CSC 139
Operating System Principles

Chapter 2
OS Structures

Herbert G. Mayer, CSU CSC

8/1/2022

Syllabus

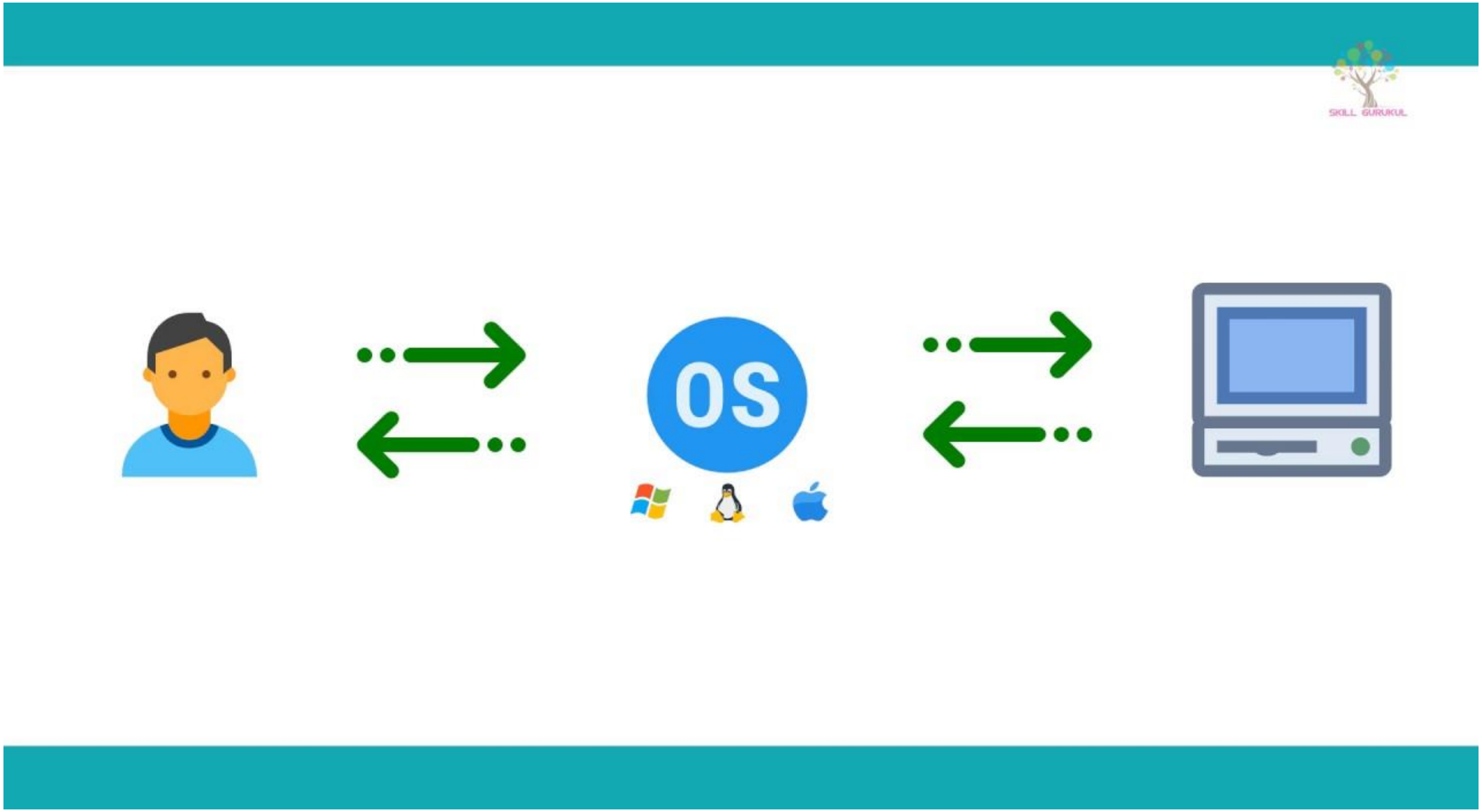
- OS Services
- User OS Interface
- System Calls
- Types of System Calls
- System Programs
- OS Design and Implementation
- OS Structure
- OS Debugging
- OS Generation
- System Boot

Objectives

- Discuss how to boot an OS
- Outline methods how a user interacts with OS
- Describe services an OS provides to users, to processes, and to other systems
- Discuss System Calls
- Present ways of structuring an operating system
- Explain how a typical OS is installed and customized

Operating System Services

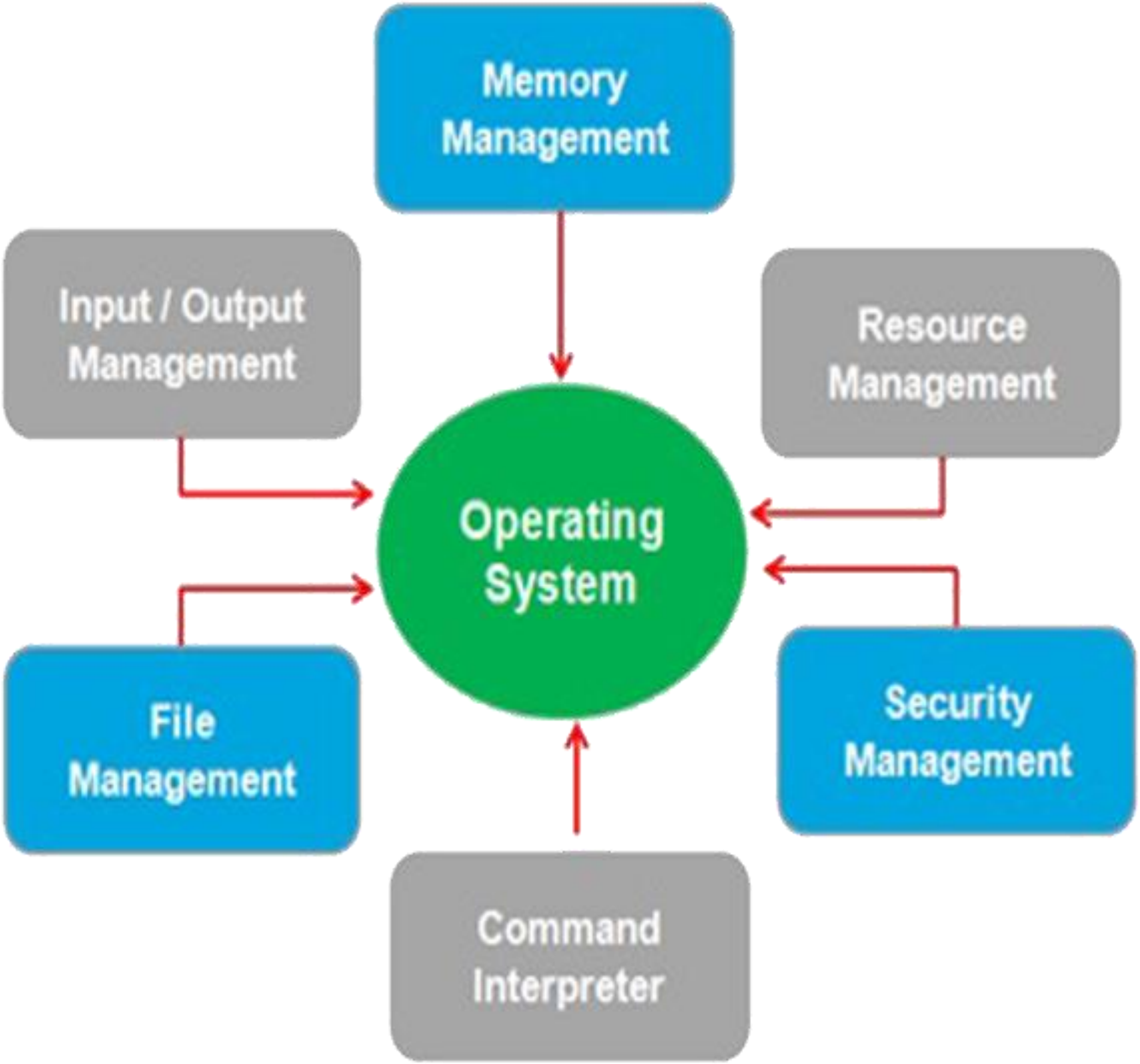
Graph describes essence of OS use, with:
User, Computer, OS at the center!



Operating System Services

- **OS provides to user a compute environment for execution of programs, and related services**
- **Sample OS services for user:**
  - **Provide a computing interface: Practically all OSes have a user interface; common user interface acronym is UI**
    - **Examples: Text-based AKA Command-Line Interface (CLI), Graphical User Interface (GUI), Batch Interface**
  - **Manage program execution: OS must 1. load a program into memory; 2. run that program; 3. end execution, and then reclaim resources**
  - **End normally, or else abnormally if error: increase speed, decrease size, correct error if possible, etc.**
  - **Offer I/O operations: A running program may require I/O, which will involve a file, or physical I/O device**

Operating System Services



Operating System Services

OS provides specific functions to user, such as:

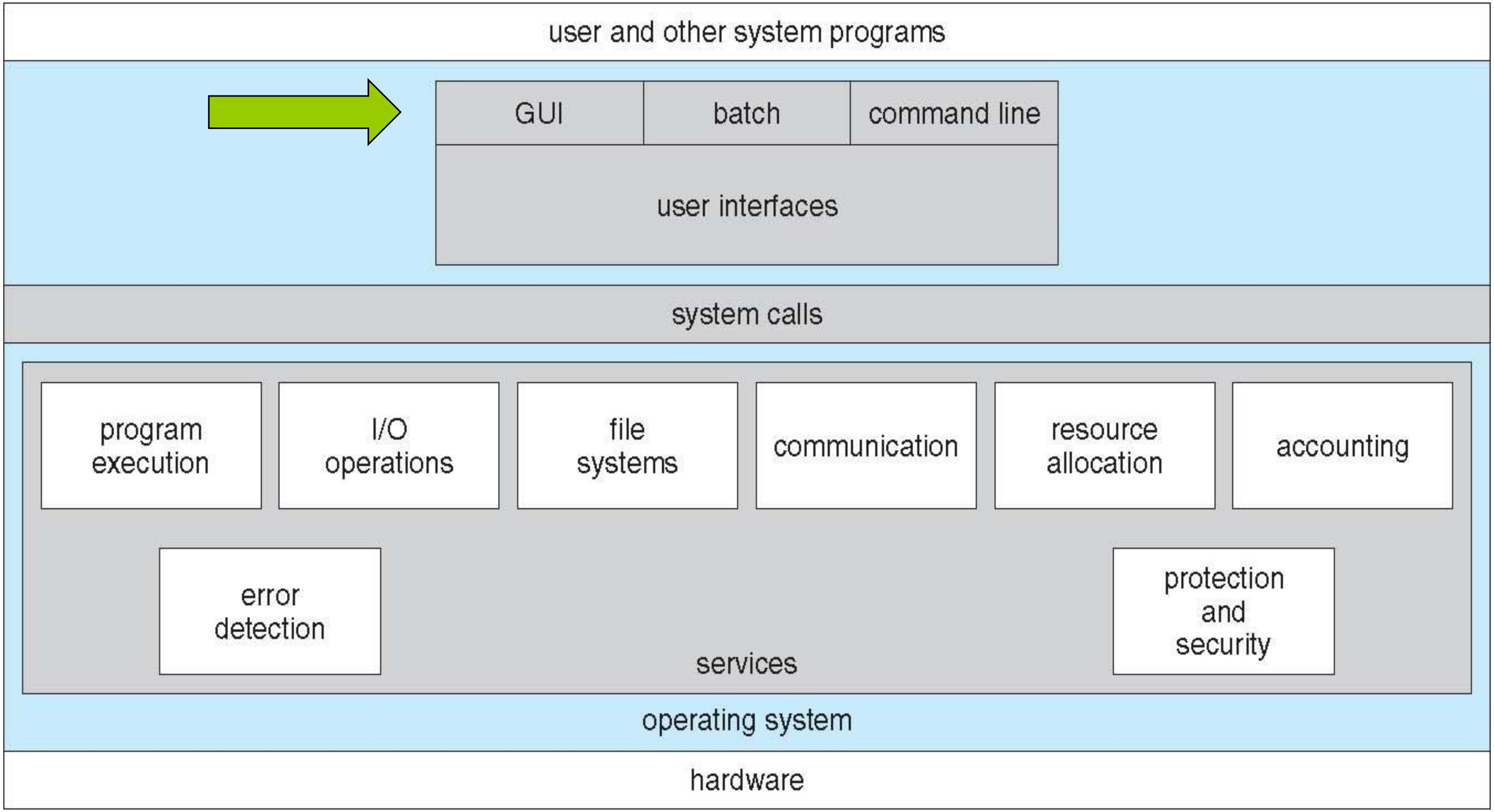
- **File system: Programs read and write files, create and delete directories, search them, list file Information, grant permission**
- **Communications: Processes exchange information, on the same computer or across networked computers**
  - **Process communication may be via shared memory or through message passing (packets moved by the OS)**
- **Error detection: Numerous kinds of possible errors can arise:**
  - **Errors may occur on CPU, in memory, I/O devices, user program, on network**
  - **For each type of error, OS takes appropriate action whenever possible to ensure correction, data consistency**
  - **At least issues complete report, then aborts program**

Operating System Services

Plus: OS ensures efficient operation and use of computing system via rational, fair resource sharing!!

- **Resource allocation: When multiple users or multiple jobs run concurrently, resources still are individually allocated**
  - **Resource types: CPU, main memory, file storage, I/O devices**
  - **Some resources exclusive (e.g. printer); some multiplexed (disks)**
- **Accounting: Track which users consume how much and which kind of compute resources; user can be piece of SW**
- **Protection and security: Owners of information in multiuser or networked computer system wish to control use of that information; concurrent processes should not interfere**
  - **Protection ensuring that all access to system resources is controlled in a way that enforces data- and ownership integrity**
  - **Safeguarding system from outsiders by requiring user authentication, defending I/O devices from invalid accesses**
  - **Must ID illegal attempt! Malicious as well as accidental ones**

Operating System Services



User-OS Interface Via CLI

Command line interpreter (CLI) allows direct, textual command entry, displays textual results:

- **Command either implemented in kernel, or via system program**
- **Sometimes multiple flavors implemented, i.e. various shells**
- **CLI fetches a command from user command line, then initiates its execution or interpretation**
- **And generates a response; some OSes in minimal form, AKA terse communication; e.g. Unix + Linux**
- **Sometimes commands are built-in OS services**
- **Other times they are names of programs**
  - **If former, OS is more voluminous ☹ appears more complex**
  - **If latter, adding new or removing old features doesn’t require OS shell mods ☺**

Bourne Shell CLI

Default

New Info Close Execute Bookmarks

Default | Default

```
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-OS Interface

- **User-friendly desktop interface: a key OS requirement!**
  - **Usual devices: mouse, keyboard, monitor, printer, touch pad . . .**
  - **Icons represent files, programs, actions, directories, etc.**
  - **Mouse clicks over objects in the interface cause actions! . . .**
    - **Or provide information, options to execute functions, open directories, etc.**
  - **Mouse invented long ago at Xerox PARC; see [7]**
- **Many 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; see [4] about Aqua**
  - **Unix and Linux have CLI with optional GUI interfaces (CDE, KDE – Kool [sic.] Development Environment, GNOME)**
  - **CDE: Common Desktop Environment, see [5]**

Touchscreen Interfaces

- **Touchscreen devices provide new interfaces**
  - **Mouse not even applicable!**
  - **Actions and selection based on touch**
  - **Virtual keyboard on screen for text entry**
- **Voice commands**
- **No mind-reading OS yet ☺**

Mac OS X GUI


Grab File Edit Capture Window Help
15:06 EDT Mon 2 Jul
fig-dir
Address Book
Apple Computer Inc.
Dictionary and Thesaurus
op•er•at•ing sys•tem
noun
the software that supports a computer's basic functions, such as scheduling tasks, executing applications, and controlling peripherals.
PCalc
Trash

System Call & API

- Programming interface in OS: Executes OS-provided system services; these are higher-level SW modules, rather than low-level HW instructions
- System services typically written in HLL, C or C++
- Accessed by programs via high-level Application Programming Interface (API); rather than via direct system call
- Common APIs are:
  - Win32 API for Windows; e.g. see: [10]
  - POSIX API for POSIX-based systems, including virtually all versions of UNIX, Linux, Mac OS X
  - And Java API for Java virtual machine (JVM)

System Call & API

An API between two SW modules is:

- Akin to a “contract” between the two, such that if the client makes a request in a specific format, it shall get a proper response in a defined, specific way
- When programming some applications, an API simplifies work by abstracting the underlying implementation, only exposing objects or actions the developer (or the running SW) needs
- All else remains hidden . . .
- An instance of information hiding! A sound SWE principle

System Call & API

An Application Programming Interface (API)

- **Provides OS services, hides detail of how service actions are accomplished**
- **Information Hiding: Hide what's not necessary!**

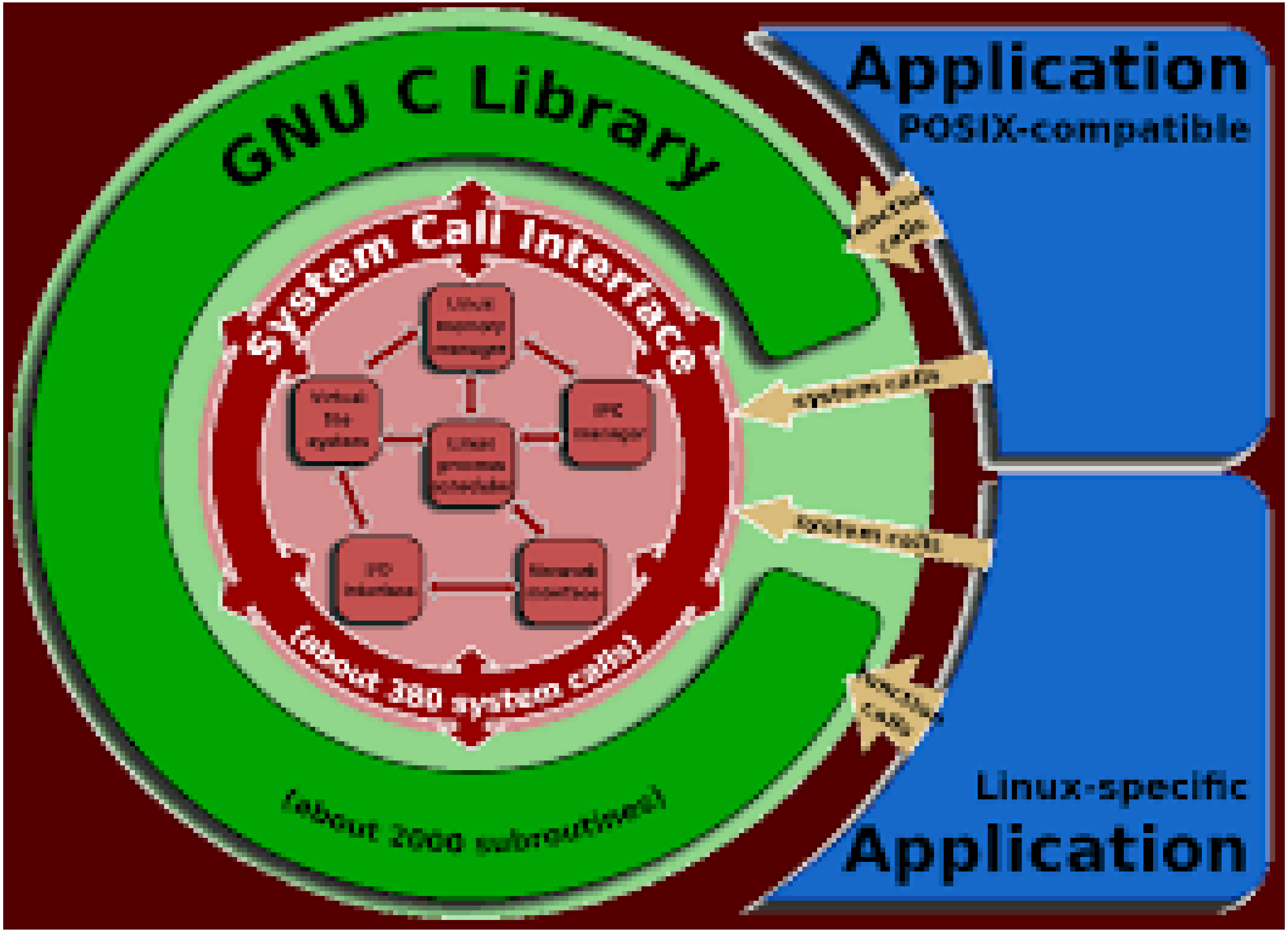
API definition from Wiki:

- **An API is a computing interface that:**
- **. . . handles interactions between multiple software intermediaries**
- **API defines kinds of calls or requests that can be made, and . . .**
- **Documents the data formats to be used, the conventions to follow under any one particular OS**

System Call & API

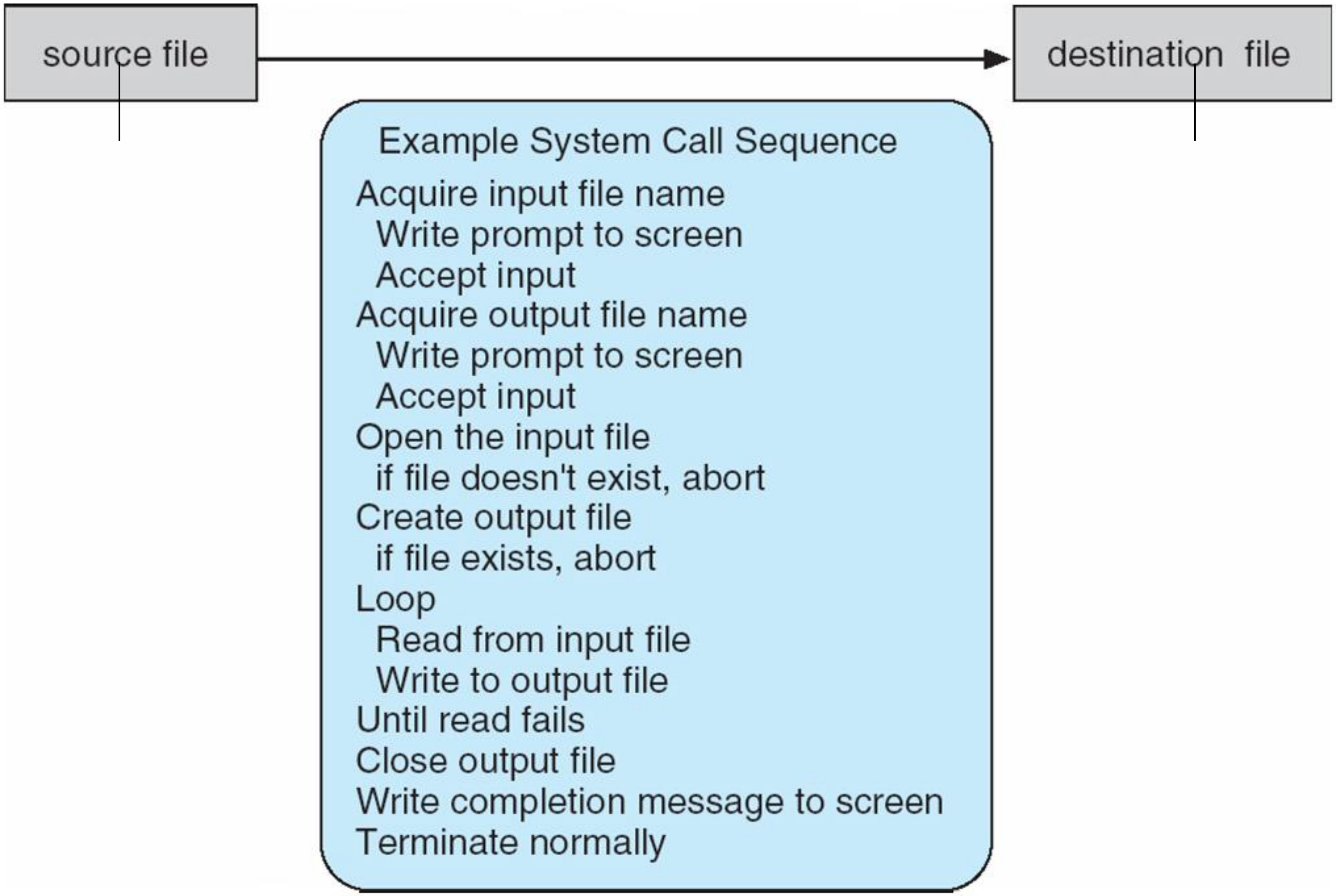
- Graphical interface for some email client: Might provide a user with a button that performs all steps for fetching and highlighting new emails
- User doesn’t need to know any detail or commands
- An API for file IO might provide a developer functions that copy a file from one location to another without requiring that the user understand details of file system permissions + operations
- Just dragging a file to a new directory can copy it!
- Much detail remains hidden behind the scenes
- E.g. no need to say “copy”, or to say “which source file” or which “destination”; just drag!

System Call & API



Example System Call

System call: copy contents of one file to another file



API and Function Specification

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:

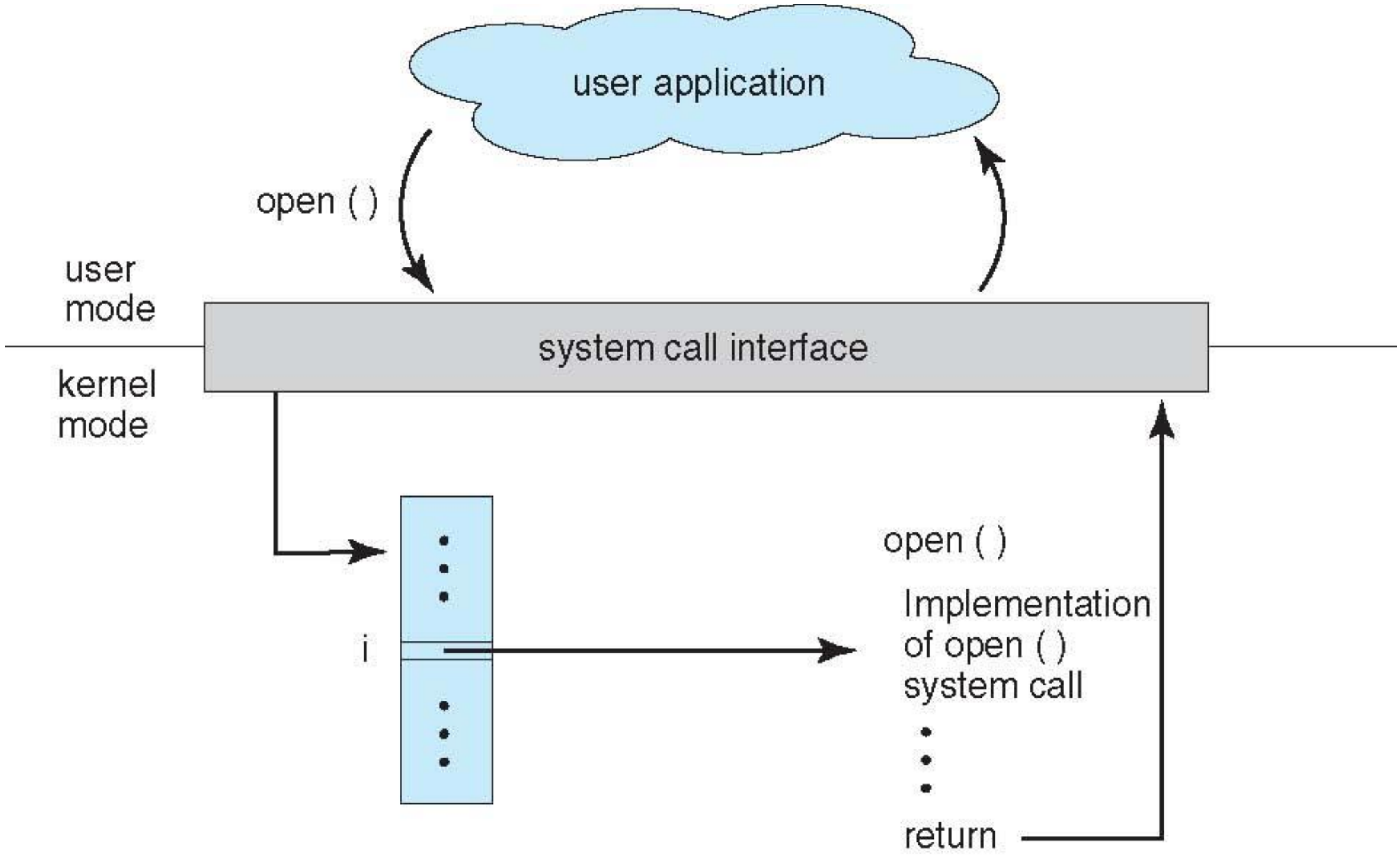
- `int fd`—the file descriptor to be read
- `void *buf`—a buffer where the data will be read into
- `size_t count`—the maximum number of bytes to be read into the buffer

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.

System Call Implementation

- A defined number is associated with each system call
  - System call interface maintains a table indexed according to these numbers
- System call interface invokes intended function in OS kernel; returns status and function return value
- Transparent to caller/user how system call is implemented
  - Just needs to obey API and understand what the OS will do as a result of such a call
  - Detail of OS interface remains hidden from programmer by API
  - Managed by run-time support library: a set of functions built into libraries

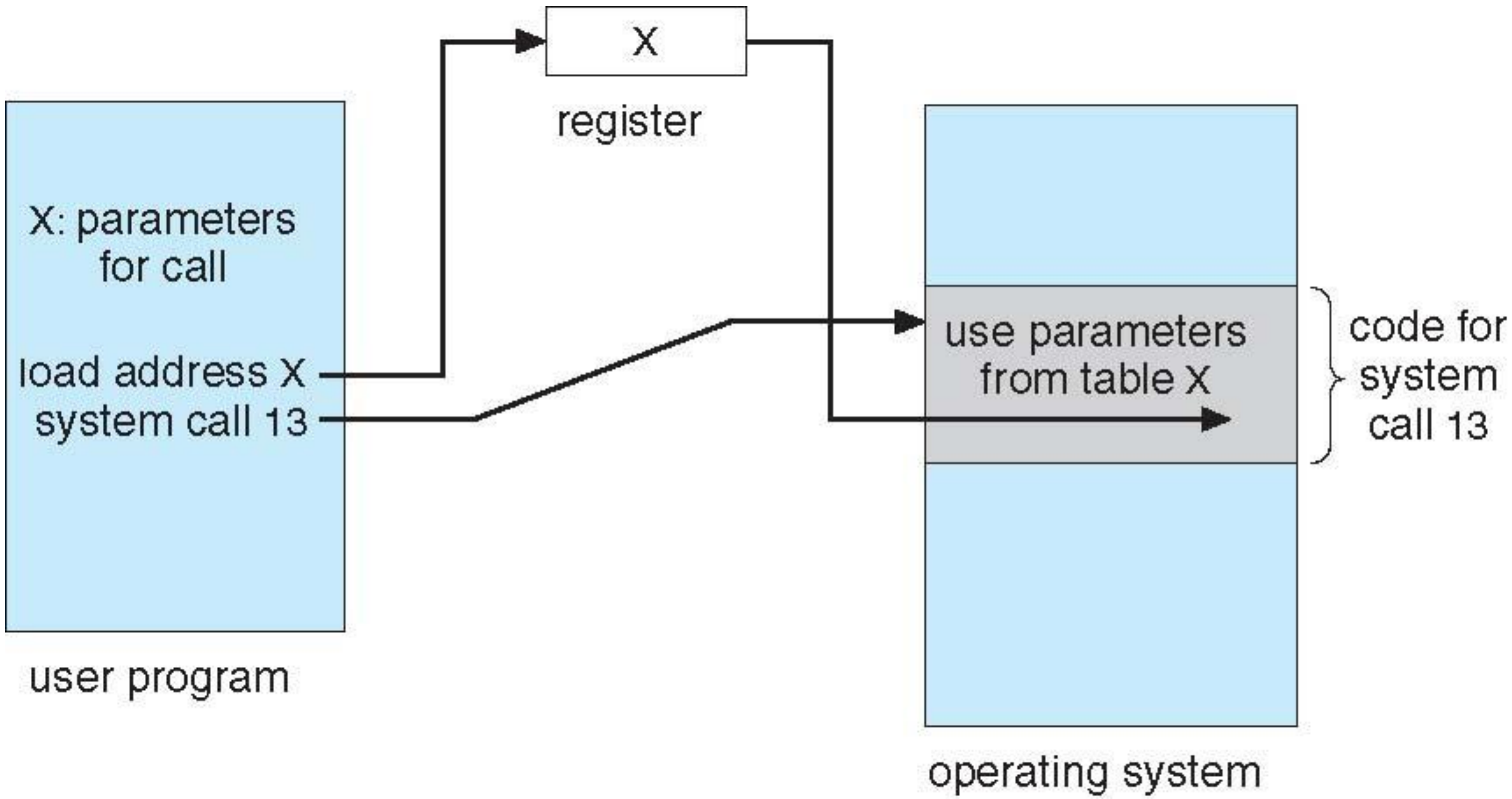
System Call & OS Relationship



System Call Parameter Passing

- Often, more information is required than simply the identity of specific system call; i.e. multiple parameters!
  - Type & amount of information vary according to OS and call
- General methods used to pass parameters to OS:
  - Simplest: pass parameters in registers
    - Fastest, hence most desirable, but limited in number, space!
    - There may be way more parameters than registers
  - B: Parameters stored in a block B, table in memory; then the address of B is passed as a parameter in a register
    - Is approach taken by Linux and Solaris
    - Notice implied indirection during execution, costs execution time
  - S: Parameters pushed onto stack S by program, popped off by operating system
  - Block B and stack S methods do not limit the number or length of parameters being passed

Parameter Passing via Table



System Calls -Process

Options and Tools for process control:

- create process, terminate process
- abort depending on “special” circumstances
- load, execute
- get process attributes, set process attributes
- wait for some defined time
- wait for an event, signal some event
- allocate and free memory
- Dump memory and key information (e.g. registers) if error
- Debugger for determining bugs, single step execution
- Locks for managing and synchronizing access to shared data between concurrent processes

System Calls -Files

- **Tools and steps for file management**
  - **create file, delete file**
  - **open, close file**
  - **read, write, reposition to defined file location or index**
  - **get, set file attributes and access restrictions**
- **Tools and steps for device management**
  - **request device, release device**
  - **read, write, reposition**
  - **get device attributes, set device attributes**
  - **logically attach further, or detach some of current devices**

System Calls -Communication

- Tools and steps for information maintenance
  - get time or date, set time or date
  - get system data, set system data
  - get and set process, file, or device attributes
- Tools and steps for communication
  - Create communicating connection, delete such 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

System Calls -Security

Tools and steps for locking and protection

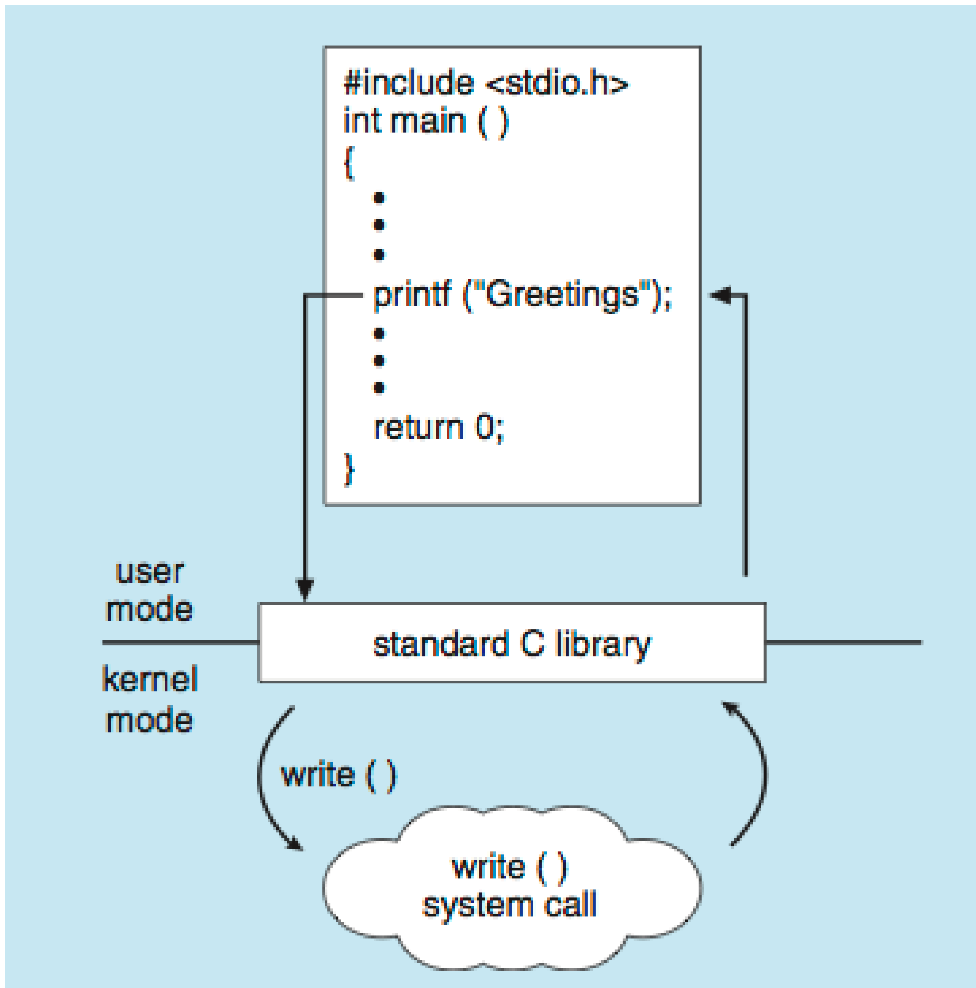
- **Control access to resources, lock or unlock access**
- **Get and set permissions, for OS "owned" resource!**
- **Allow and deny user access**

Windows & Unix 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()

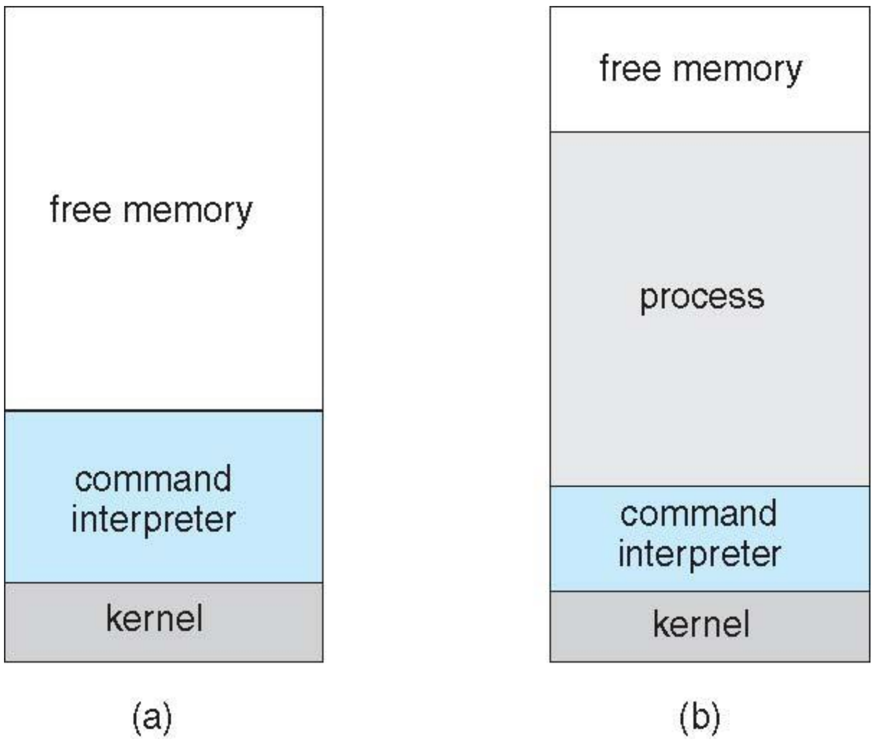
Standard C Library Example

Run-time system invokes printf(), which in turn invokes write() system call; see p. 23 example!



Example: MS-DOS

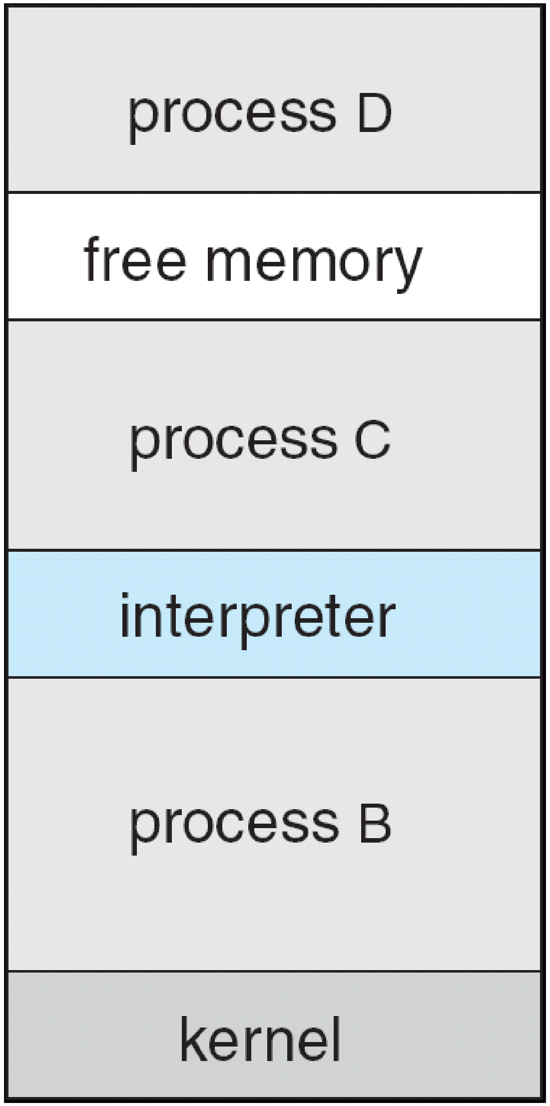
- Single-tasking OS
- Shell invoked when system booted
- Simple method to run
  - No process created
- Single memory space
- Loads program into memory, overwriting all but the kernel



At system startup running a program

Example: BSD

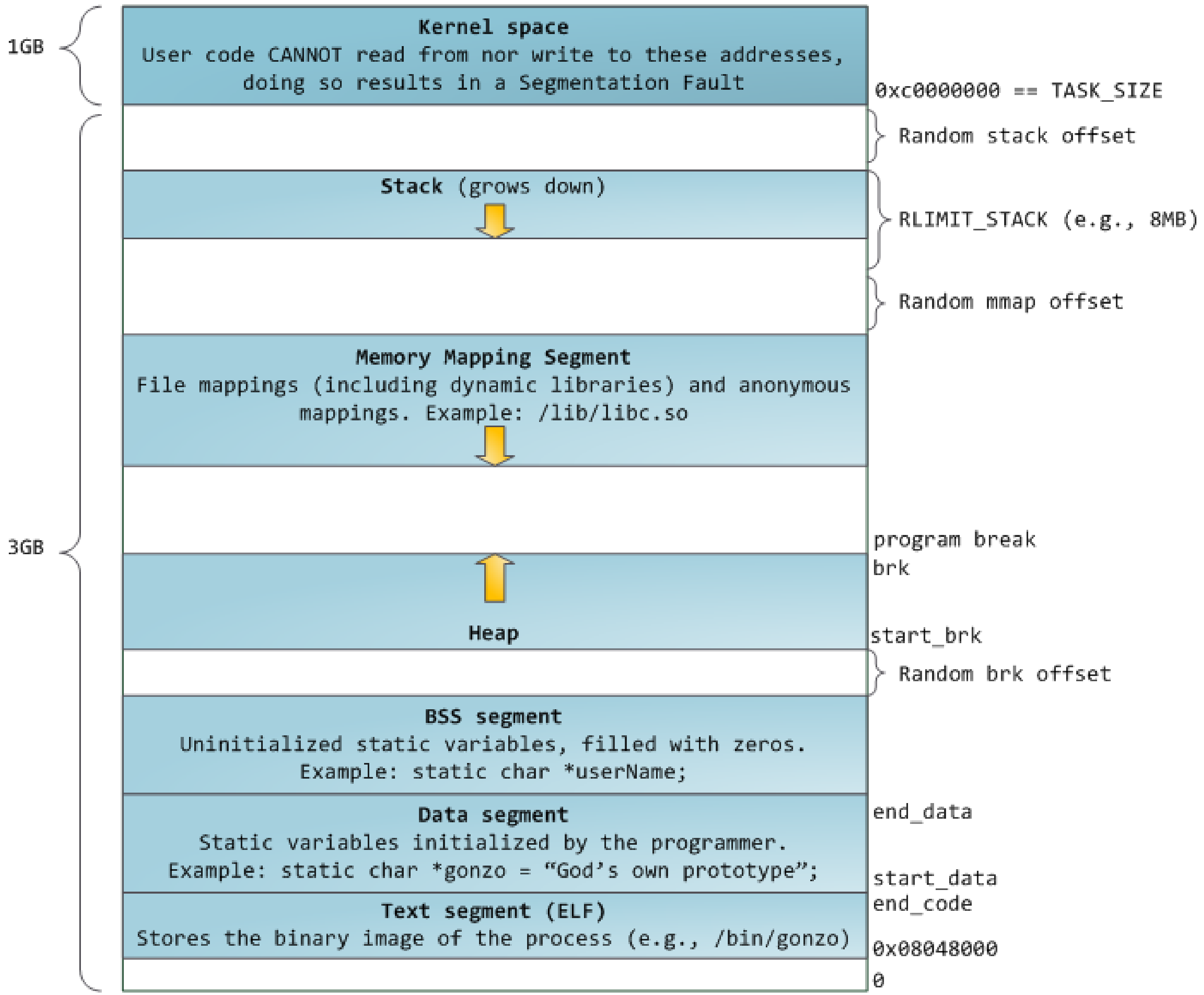
- Unix: Berkeley SW Distribution
- Multitasking OS
- User login -> invoke user's choice of shell
- Shell executes fork() system call to create process
  - Executes exec() to load program into process
  - Shell waits for process to terminate or continues with user commands
- Process exits with:
  - code = 0 – no error
  - code > 0 – that number is error code



Example: Linux

- Linux is a family of open source (Unix-like) OSes based on the Linux kernel
- First released September 17, 1991, by Finnish computer enthusiast Linus Torvalds
- Distribution includes kernel and supporting system SW and libraries, many provided by the GNU project
- Some distributions use "Linux" in their name, but the Free SW Foundation [12] uses the name GNU/Linux
- Popular Linux [11] distributions include Debian, Fedora, and Ubuntu
- Commercial distributions include Red Hat and SUSE
- Because Linux is freely redistributable, anyone may create a distribution for any purpose

Example: Linux Memory



System Programs vs. OS

OS & System Programs

Application Programs: -
MS Office, Banking System

System Program: -
Editors, Compiler and Command Interpreter

Operating System

Physical Devices/Computer hardware

OS & System Programs

- System programs provide a convenient environment for program development and execution; e.g.:
  - Programming language support: create source file, provide libs, linker, loader, debugger, file manager, etc.
  - Run application programs
  - File manipulation
  - Status information of executing programs
  - Program loading and execution
  - Performance measurement; e.g. “time my_prog” even in DOS
  - Communications, with other programs of same user; others
  - Batch services, running concurrently in background via &
- User view of OS formed by system program calls plus their responses

OS & System Programs

- **Provide a convenient environment for program development and execution**
  - **Some are simply user interfaces to system calls; others more complex**
- **File management - Create, delete, copy, rename, print, dump, list, manipulate files, directories, access**
- **Status information**
  - **Requests may ask system for info – current date, time; amount of available memory, disk space, number and UIDs of users**
  - **Or to get detailed performance info (performance history)**
  - **Typically, these system programs format data, output to stdout, some to stderr**
  - **Some systems implement a registry –a registry is used to store and retrieve configuration information**

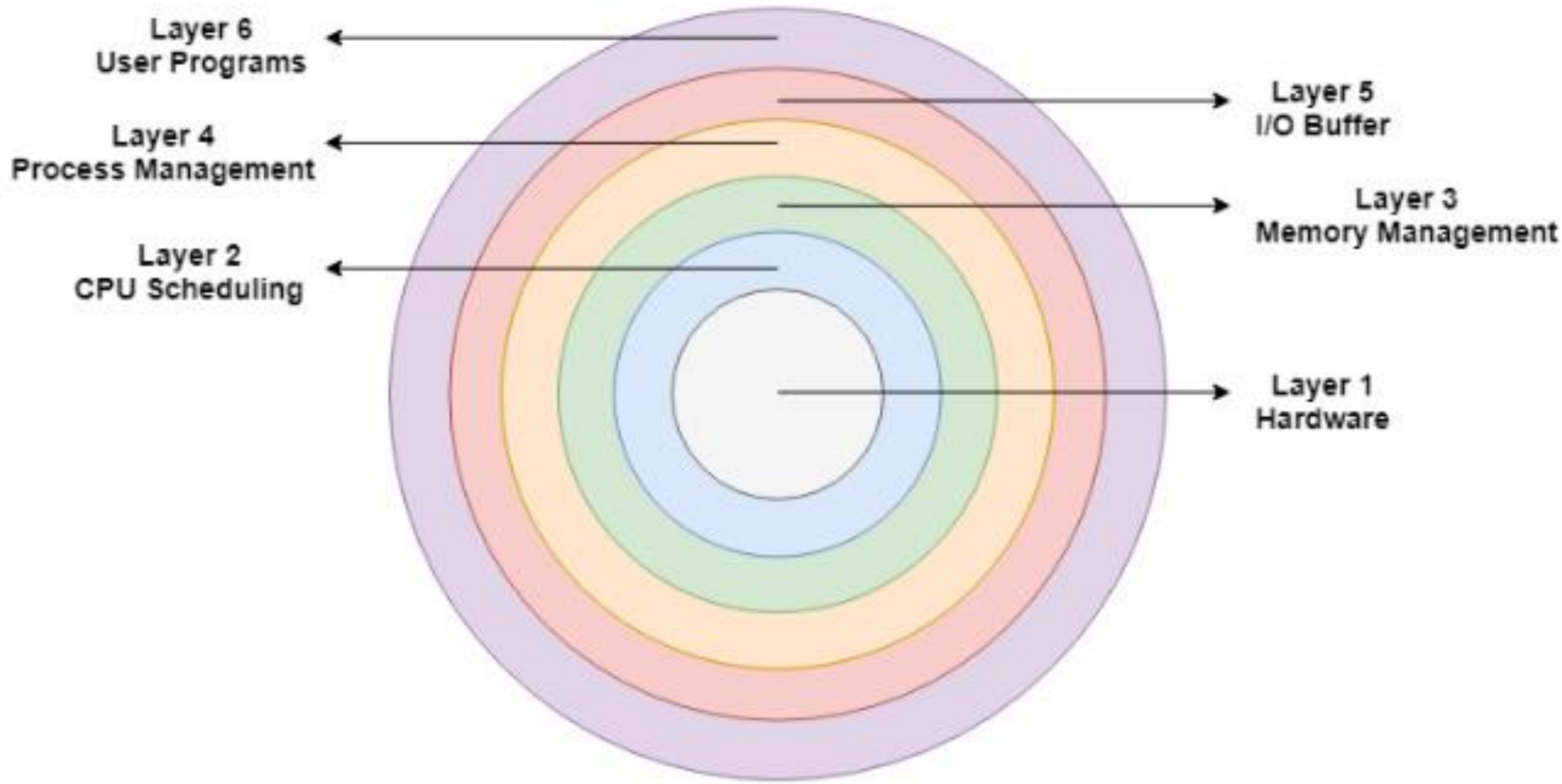
OS & System Programs

- File manipulation, includes:
  - Text editors to create, modify, delete, restrict files
  - Special commands to search contents of files or perform transformations of text
- Language support: Compilers, assemblers, debuggers and interpreters
- Program loading and execution: Absolute loaders, *relocatable* loaders, linkage editors, and overlay-loaders, debugging systems for high-level and machine language
- Communication: Provide 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, etc.

OS & System Programs

- **Background System Services**
  - **Some launched at boot time; regular services**
    - **Some only active at start, then terminate when complete**
    - **Others run from system boot to shutdown, in background**
  - **OS performs: disk checking, process scheduling, error logging, printing**
  - **Run in user context (AKA mode), not kernel context**
  - **Known as services, subsystems, daemons**
  - **Initiate background even explicitly via &** (in Unix, Linux)
- **Application programs receive OS services**
  - **Run is initiated by users; resources are granted by OS**
  - **Apps not part of OS, but may be running under OS control**
  - **Launched by command line, mouse click, finger poke, etc.**

OS Design and Implementation



LAYERED OPERATING SYSTEM

See also "Layered Approach" p. 51

OS Design and Implementation

- **Design and Implementation of OS not perfected yet ☺ but some approaches have proven quite successful**
- **Internal structures of different OSes vary widely**
- **In each OS design, specify a priori: targeted functions, performance goals, user convenience, etc.**
- **Affected by choice of HW, type of HW & SW components, state of the art of SW Engineering**
- **User goals and System goals**
  - **User goals – OS should be convenient to use, easy to learn, reliable, safe, fast, i.e. low overhead (time left for user SW)**
  - **System goals – OS should be cleanly designed, easy to implement and maintain, and flexible, reliable, error-free, efficient, also safe!**

OS Design and Implementation

- **Important principles to separate during creation and productizing of an OS:**

  OS Policy: ***What*** **will be done?**
  OS Mechanism: ***How*** **to do it?**

- **Mechanisms determine how, policies decide what**
- **Separation of policy from mechanism grants flexibility if policy decisions change later**
- **Yes, OS will evolve over time!**
- **Specifying and designing an OS is highly creative task of software engineering (SWE)**
  - **Great fun, and a grand challenge**
  - **Also: Maintaining, improving, correcting**

Implementation

- **Great variation in actual implementation over time:**
  - **Early OSes implemented in assembly language!**
  - **Then in higher-level languages like Jovial, PL/1, . . .**
  - **Now commonly in C++**
- **Or even a mix of languages: compromise!**
  - **Lowest levels in assembly; value: efficiency, speed**
  - **Main body in C++; value: ease of implementation, portability**
  - **In practice, many older system programs are coded in C, C++, scripting languages like PERL, Python, shell scripts**
- **HLL easier to port to other hardware**
  - **But usually slower than low-level (asm) language**
- **Emulation allows OS to run in non-native environment**
  - **Typically slow execution**
  - **Ease of porting to another host**

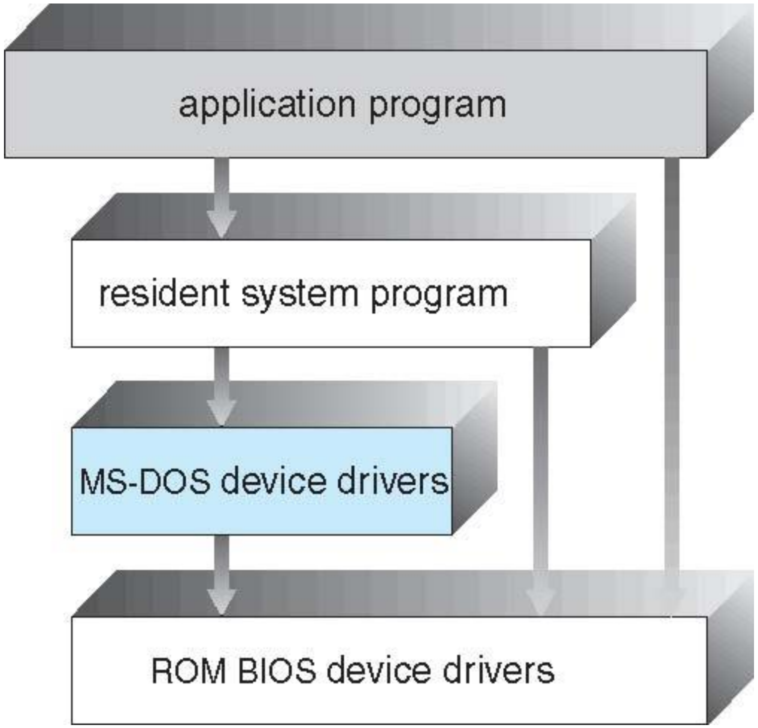
Operating System Structure

- General-purpose OS forms a large body of SW
- Various SWE ways to structure and modularize during OS design and build:
  - Simple structure: MS-DOS
  - More complex: UNIX
  - Layered: an abstraction to next + - level
  - Microkernel: e.g. Mach

Simple Structure MS-DOS

MS-DOS designed to maximize functionality in minimum space

- **Not divided into distinct modules**
- **Although MS-DOS has some SWE structure, its interfaces and levels of functionality are not well separated**



Not So Simple Structure UNIX

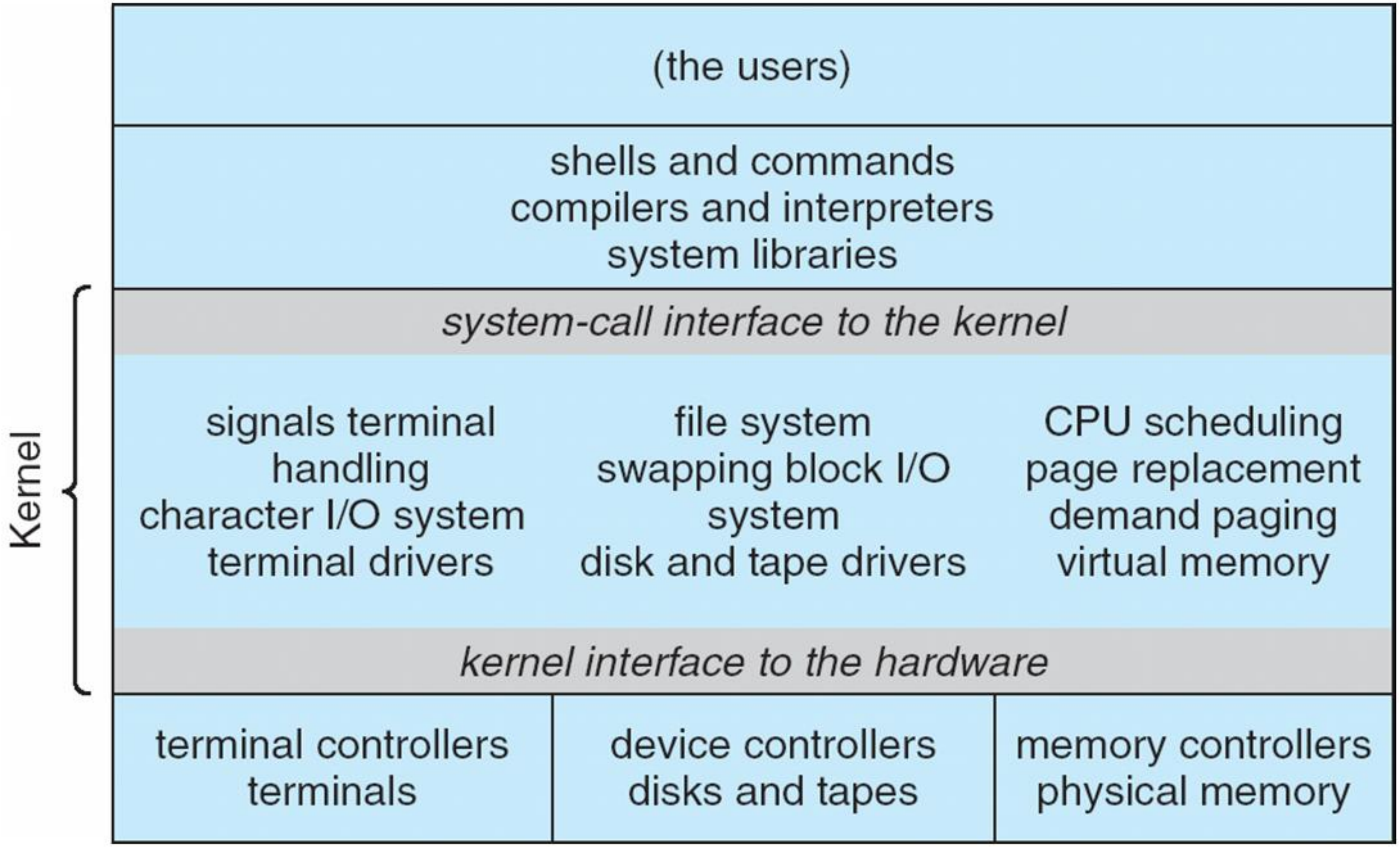
- Original UNIX operating system had limited structuring (in the sense of modular SWE)
- Structure evolved, following SWE principles over time, but as an afterthought
- UNIX OS consists of two separable parts:
  - 1. System programs
  - 2. Kernel consists of everything below the system program interface and above the physical HW
  - Unix OS provides file system, CPU scheduling, load-balancing, memory management, other OS functions
  - Large number of diverse functions

UNIX Kernel

- **A Unix kernel —AKA the core, key component of the operating system— consists of many kernel submodules like:**
  - **Process manager, scheduler, file manager, device manager, network manager, memory manager**
- **Handles interrupts originating from hardware devices**

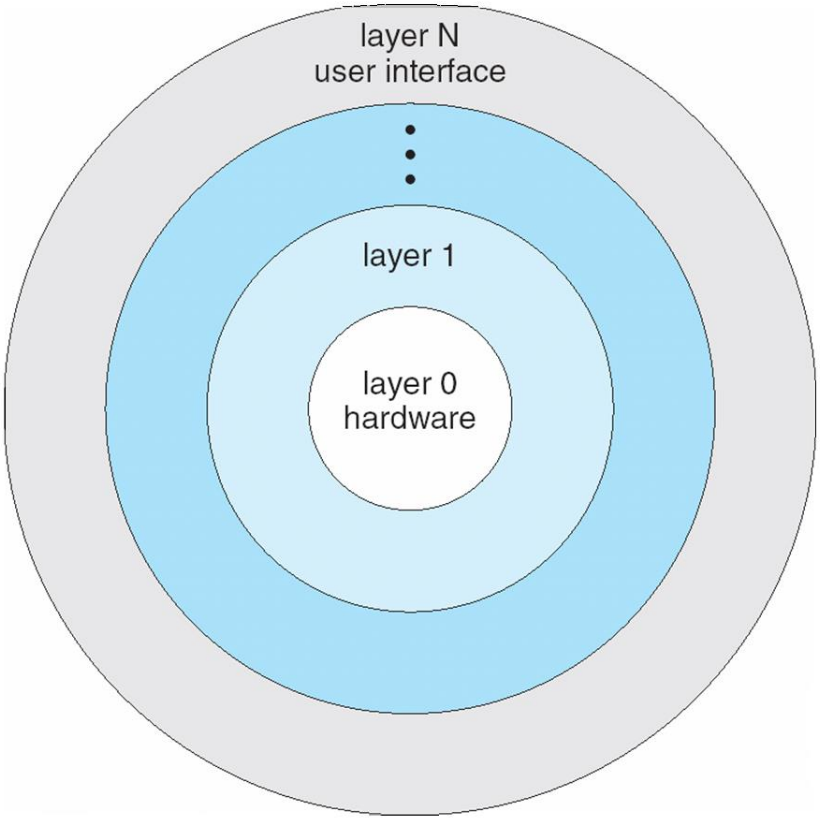
UNIX System Structure

Way beyond simple, now close to *perfectly* layered



Layered Approach

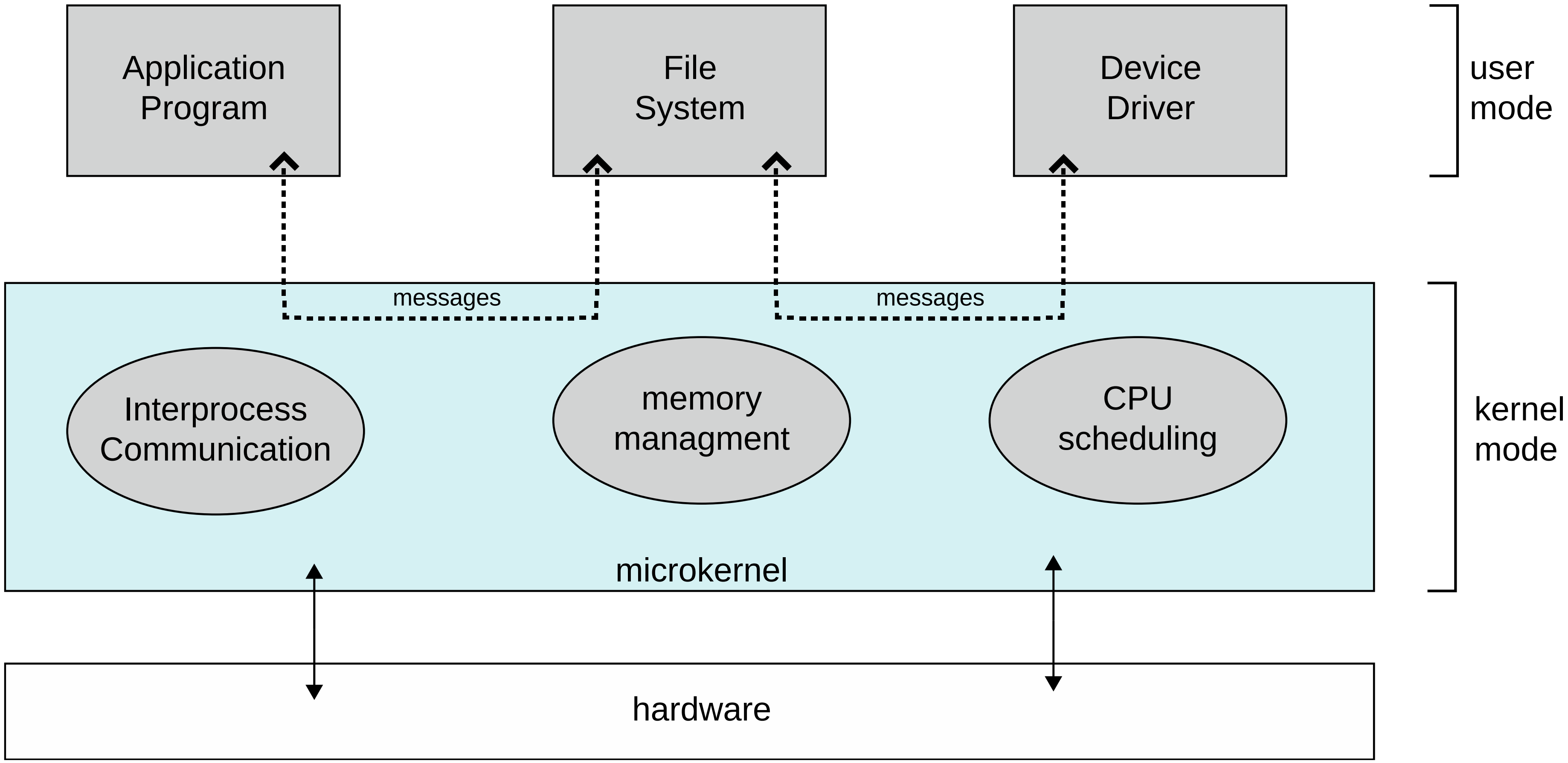
- **Ideal OS divided into number of N layers (levels), each built on top of lower layers**
- **Bottom layer 0 being HW**
- **Highest layer is UI: user interface; user on one side**
- **With modularity, layers are selected such that each layer uses functions and services of only lower-level layers**
- **See Layered Structure shown earlier p. 42; here just abstract "layer" 0 . . N**



Microkernel System Structure

- **Move as much as feasible from kernel to user space, or to system program level**
- **Mach example of microkernel:**
  - **Mac OS X kernel (Darwin) partly based on Mach**
  - **Mach OS detail: see [6]**
  - **OS provides user communication via message passing**
- **Benefits of small microkernel:**
  - **Easy to port OS to a new architectures**
  - **Reliable, less code actively running in kernel mode**
  - **Security, easier to ensure solely allowed access with small kernel**
- **Cost of Message Passing:**
  - **Performance overhead due to extra communication between user-space and kernel-space**
  - **So a plausible design goal is: To reduce the amount of communication and minimize the cost per message**

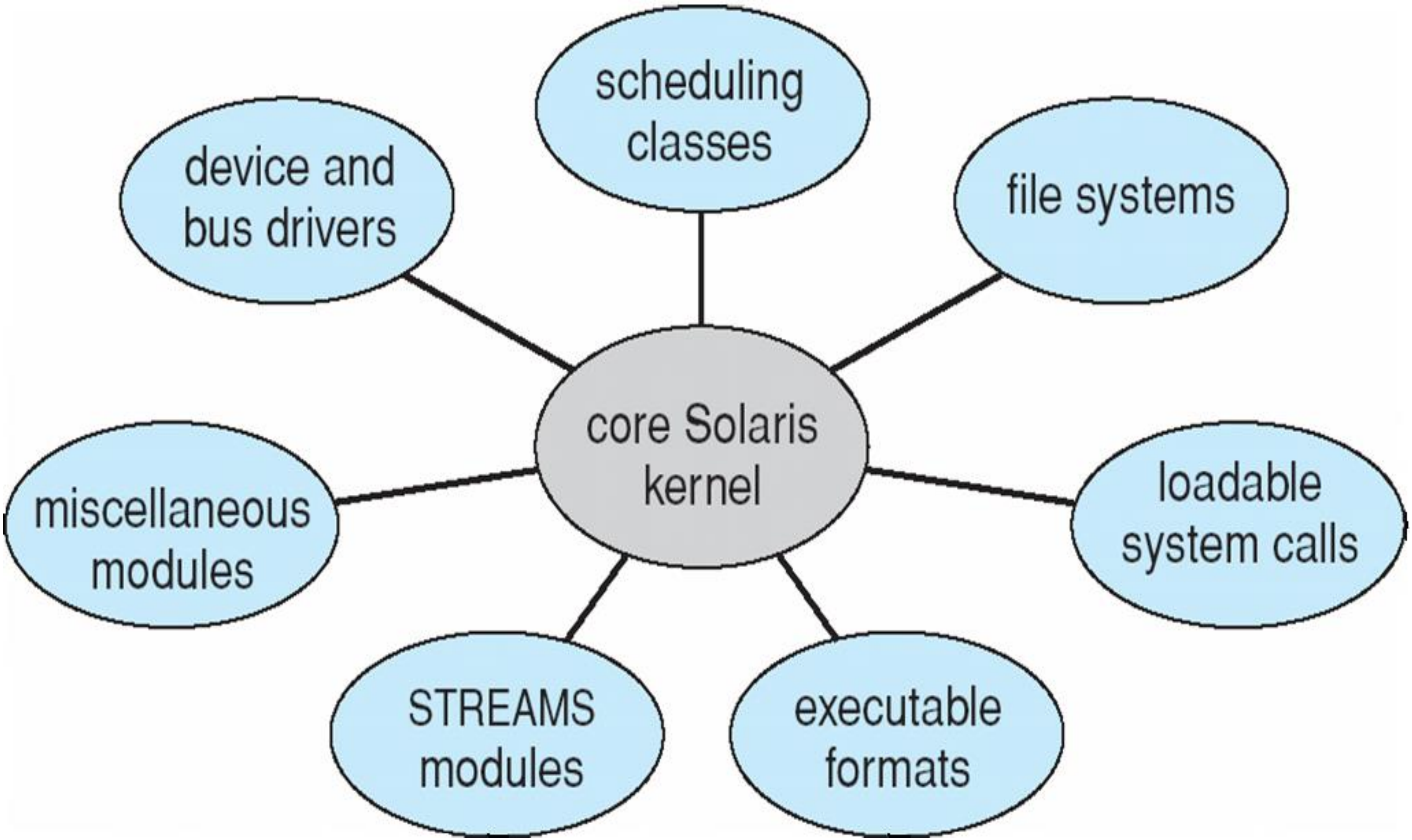
Microkernel System Structure



OS Modules

- Many modern operating systems implement and provide to user: loadable kernel modules!
  - Grants great flexibility in OS configuration
  - Each crucial component being a separate, explicit option
  - Each talks to the others over defined interfaces
  - Each is loadable as needed within the kernel
  - OS runs fine with some kernel modules not loaded!
  - Modules not loaded clearly cannot provide services! ☺
- Similar to layers but with added flexibility; samples:
  - Linux
  - Solaris
  - Both very fine *OS specimens* ☺

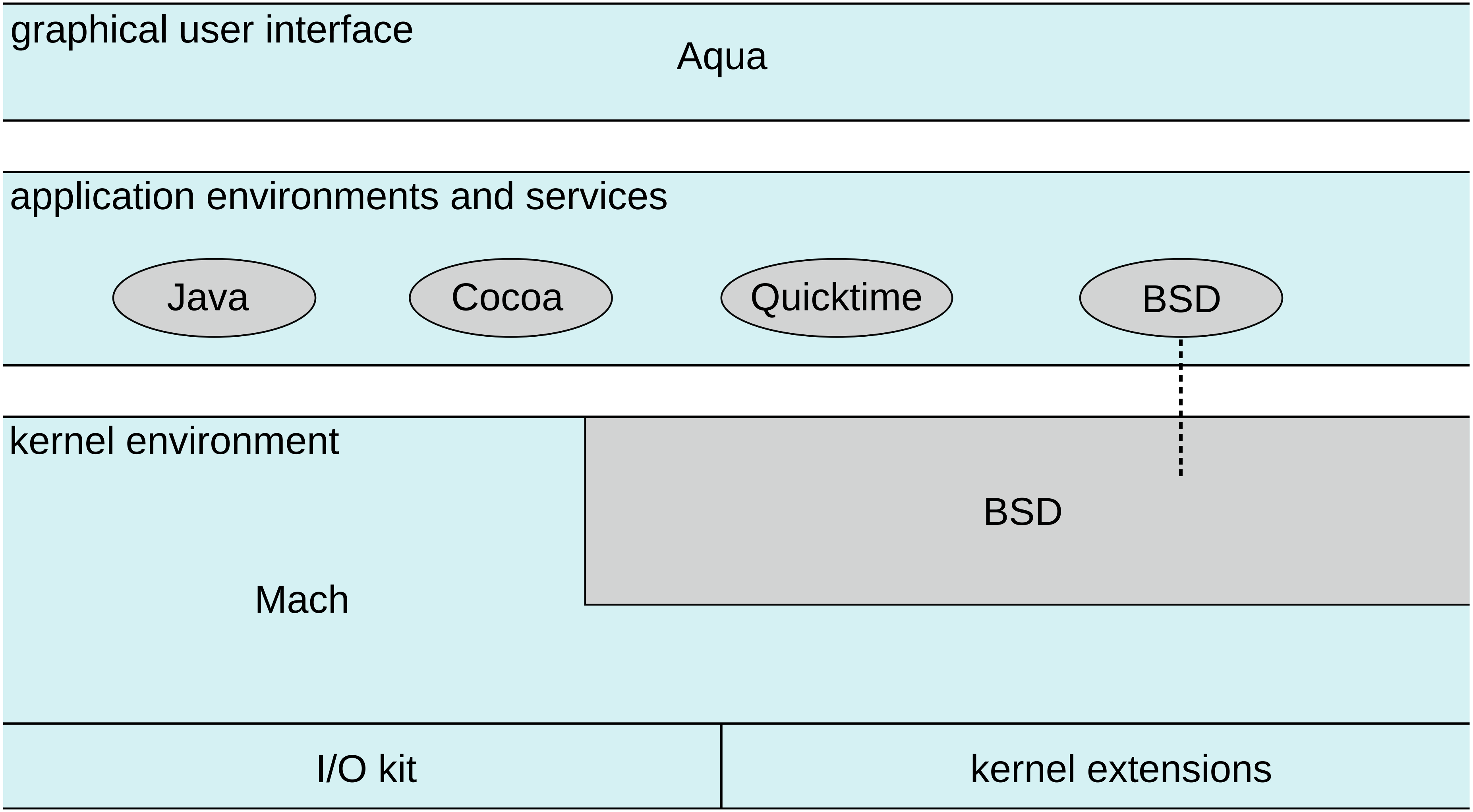
Solaris Modular Approach



Hybrid Systems

- **Modern OSes are frequently not pure, single models; some are hybrid models!**
  - **Hybrid SW solution combines multiple OS design approaches to address: performance, security, usability needs**
  - **Linux and Solaris kernels execute in *kernel address space*, monolithic, sacred ☺; yet modular for dynamic, flexible loading of added functionality**
  - **Windows mostly monolithic, plus configures different microkernels for different subsystem *personalities***
- **Apple Mac OS X hybrid, layered, Aqua UI plus Cocoa programming environment; see [4] Aqua, Cocoa [9]**
  - **Cocoa: see later**
  - **Next: kernel = Mach microkernel + BSD Unix + I/O kit and dynamically loadable modules (called kernel extensions)**

Mac OS X Structure



Cocoa

- Cocoa is Apple's native OO API for its desktop operating system MacOS
- Cocoa consists of the Foundation Kit, Application Kit, and Core Data frameworks
- For end-users, Cocoa applications are those written using the Cocoa programming environment; actual programming language
- Won't cover Cocoa or language here; for detail see [9]

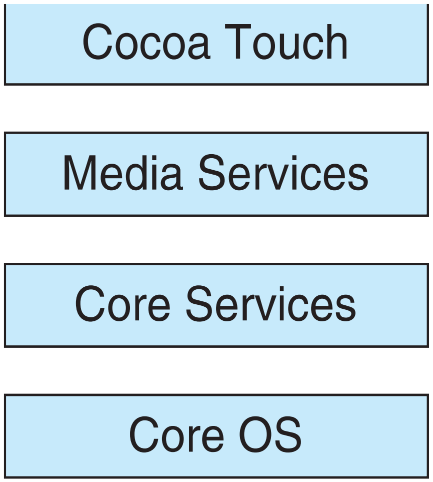
iOS

- **IOS is a *mobile OS* created, developed by Apple Inc. to run on Apple systems**
- **IOS powers many of the company's mobile devices, including the iPhone, and iPod Touch**
- **It also powered the iPad prior to the introduction of iPad OS in 2019**
- **For detail see [8]**

iOS

Apple mobile OS for *iPhone*, *iPad*

- Structured on Mac OS X, added functionality
- Does not run OS X applications natively
  - Also runs on different CPU architectures (ARM, Intel); Intel found ARM hard=good competitor!
- Cocoa Touch: Objective-C API to develop apps
- Media services layer for graphics, audio, video
- Core services provides cloud computing, databases; cloud = delivery of computing services over the Internet
- Core OS, based on Mac OS X kernel
- Student question: Why so many languages: C, C++, C#, Cocoa, etc.? Pro and Con of multiple languages?



Android OS

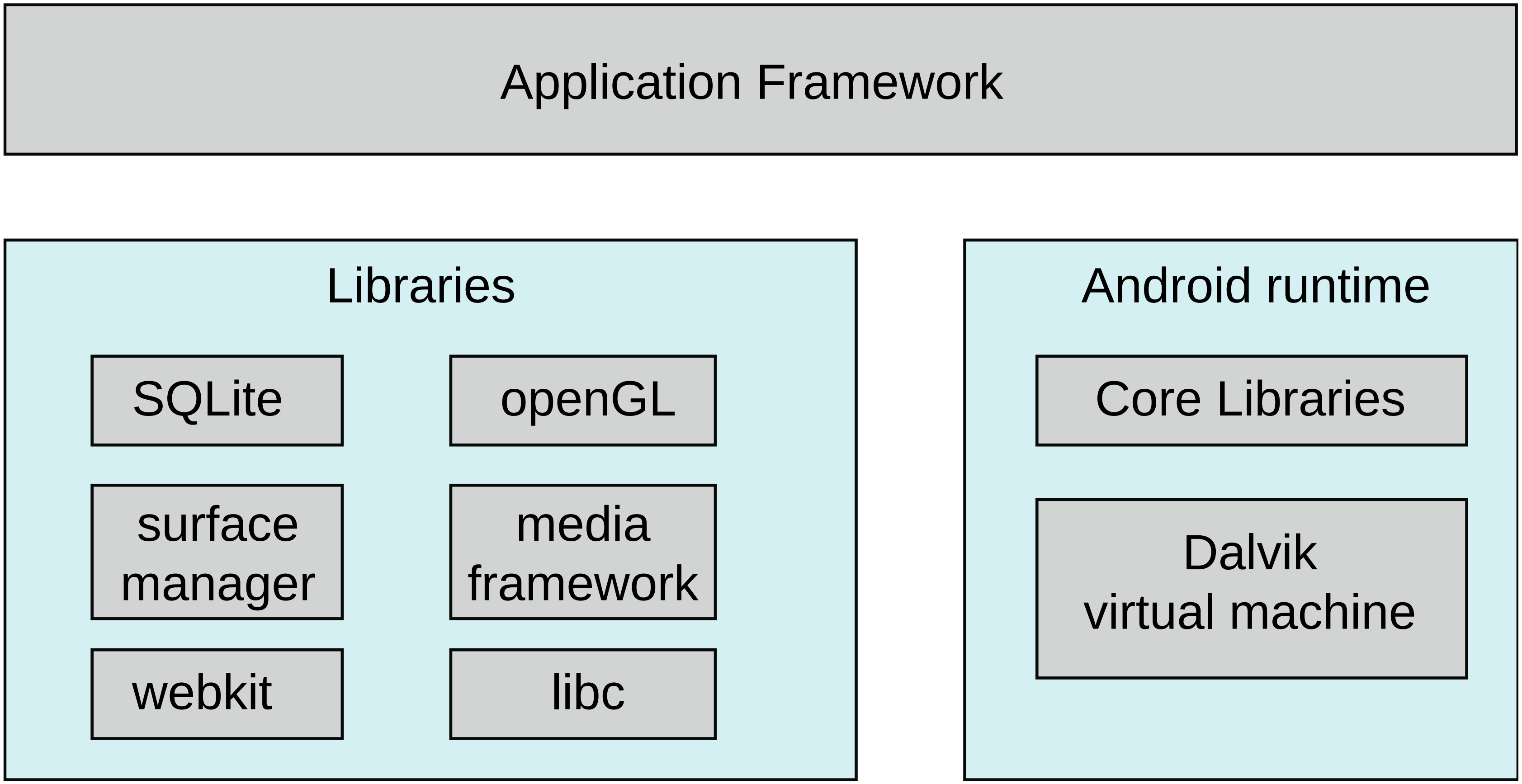
Android

- **Android is a mobile operating system based on a modified version of the Linux kernel and other open source software**
- **Designed primarily for touchscreen mobile devices such as smartphones and tablets**
- **Commercially sponsored by Google**
- **Unveiled 2007**
- **First commercial Android devices launched 2008**

Android

- Developed by Open Handset Alliance
  - Open Source
- Similar stack to IOS
- Based on Linux kernel with heavy mods
  - Provides process, memory, device-driver management
  - Adds power management; critical for and handheld device!
- Runtime environment includes core set of libraries and Dalvik virtual machine (now discontinued)
  - Apps developed in Java, plus Android API
    - Java class files compiled to Java bytecode then translated to executable that runs in Dalvik VM
    - Complex, slow!
- Libraries include frameworks for web browser (webkit), database (SQLite), multimedia use

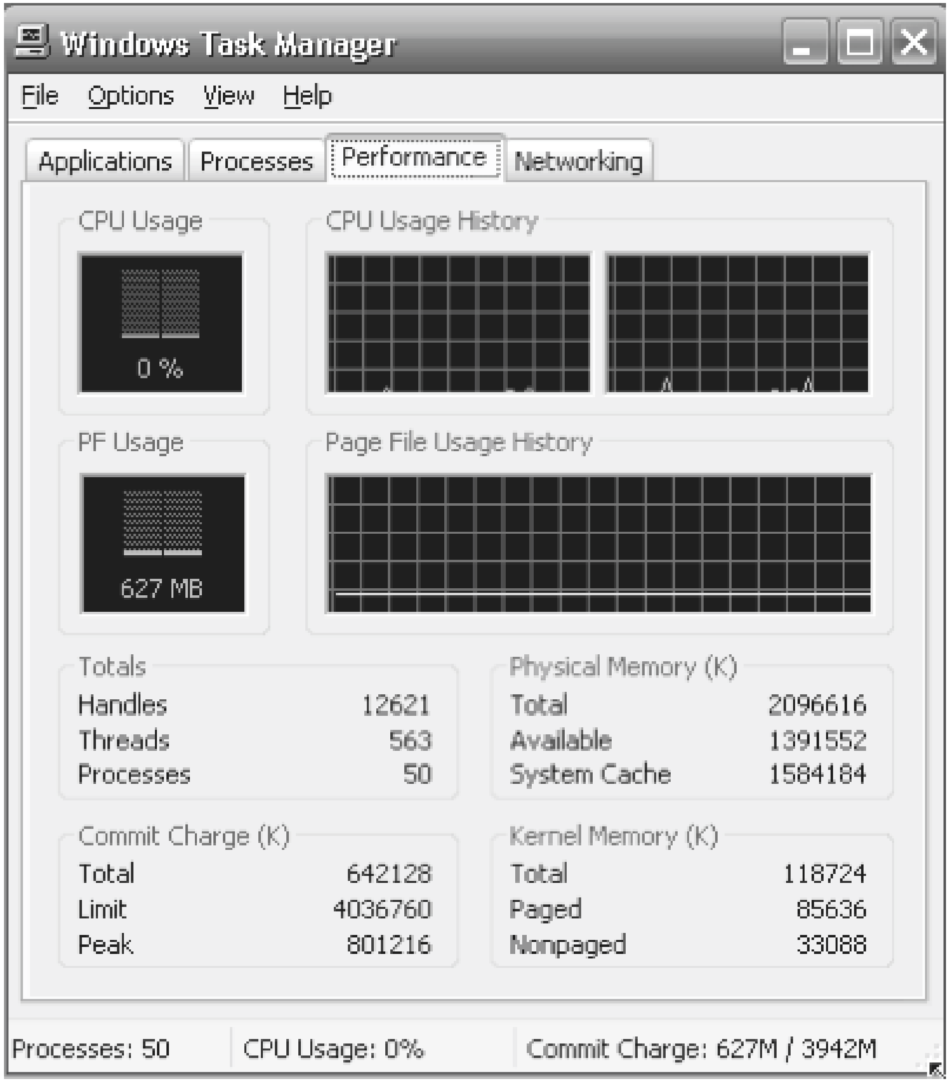
Android HL Architecture



OS Debug & Tuning

Performance Tuning

- **Improve performance by identifying, then removing bottlenecks**
- **OS must provide means of measuring and displaying information about system behavior**
- **Measured data then help tune, optimize, eliminate resource bottlenecks**



OS Debugging

- **Debugging:** finding and fixing errors, AKA **bugs**
- **OS generates log files containing error information**
- **Aborting user SW can generate core dump: a file, capturing memory of process at the time of *disaster***
- **OS failure can generate crash dump file, containing kernel memory! Different from core dump!**
- **Performance tuning, an art and a science, intended to optimize system performance**
  - **Sometimes using *trace listings* of activities; analyze! Hard!**
  - **Profiling is periodic sampling of pc to look for statistical trends**

Kernighan's Law: "Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it." ☺

DTrace of Solaris

- DTrace tool in Solaris (now Oracle), FreeBSD
- Probes send status when code is executed within a provider, capturing state info and sending it to users of those probes
- Example of following XEventsQueued() system call
- Show sample via:

  #ifdef DEBUG . . .

```
# ./all.d 'pgrep xclock' XEventsQueued
dtrace: script './all.d' matched 52377 probes
CPU FUNCTION
  0 -> XEventsQueued                      U
  0   -> _XEventsQueued                   U
  0     -> _X11TransBytesReadable         U
  0     <- _X11TransBytesReadable         U
  0     -> _X11TransSocketBytesReadable U
  0     <- _X11TransSocketBytesreadable U
  0     -> ioctl                          U
  0       -> ioctl                        K
  0         -> getf                       K
  0           -> set_active_fd            K
  0           <- set_active_fd            K
  0         <- getf                       K
  0         -> get_udatamodel             K
  0         <- get_udatamodel             K
...
  0         -> releasef                   K
  0           -> clear_active_fd          K
  0           <- clear_active_fd          K
  0           -> cv_broadcast             K
  0           <- cv_broadcast             K
  0         <- releasef                   K
  0       <- ioctl                        K
  0     <- ioctl                          U
  0   <- _XEventsQueued                   U
  0 <- XEventsQueued                      U
```

Tracing Calls

```
#ifdef DEBUG

int indent = 0;                 // track call-depth 1..n

void trace_in( char * msg )     // at each call
{ // trace_in
    int blanks;
    if( want_cr_trace ) {       // defined outside
        ++indent;
        for( blanks = 0; blanks < indent; blanks++ ) {
            putchar( ' ' );
        } //end for
        printf( "-> %s\n", msg );
        fflush( stdout );       // force output!!
    } //end if
} //end trace_in
```

Tracing Calls

```
void trace_out( char * msg )  // at each “return”
{ // trace_out                // before each return
    int blanks;               // also implicit return }
    if( want_cr_trace ) {     // defined outside
        for( blanks = 0; blanks < indent; blanks++ ) {
            putchar( ‘ ‘ );
        } //end for
        printf( "<- %s\n", msg );
        fflush( stdout );     // force output!!
        --indent;
    } //end if
} //end trace_out

#else // DEBUG not defined!
#     define trace_in( msg )  // empty body!
#     define trace_out( msg ) // no code if !DEBUG
#endif DEBUG
```

Generating an OS

- An OS is ideally designed to run on **several, diverse target** machines
- For multiple run-time targets multiple OSes will be needed
- One can write a distinct OS for each target machine; much work ☹
- Or derive multiple OSes from a single source ☺
  - Saves OS development and increases # of executing hosts!
  - Supports ***user familiarity*** by “standardization”
  - Is slightly harder to code, but saves enormous design and re-implementation costs; long-term = building multiple OSes

Generating an OS

- **Operating System to be configured separately for each specific target system; ideally from a single source**
- **SYSGEN program receives info concerning specific target HW to configure a correspondingly specific OS**
  - **Used to build a separate, system-specific kernel**
  - **Can be more efficient code –custom made– than one single, general kernel for all different target systems**
  - **Yet all other (non-kernel) OS source code is often generated from a single base**
  - **Sounds straight-forward, yet is highly delicate SWE effort**

System Boot

- When powering on (AKA *booting*) the system, OS execution starts with special device at fixed location
  - First: firmware ROM holds initial boot code, to load part of OS
  - Similar to C or C++ program, starting at a defined point of main(), the OS begins at program label `_start:`
  - Code in crt0.o provides that C or C++ label `_start:`
  - Other OSes of course use different, but similar convention
- To get OS started, execute boot procedure:
  - Small piece of code – bootstrap loader, stored in ROM or EEPROM locates the kernel, loads it into memory, and starts

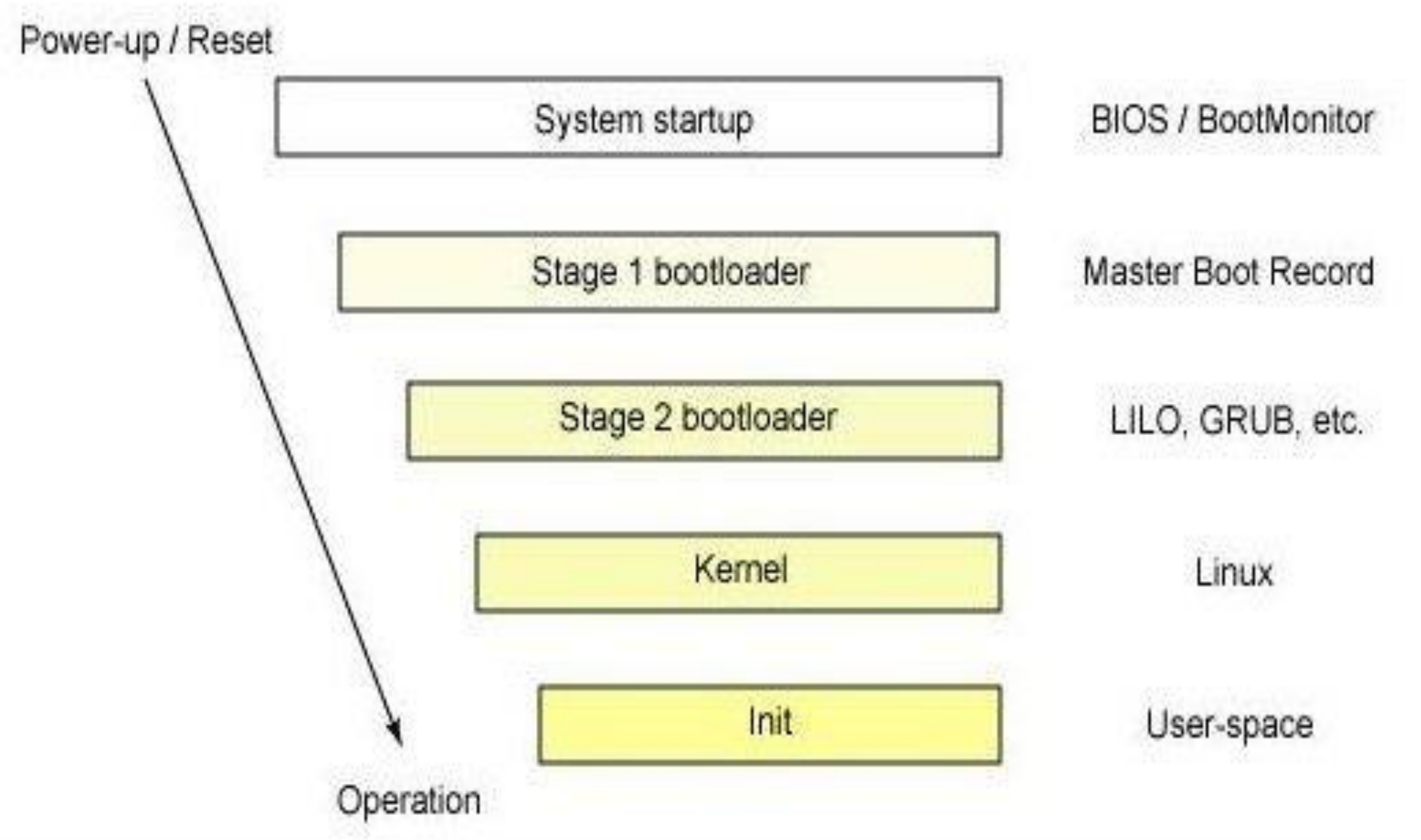
System Boot

- Common Linux bootstrap loader, GRUB, allows selection of kernel from various disks, versions, kernel options
- Famous GNU GRUB (GRand Unified Bootloader) is a boot loader package from the GNU Project
- GRUB is the reference implementation of the Free Software Foundation's Multiboot Specification, which provides user the choice to boot one of multiple OSes installed on a computer
- Or select a specific kernel configuration available on some particular OS'es partitions; key requirement, the combinations must work!
- That is SWE to the max, applied to OS build! ☺

System Boot

- **The kernel loads the rest of the OS, then system starts running**
- **Next page: The GRUB** (**GR**and **U**nified **B**ootloader) is a **bootloader** available from the GNU OS project

Linux System Boot Steps



Summary

- Introduced high-level functions for an OS
- Characterized resources managed by OS
- Mentioned accounting: which process and which user use how many resources, how long, at what cost
- Characterized protection that processes need from other, possibly competing processes
- Introduced system calls, which are allowed requests for help from OS, demanded by user program

Bibliography

1. Unix file system: https://en.wikipedia.org/wiki/Unix_filesystem
2. Unix history: https://en.wikipedia.org/wiki/Unix
3. Rutgers University File Protection: https://www.cs.rutgers.edu/~watrous/unix-protections.html
4. Aqua user interface: https://en.wikipedia.org/wiki/Aqua_(user_interface)
5. CDE: https://en.wikipedia.org/wiki/Common_Desktop_Environment
6. Mach OS: https://en.wikipedia.org/wiki/Mach_(kernel)
7. Mouse invention: https://www.computerhope.com/issues/ch001083.htm

Bibliography

8. Apple IOS: https://en.wikipedia.org/wiki/IOS
9. Cocoa detail: https://en.wikipedia.org/wiki/Cocoa_(API)
10. Windows: https://docs.microsoft.com/en-us/windows/win32/apiindex/windows-api-list
11. Fedora: https://en.wikipedia.org/wiki/Fedora_(operating_system)
12. Free Software Foundation: https://en.wikipedia.org/wiki/Free_Software_Foundation
13. Unix Kernel: https://en.wikipedia.org/wiki/Unix_architecture