)
- “Jump” to the start of the function (set the PC)
- Execute the function call
- Reset the PC to its original value
- Pop processor state off of the stack
- Calling function then continues execution

System API

- The API is a set of functions that provide a convenient interface to the available system calls. You will typically work at this level
- However, the OS must ensure that the system calls cannot be abused/exploited
  - Want to avoid jumping into arbitrary points in the code
  - Want to prevent user space processes from accessing data structures used in the system calls

System Call Process: All About Safety

- There is one common entry point for all system calls: this is done through the *sys call "trap"* instruction
- Each system call is referenced using a unique integer
- Each has its own set of arguments to be transferred
- The trap instruction switches the processor state from user to kernel mode
- The trap handler function then translates the sys call number into an appropriate function call


user application
open ( )
user mode
kernel mode
system call interface
i
open ( )
Implementation of open ( ) system call
return

Using the API

- All system calls from the API will return a value indicating success/status or an error
  - See the system call's man page for details about the meaning of the return value
- `int errno` is a global variable that is set by the kernel side of the system call to provide more details about any error that has occurred
  - Many system calls return -1 to indicate an error; then, you can use `errno` to extract additional meaning
  - Symbol is defined in errno.h

Interpreting errorno

- The value of the integer is hard to use directly (you have to remember what the different codes mean)
- A variety of functions will help with this. For example:
  - Print the specified string + an English translation of errorno:

```
perror(strg);
```

System Calls for File Manipulation

- Create file, delete file
- Open, close file
- Read, write, reposition
- Get and set file attributes

System Calls for Process Control

- Create process, terminate process
- Load, execute
- Get process attributes, set process attributes
- Wait for time
- Wait event, signal event
- Allocate and free memory
- Dump memory (for error analysis)
- Locks for managing access to shared data between processes
- Debugger for determining bugs, single step execution

System Calls for Device Management

- Request device, release device
- Read, write, reposition
- Get device attributes, set device attributes
- Logically attach or detach devices

System Calls for System Monitoring and Control

- Get time or date, set time or date
- Get system data, set system data
- Get and set process, file, or device attributes
- Control access to resources
- Get and set permissions
- Allow and deny user access

System Calls for Communication

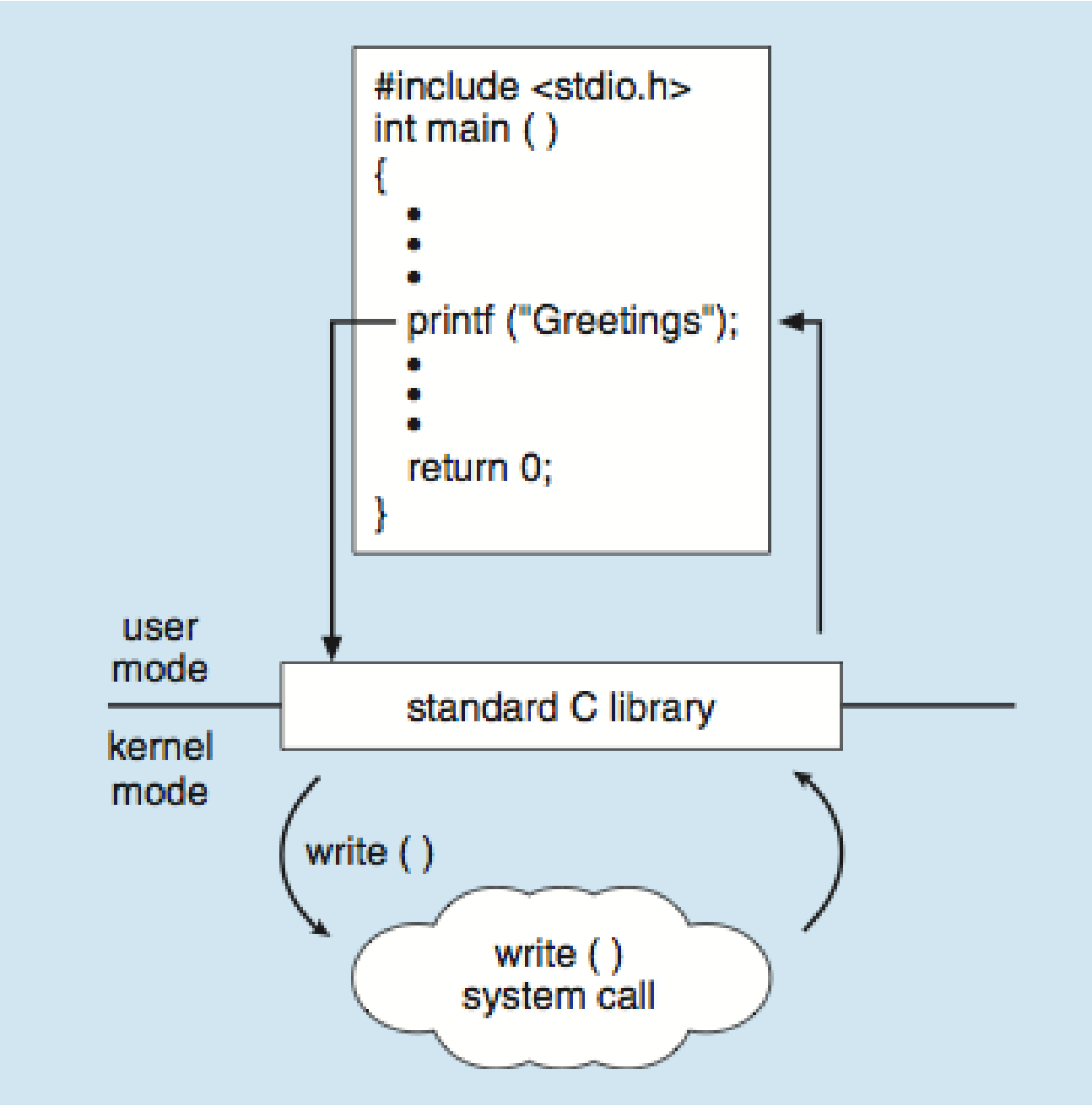
- Create, delete communication connection
- Send, receive messages to host name or process name
- Shared-memory model: create and gain access to memory regions
- Transfer status information
- Attach and detach remote devices

Some System Calls

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

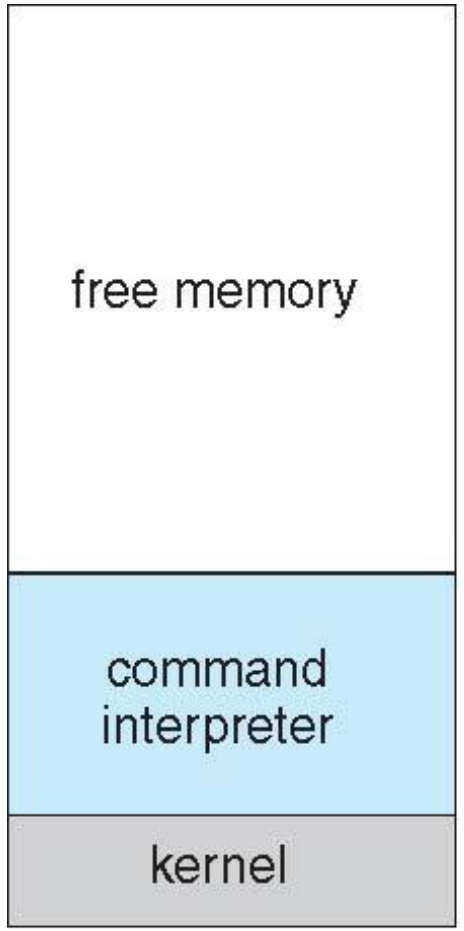
Libraries and System Calls

Unix standard I/O library (stdio) does not provide system calls, but it does call them

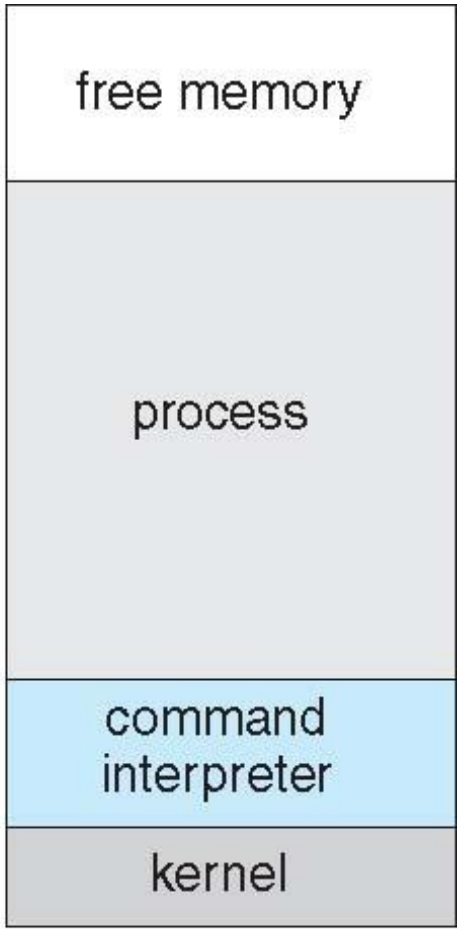


Example: MS-DOS

- Single-task system
- At time of boot, lots of free memory (a)
- Invoke a new program: system call allocates the necessary space in memory and begins program execution (b)



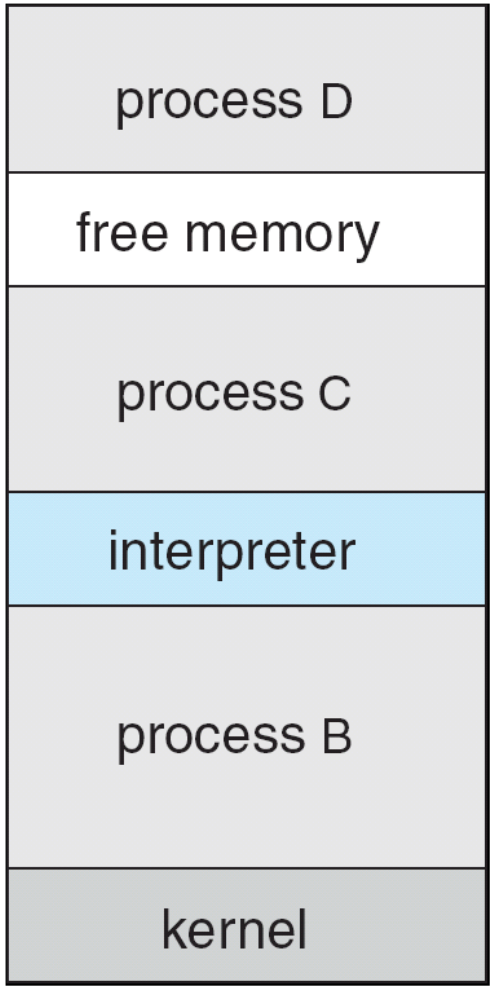
(a)



(b)

Example: FreeBSD

- Flavor of unix
- Multitasking
- 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



Example: Unix File Operations

A simple program:

- Open a file for writing
- Write an integer
- Write a string
- Close the file

open()

```
open(char * pathname, int flags, mode_t mode)
```

- pathname = path + file name to open
- flags = how to open the file (reading, writing, appending, …)
- mode = file permissions
  - Possible permissions: read, write and execute
  - Defined for: the user, a group and everyone on the system
- returns -1 on error
  - or an integer file descriptor if successful

Opening a File

```
int fd = open("out.bin", O_WRONLY | O_CREAT | O_TRUNC,
                                    S_IRUSR | S_IWUSR);
if(fd == -1) {
      printf("Error opening\n");
      exit(-1);
}
```

write()

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

- fd = file descriptor of an open file
- buf = pointer to an array of bytes to send
- count = number of bytes to send
- Returns -1 on error
  - Otherwise, the number of bytes sent

Writing to a File

```
// Write all bytes associated with the int
int value = 42;

if(write(fd, &value, sizeof(int)) != sizeof(int)) {
    printf("Write error\n");
    exit(-1);
}
```

Writing to a File

```
// Write a single character
char c = '#';

if(write(fd, &c, 1) != 1) {
    printf("Char write error\n");
    exit(-1);
}
```

Demo

hexdump

Implementation of an OS

- Early days: assembly language
- Today: a variety of languages, depending on the tasks
  - OS itself: C / C++
  - System programs can be implemented in: shell script, perl, python

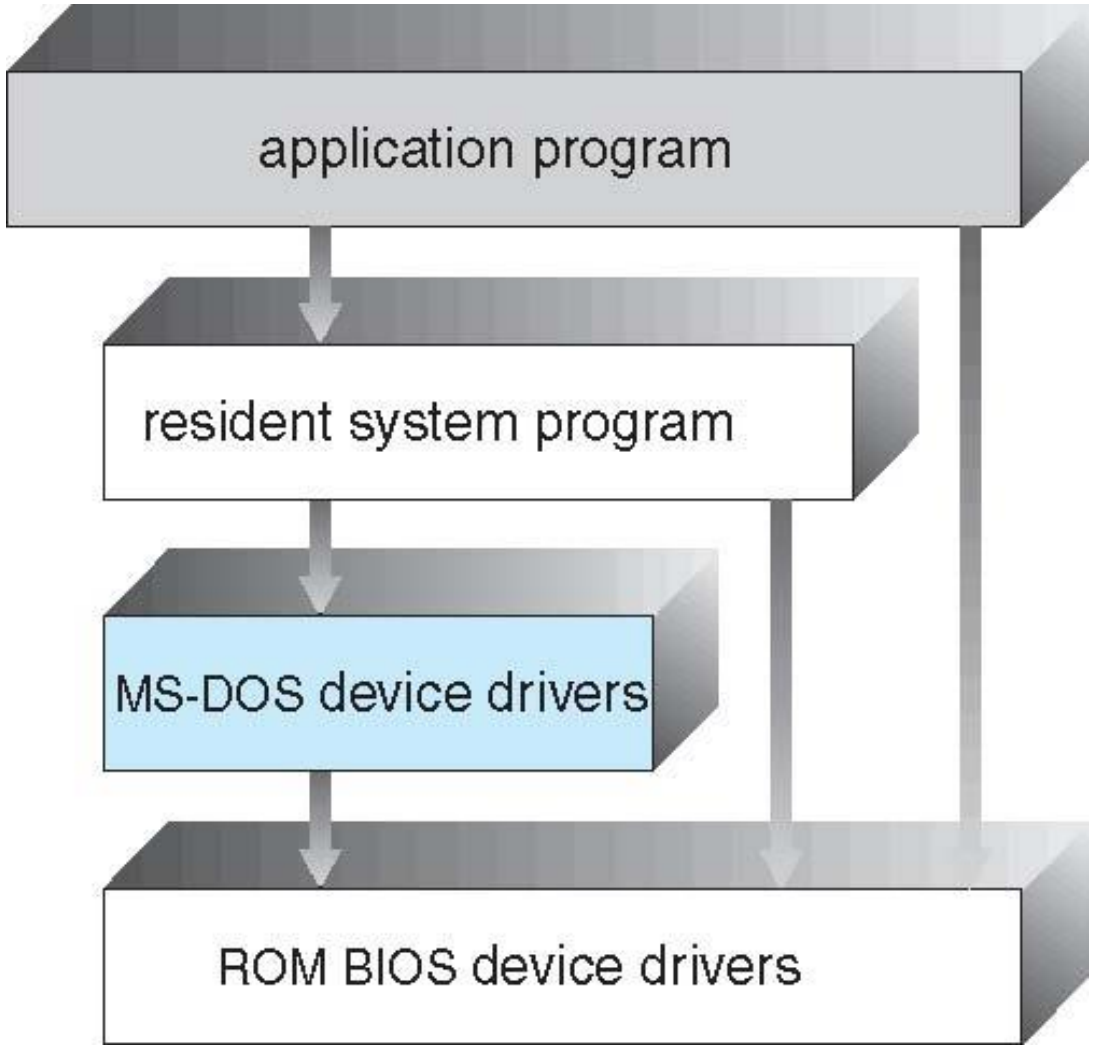
OS Code Structure

Choices depend on a lot of factors:

- History
- Required complexity
- Real-time vs not
- Memory constraints
- Types of tasks to be performed

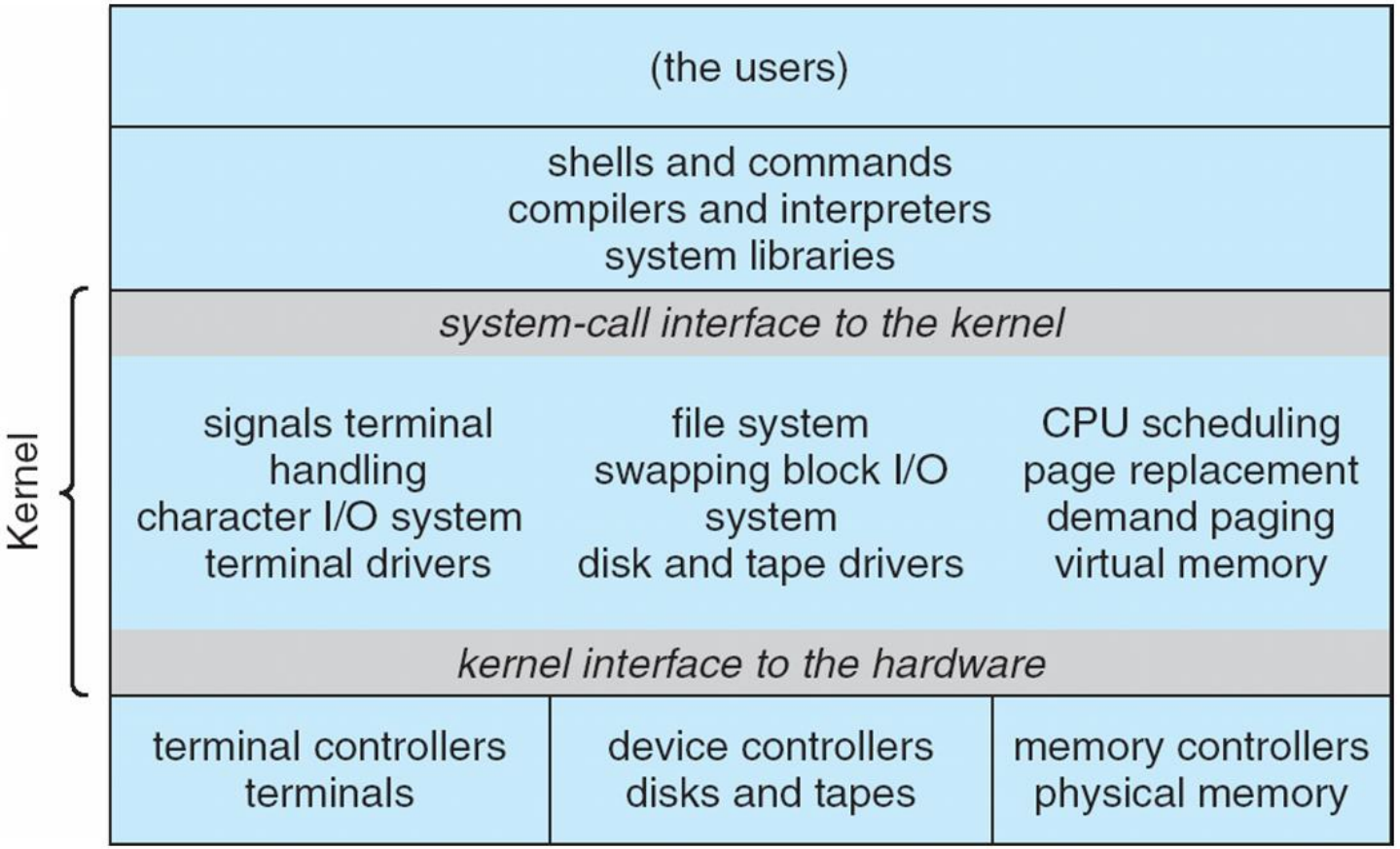
MS-DOS

- Space constrained
- Monolithic
- Simple



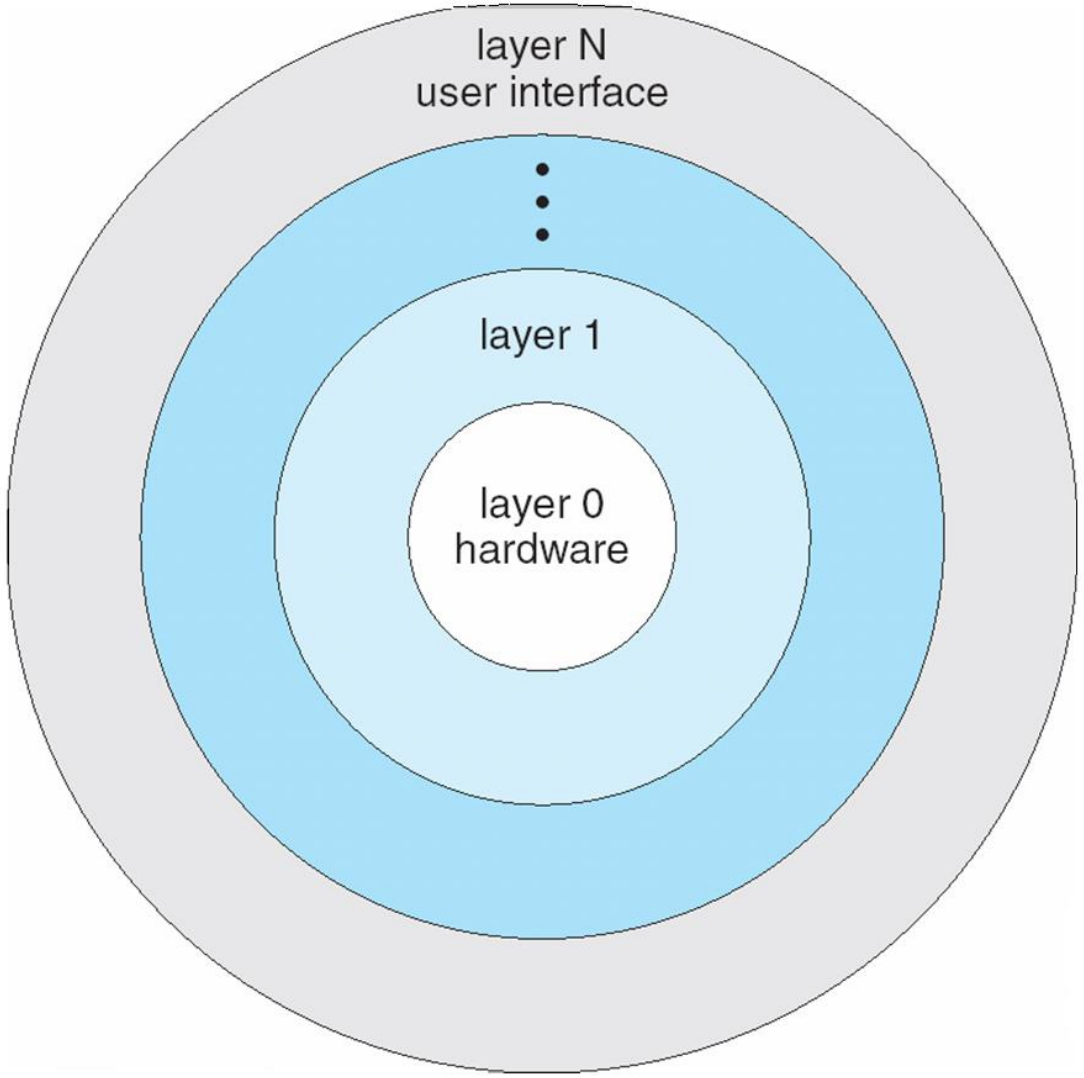
Unix

- Complex
- Early implementations: components not well separated



Layered

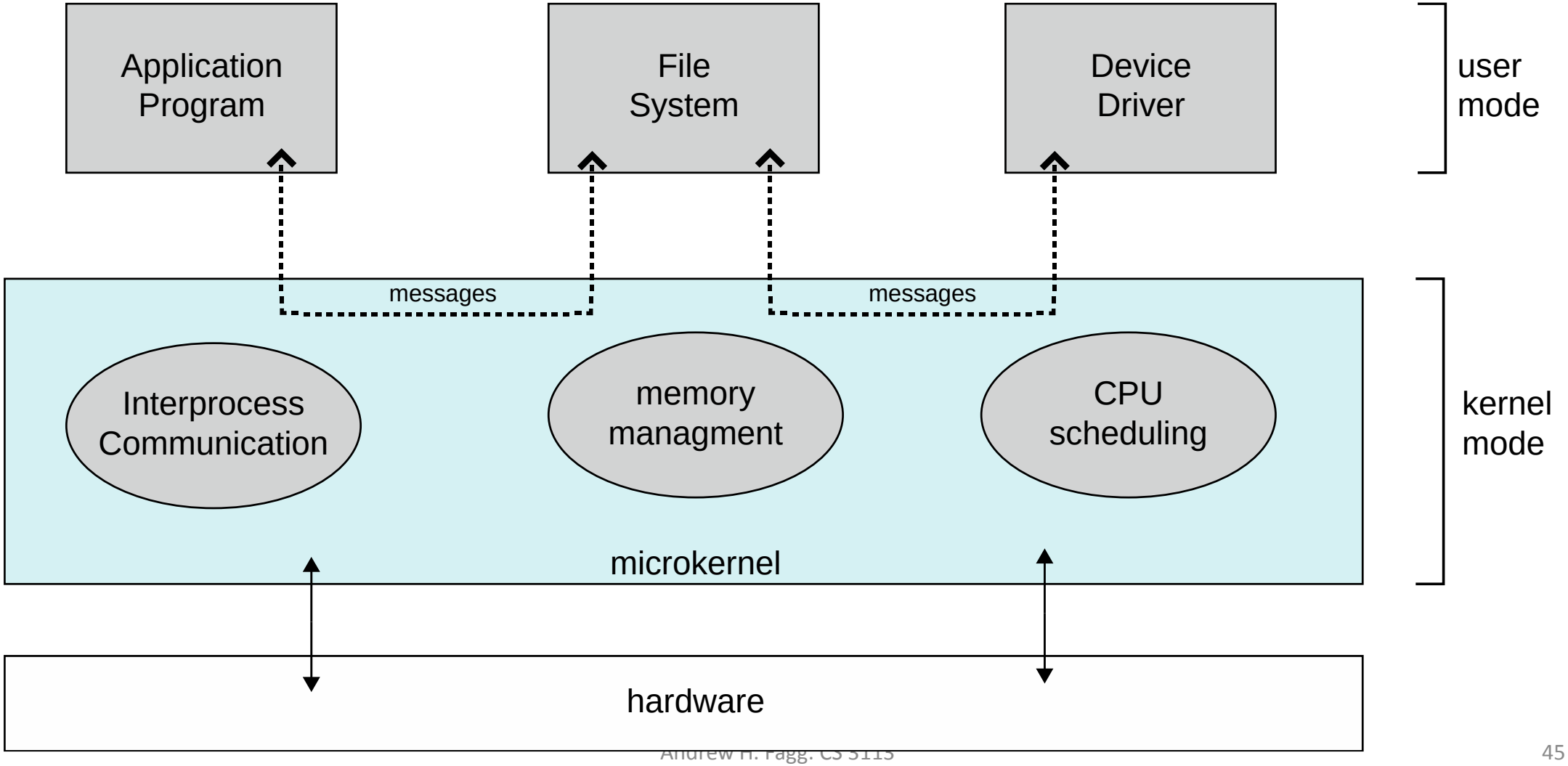
- Inner layers closest to the hardware
- One layer only depends on the layer (or layers) below it
- Possible to test layers k … 0 before testing layer k+1



Microkernel Architecture

- Move as much functionality into user space (even things like the process scheduler)
- Makes the kernel space program small, lightweight and easy to test
- Can be more secure than a monolithic kernel
- Challenge: user space system programs must use system calls to perform kernel-level tasks (this is a lot more expensive than executing a function).

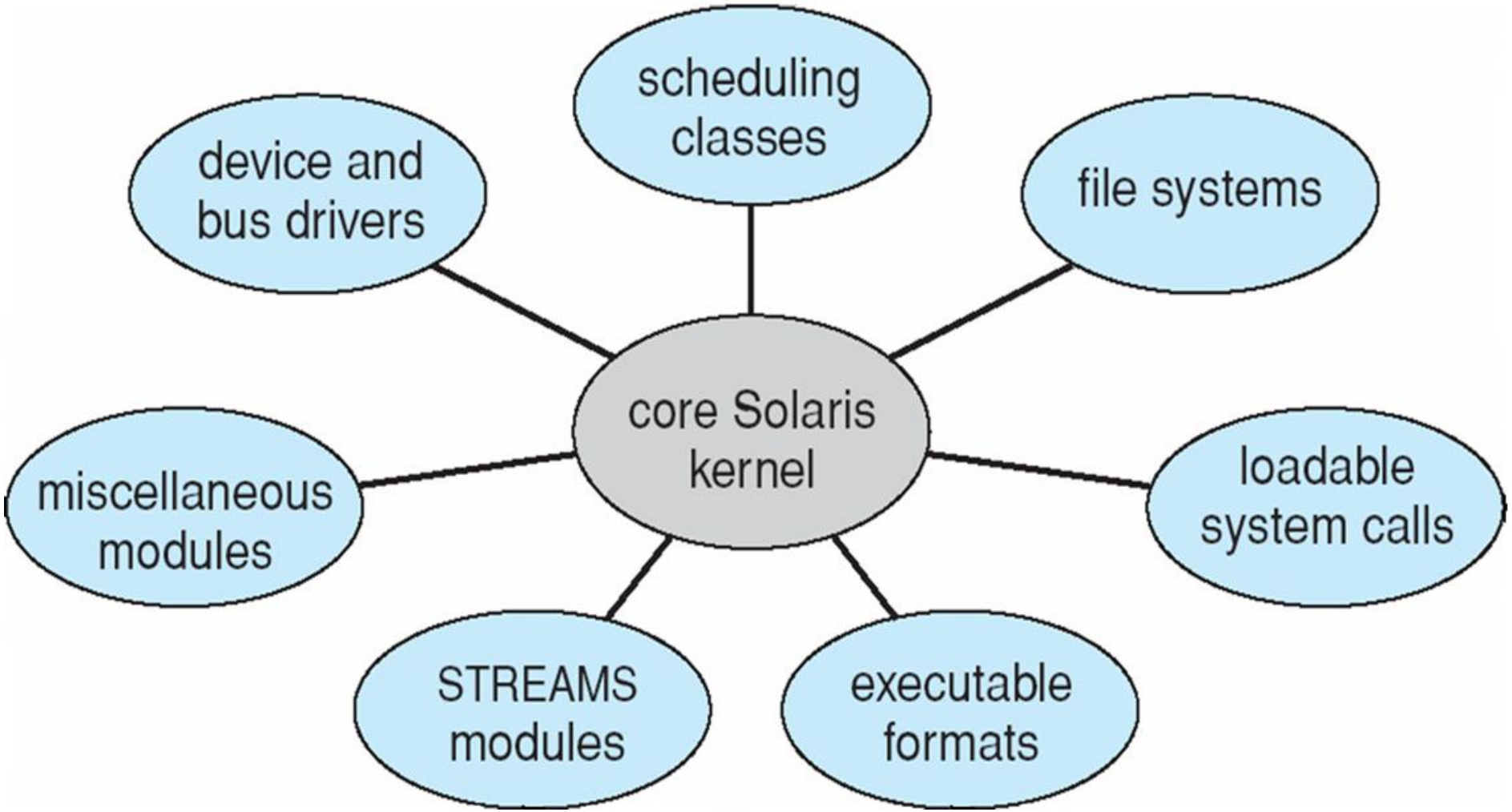
Microkernel Architecture



Kernel Modules

- Many modern operating systems 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
- Overall, similar to layered approach, but more flexible
  - Linux, Solaris, etc

Modular Kernels



System Boot Process

- When power initialized on system, execution starts at a fixed memory location
  - Firmware ROM used to hold initial boot code
- Operating system must be made available to hardware so hardware can start it
  - Small piece of code – bootstrap loader, stored in ROM or EEPROM locates the kernel, loads it into memory, and starts it
  - Sometimes two-step process where boot block is at fixed location loaded by ROM code, which loads bootstrap loader from disk
- Common bootstrap loader, GRUB, allows selection of kernel from multiple disks, versions, kernel options
- Kernel loads and system is then running

Coming soon …

- Writing programs in the unix environment