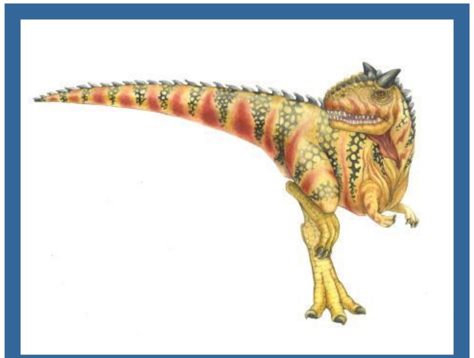
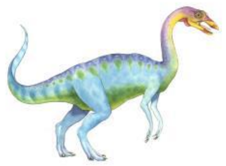
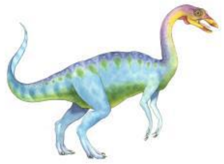
3. System Calls and OS Architecture





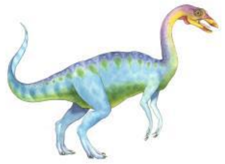
3. System Calls and OS Architecture

- System Calls
  - Interface between user program and OS
- OS architecture
  - Organization and Design of OS code



Objectives

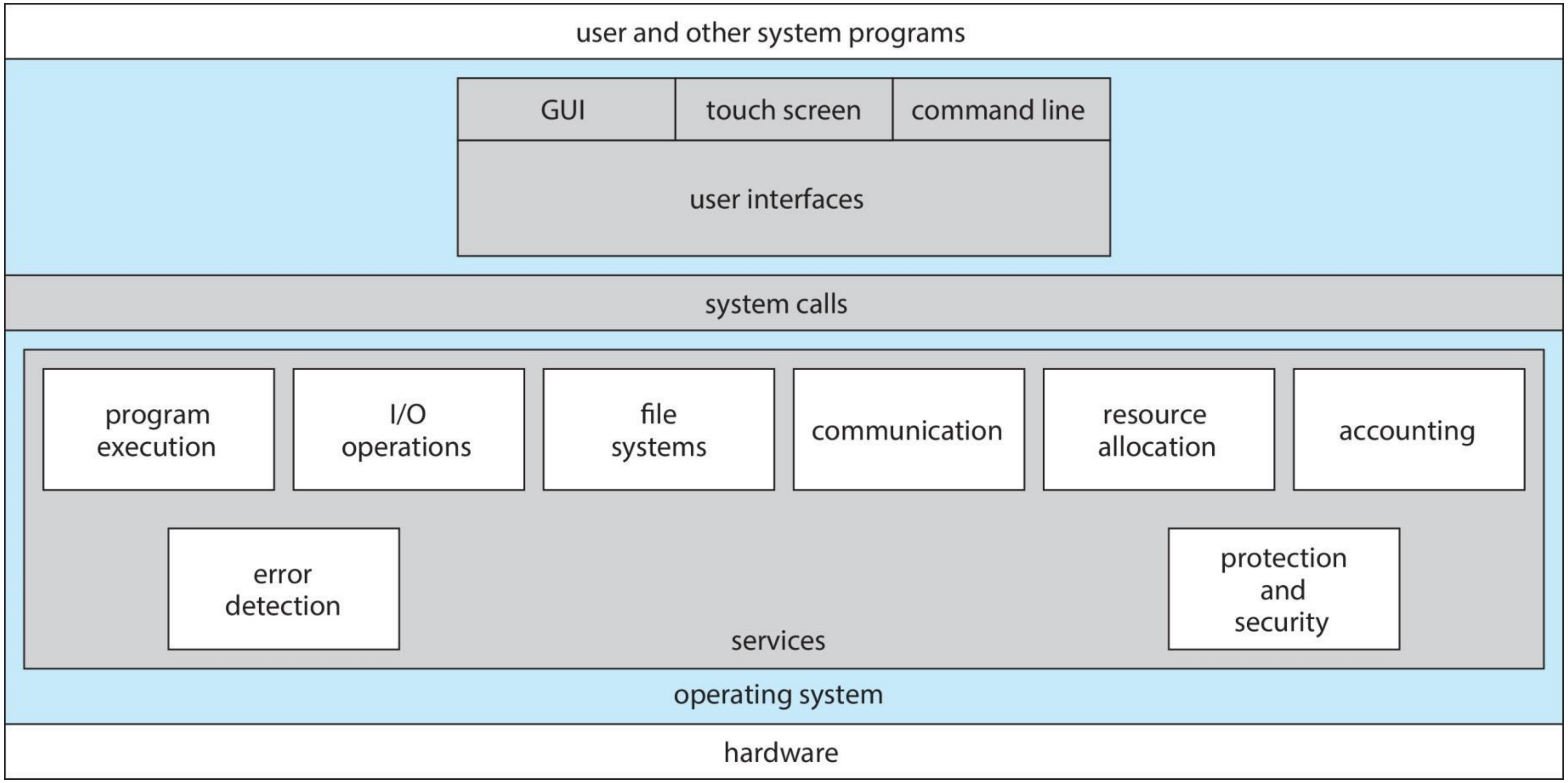
- **Identify services** provided by an operating system
- Illustrate **how system calls are used** to provide operating system services
- Compare and contrast **monolithic, microkernel, and hybrid** strategies for designing operating system

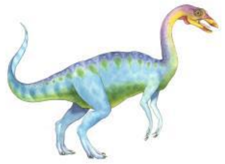


Operating System Services

- Process Management
  - A process a running program.
  - Starting, terminating processes
  - Coordinating processes (allowing processes to talk to one another, protecting them from one another)
- Memory Management
  - Allocation of memory for the processes
  - Protect memory from being written by another process
  - How to handle when we run out of RAM
- I/O Management
  - Device drivers
  - Providing API for the user programs (encapsulation principle)
- Storage Management
  - File systems and data

A View of Operating System Services





System Calls

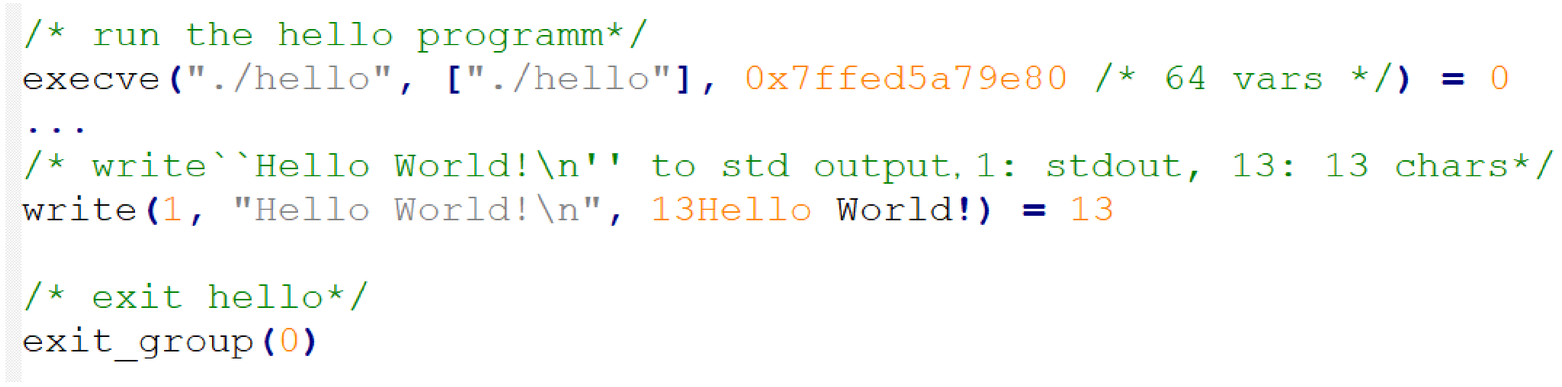
- Programming interface to the services **provided by the kernel**
  - Examples: write(…), read(…) on Unix/Linux
- Typically written in a high-level language (C or C++)
- Mostly accessed by programs via a high-level **Application Programming Interface** **(API)** rather than direct system call use
  - Examples: printf(…), scanf(…) in C
- Three most common APIs:
  - Win32 API for Windows
  - POSIX[1] API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X)
  - Java API for the Java virtual machine (JVM)

Standard C library vs C POSIX library
subset superset

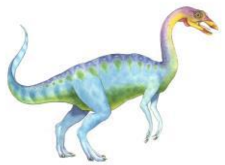
[1]The **Portable Operating System Interface** (**POSIX**) is a family of standards specified by the IEEE Computer Society for maintaining compatibility between operating systems. POSIX defines the application programming interface (API), along with command line shells and utility interfaces, for software compatibility with variants of Unix and other operating systems.

System Calls

- E.g. printf() -> write() -> sys_write()
    - write(1, “Hello World!\n”, 13)
- Make Calling OS services like calling normal API functions
- Whenever a system call is made, Linux can print the system call, its parameters and return values
    - ptrace() traces system calls in Linux
        - Widely used in various debuggers
    - Command line
        - **strace** traces system calls
        - **ltrace** traces library function calls



```
/* run the hello programm*/
execve("./hello", ["./hello"], 0x7ffed5a79e80 /* 64 vars */) = 0
...
/* write``Hello World!\n'' to std output, 1: stdout, 13: 13 chars*/
write(1, "Hello World!\n", 13Hello World!) = 13

/* exit hello*/
exit_group(0)
```



System Call Steps

User address space

App

```
#include <stdio.h>
int main() {
   printf("Hello World!\n");
   return 0;
}
```

libc

```
write(1, "Hello World!\n", 13){
  ...                            /*parsing parameters*/
push $_NR_write      /*first: syscall number*/
push $1                       /*second: file descriptor*/
push "Hello World!\n"
push $13
syscall                       /*instruction differs on different  platforms,
   ...                        e.g., svc for ARM, sysenter for x86,
}                              ecall for RISC-V*/
```

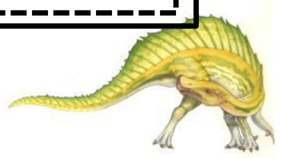
Kernel address space

Trap handler

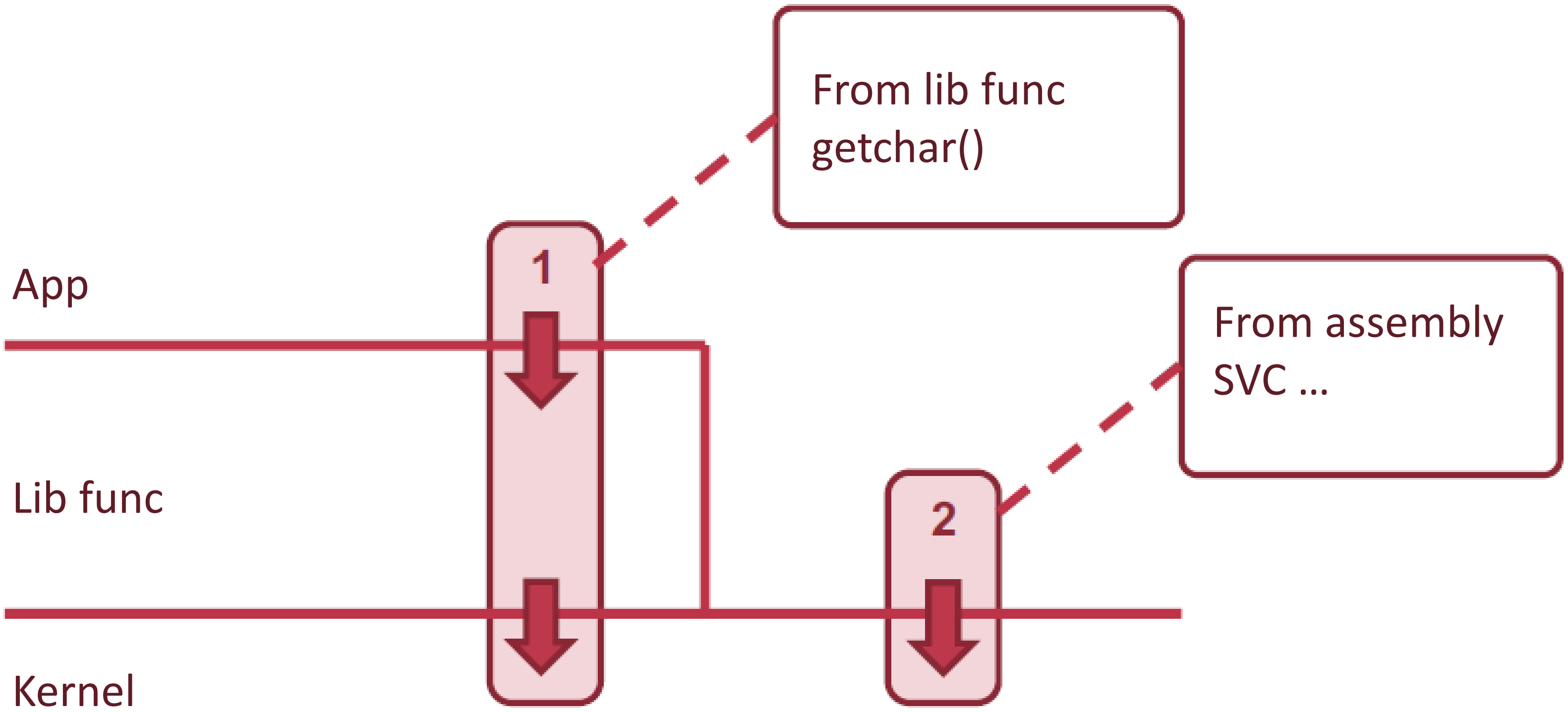
```
sys_syscall:
…
call syscall_table[__NR_write]
…
```

System call handler

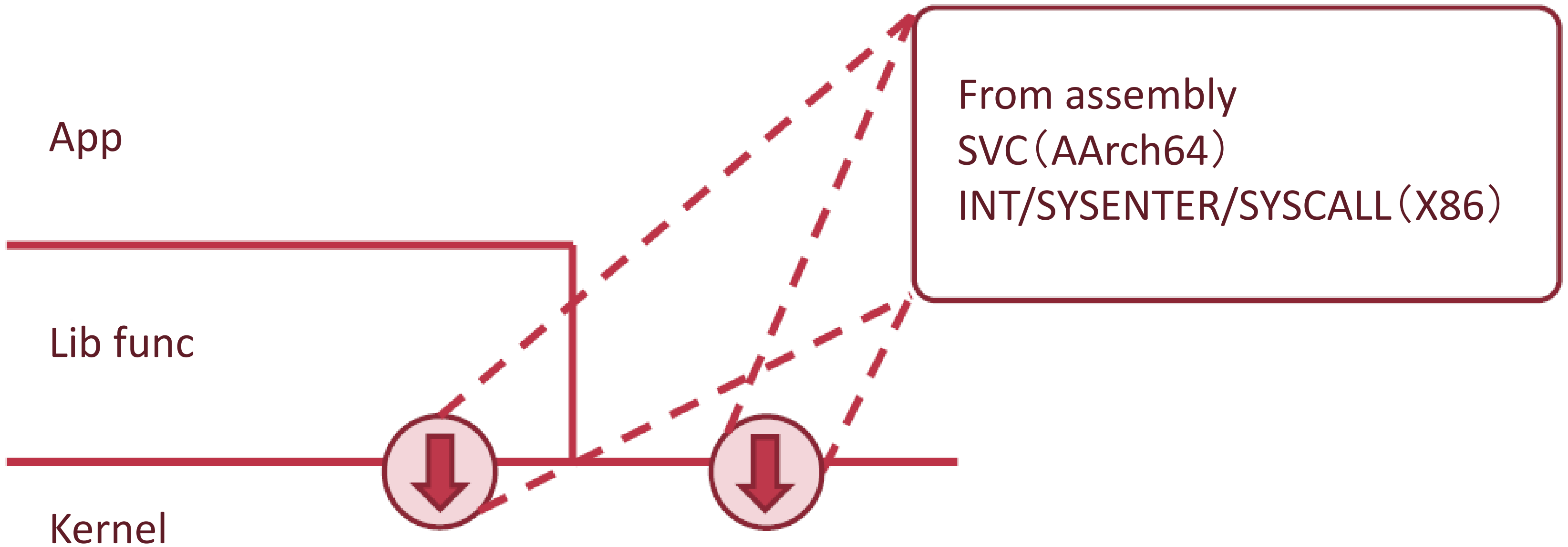
```
sys_write {
…
return error_no;
}
```



System call from APP perspective



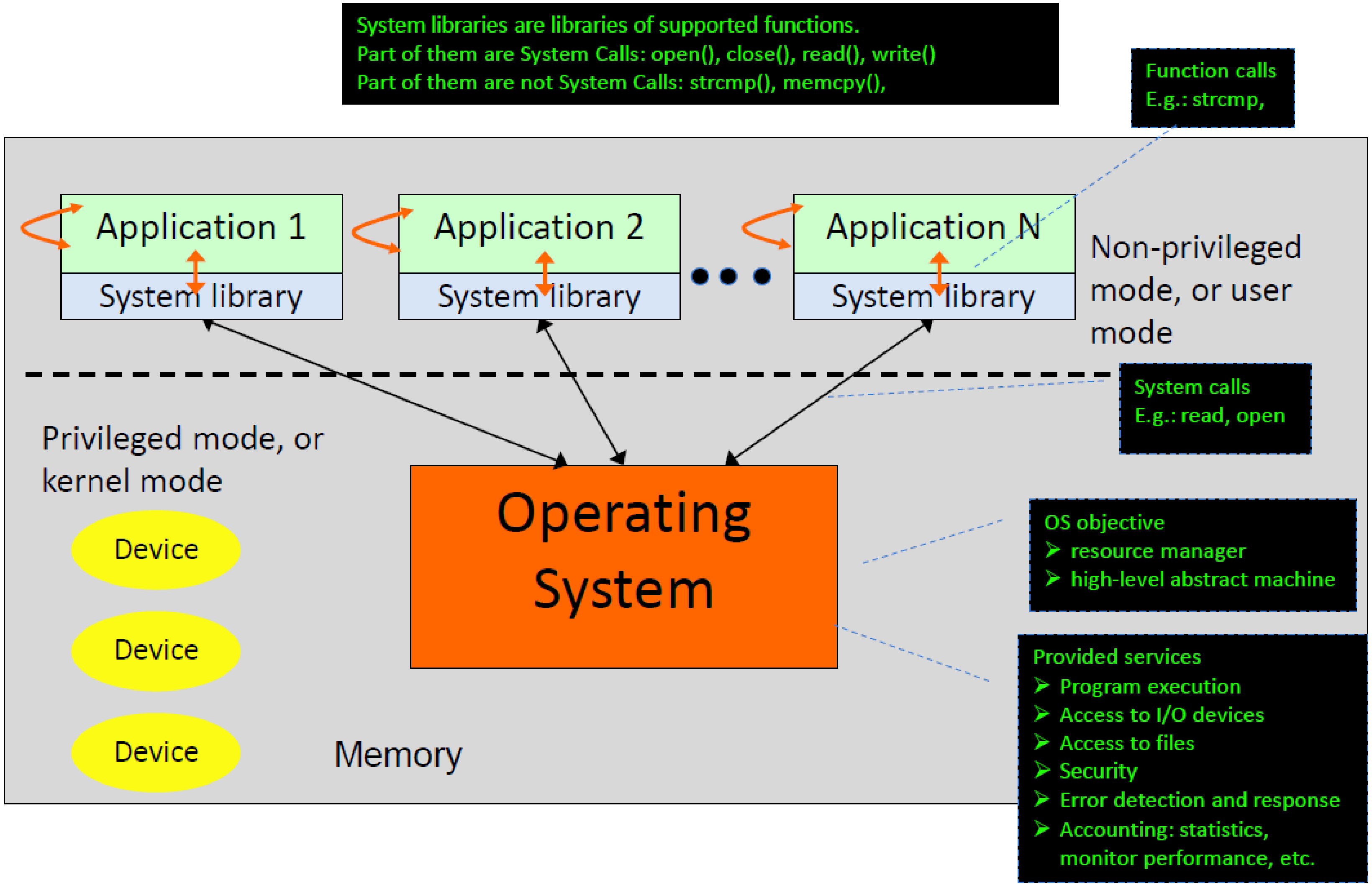
System call from HW perspective



From assembly
SVC（AArch64）
INT/SYSENTER/SYSCALL（X86）

System call

- INT instruction
  - Generate the software interrupt
    - INT 80h
    - 80h: the interrupt vector for system call
- Newer Instructions
  - x86 proposes syscall/sysenter/sysexit to replace INT for system call
- vDSO (virtual dynamic shared object)
  - No need to trap to kernel
  - vsyscall – deprecated
  - Frequent, read only,
  - gettimeofday



System libraries are libraries of supported functions.
Part of them are System Calls: open(), close(), read(), write()
Part of them are not System Calls: strcmp(), memcpy(),

Function calls
E.g.: strcmp,

System calls
E.g.: read, open

OS objective
- resource manager
- high-level abstract machine

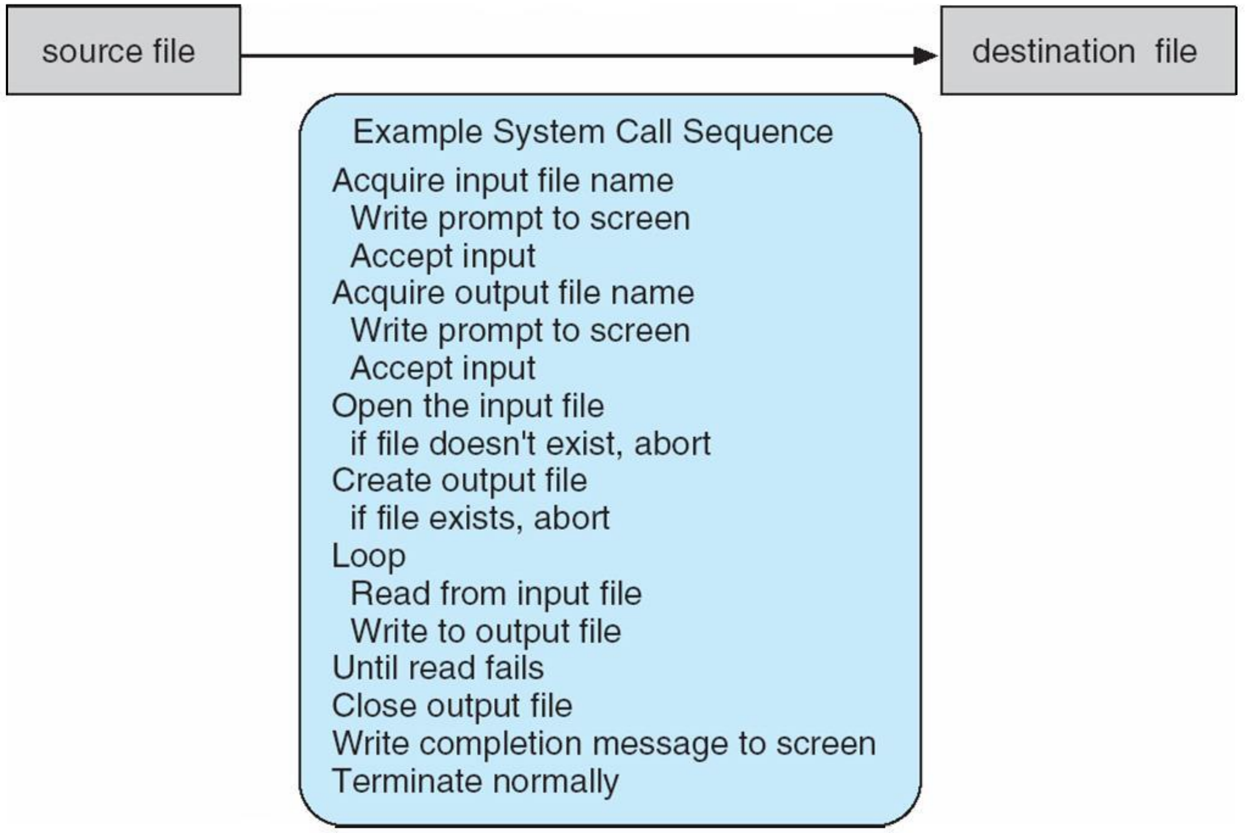
Provided services
- Program execution
- Access to I/O devices
- Access to files
- Security
- Error detection and response
- Accounting: statistics, monitor performance, etc.

System call example

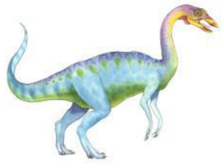
- Copy contents of file A to file B
    - How many system calls involved?

Example of System Calls

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

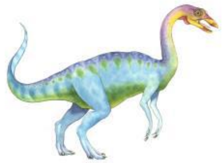


```
Acquire input file name
  Write prompt to screen
  Accept input
Acquire output file name
  Write prompt to screen
  Accept input
Open the input file
  if file doesn't exist, abort
Create output file
  if file exists, abort
Loop
  Read from input file
  Write to output file
Until read fails
Close output file
Write completion message to screen
Terminate normally
```



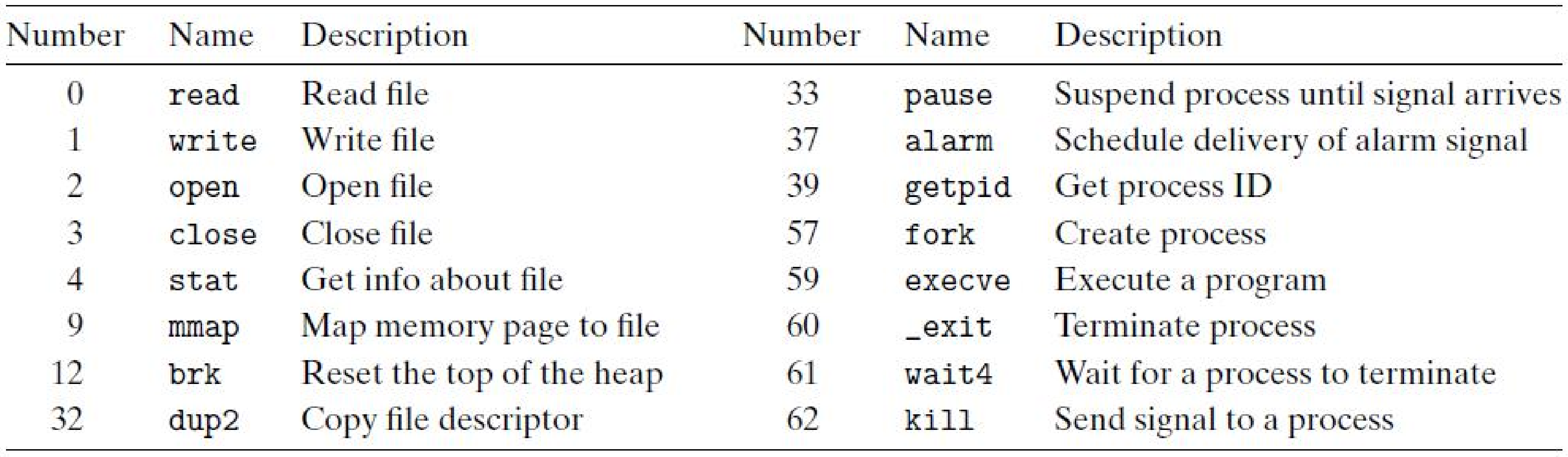
System Call Implementation

- Typically, a number associated with each system call
  - **System-call interface** maintains a table indexed according to these numbers
- The system call interface invokes the intended system call in OS kernel and returns status of the system call and any return values
- The caller need know nothing about how the system call is implemented
  - Just needs to **obey API** and understand what OS will do as a result call
  - Most details of OS interface hidden from programmer by API
    - Managed by run-time support library (set of functions built into libraries included with compiler)



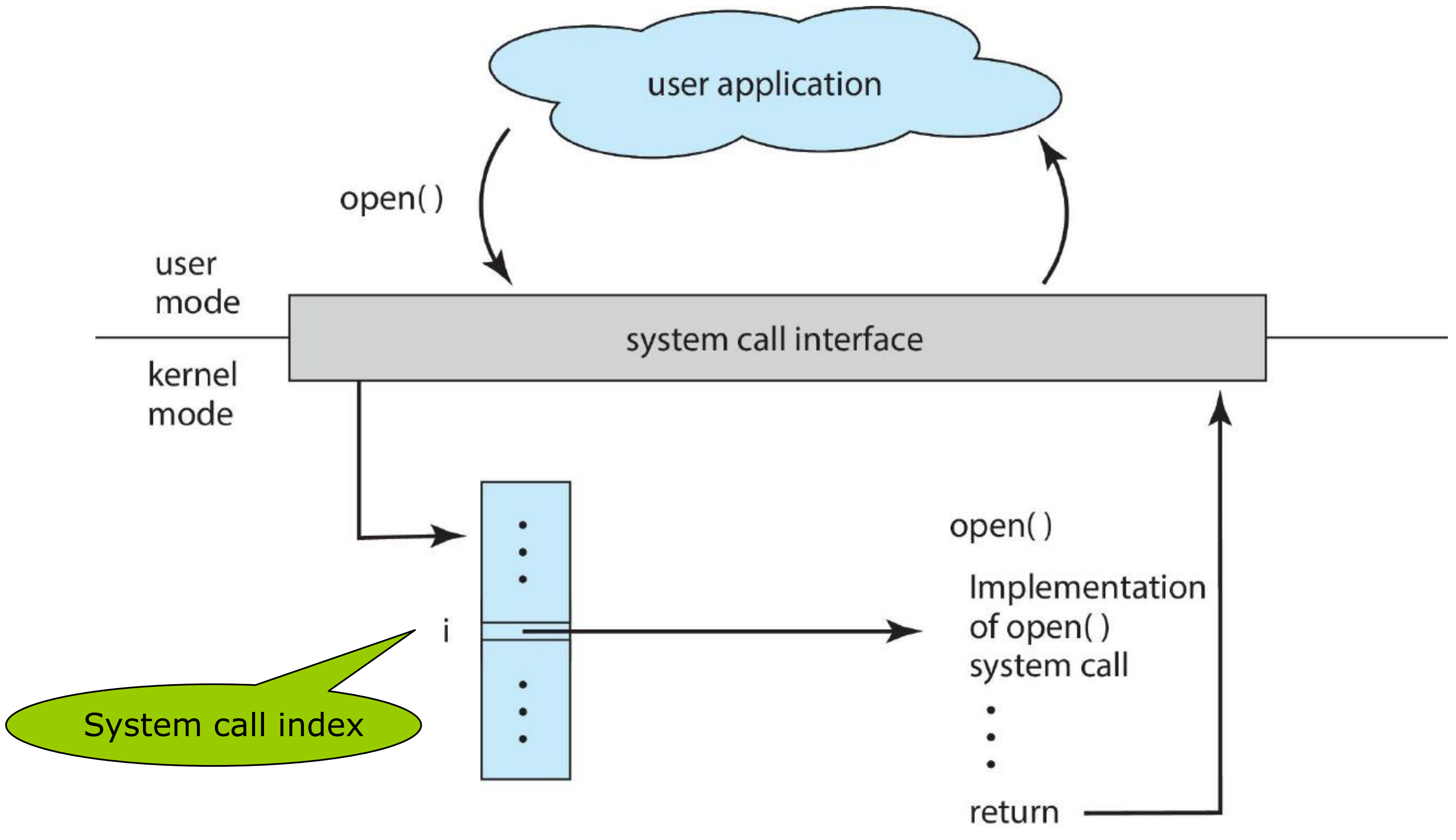
Examples of popular system calls in Linux x86-64 systems

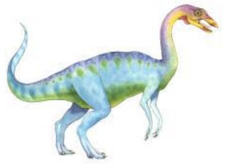
- Notice that this jump table is not the same as the Interrupt Vector Table



Number	Name	Description	Number	Name	Description
0	`read`	Read file	33	`pause`	Suspend process until signal arrives
1	`write`	Write file	37	`alarm`	Schedule delivery of alarm signal
2	`open`	Open file	39	`getpid`	Get process ID
3	`close`	Close file	57	`fork`	Create process
4	`stat`	Get info about file	59	`execve`	Execute a program
9	`mmap`	Map memory page to file	60	`_exit`	Terminate process
12	`brk`	Reset the top of the heap	61	`wait4`	Wait for a process to terminate
32	`dup2`	Copy file descriptor	62	`kill`	Send signal to a process

API – System Call – OS Relationship

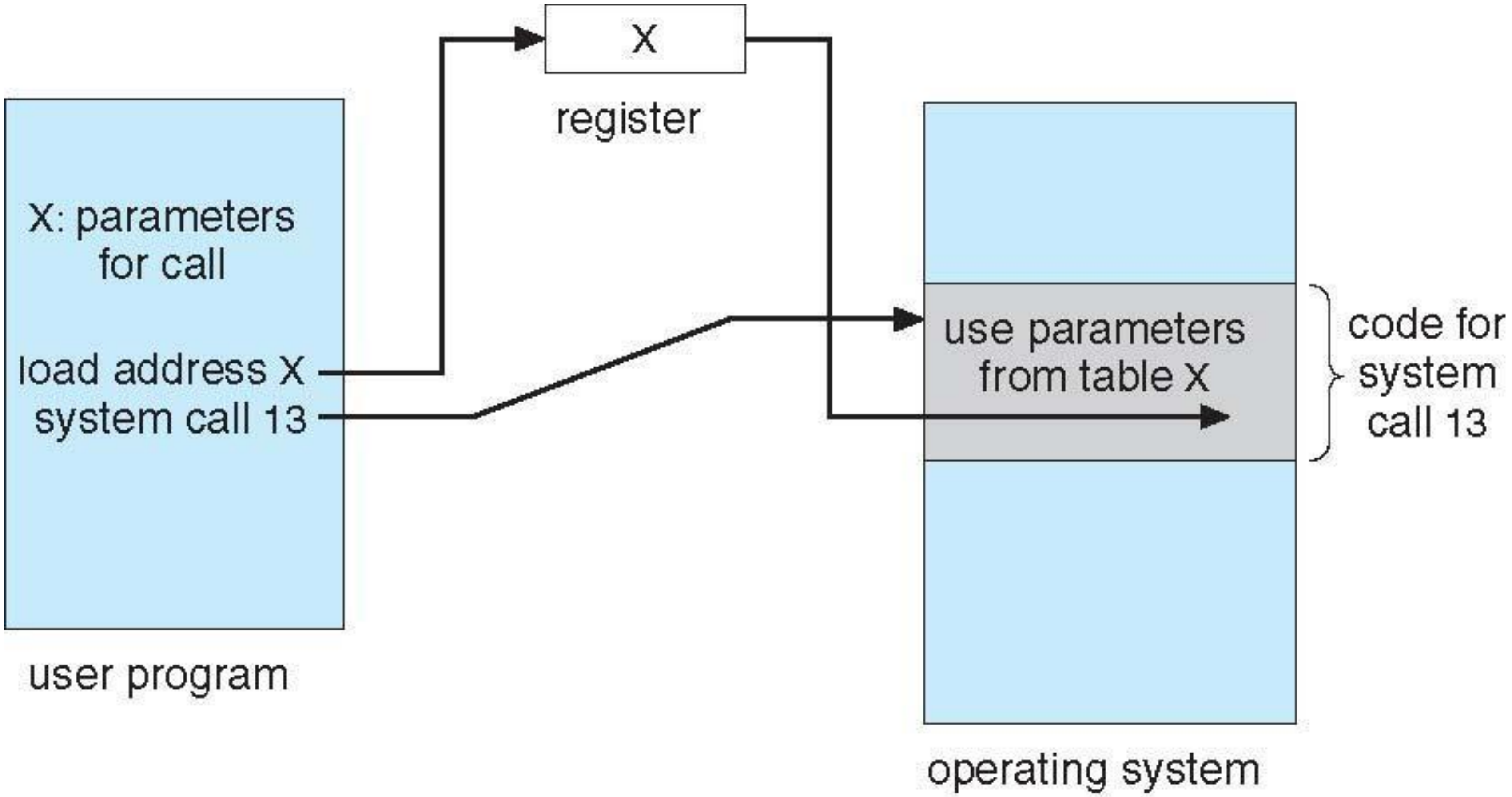


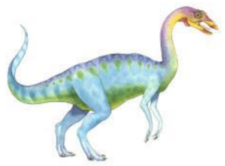


System Call Parameter Passing

- Often, more information is required than simply identity of desired system call
  - Exact type and amount of information vary according to OS and call
- **Three** general methods used to **pass parameters to the OS**
  - Simplest/Fastest: pass the parameters in registers
    - In some cases, may be more parameters than registers
    - By convention, register %rax contains the syscall number, with up to six arguments in %rdi, %rsi,%rdx, %r10, %r8, and %r9.
    - On return from the system call, registers %rcx and %r11 are destroyed, and %rax contains the return value
  - Parameters stored in a block, or table, in memory, and address of block passed as a parameter in a register
    - This approach taken by Linux and Solaris
  - Parameters placed, or **pushed**, onto the **stack** by the program and **popped** off the stack by the operating system
  - Block and stack methods do not limit the number or length of parameters being passed

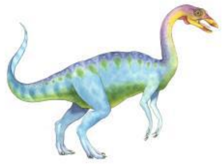
Parameter Passing via Table





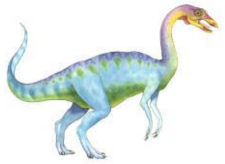
Types of System Calls

- Process control
  - create process, terminate process
  - end, abort
  - load, execute
  - get process attributes, set process attributes
  - wait for time
  - wait event, signal event
  - allocate and free memory
  - Dump memory if error
  - **Debugger** for determining **bugs, single step** execution
  - **Locks** for managing access to shared data between processes



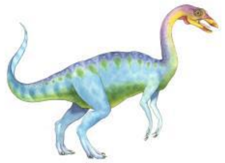
Types of System Calls (cont.)

- File management
  - create file, delete file
  - open, close file
  - read, write, reposition
  - get and set file attributes
- Device management
  - request device, release device
  - read, write, reposition
  - get device attributes, set device attributes
  - logically attach or detach devices



Types of System Calls (Cont.)

- Information maintenance
  - get time or date, set time or date
  - get system data, set system data
  - get and set process, file, or device attributes
- Communications
  - create, delete communication connection
  - send, receive messages if **message passing model** to **host name** or **process name**
    - From **client** to **server**
  - **Shared-memory model** create and gain access to memory regions
  - transfer status information
  - attach and detach remote devices

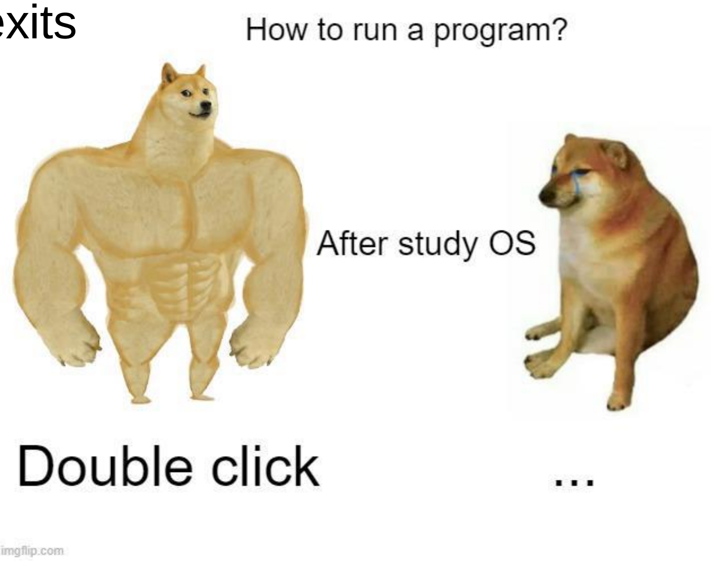


Types of System Calls (Cont.)

- Protection
  - Control access to resources
  - Get and set permissions
  - Allow and deny user access

Essentially, all programs are similar to Hello World

- Program = compute → syscall → compute →
- Loaded by the operating system
- Execute **execve** by another process to set to initial state
- Program execution
  - Process management: **fork**, **execve**, **exit**, ...
  - File/device management: **open**, **close**, **read**, **write**, ...
  - Storage management: **mmap**, **brk**, ...
  - until _exit (**exit_group**) exits



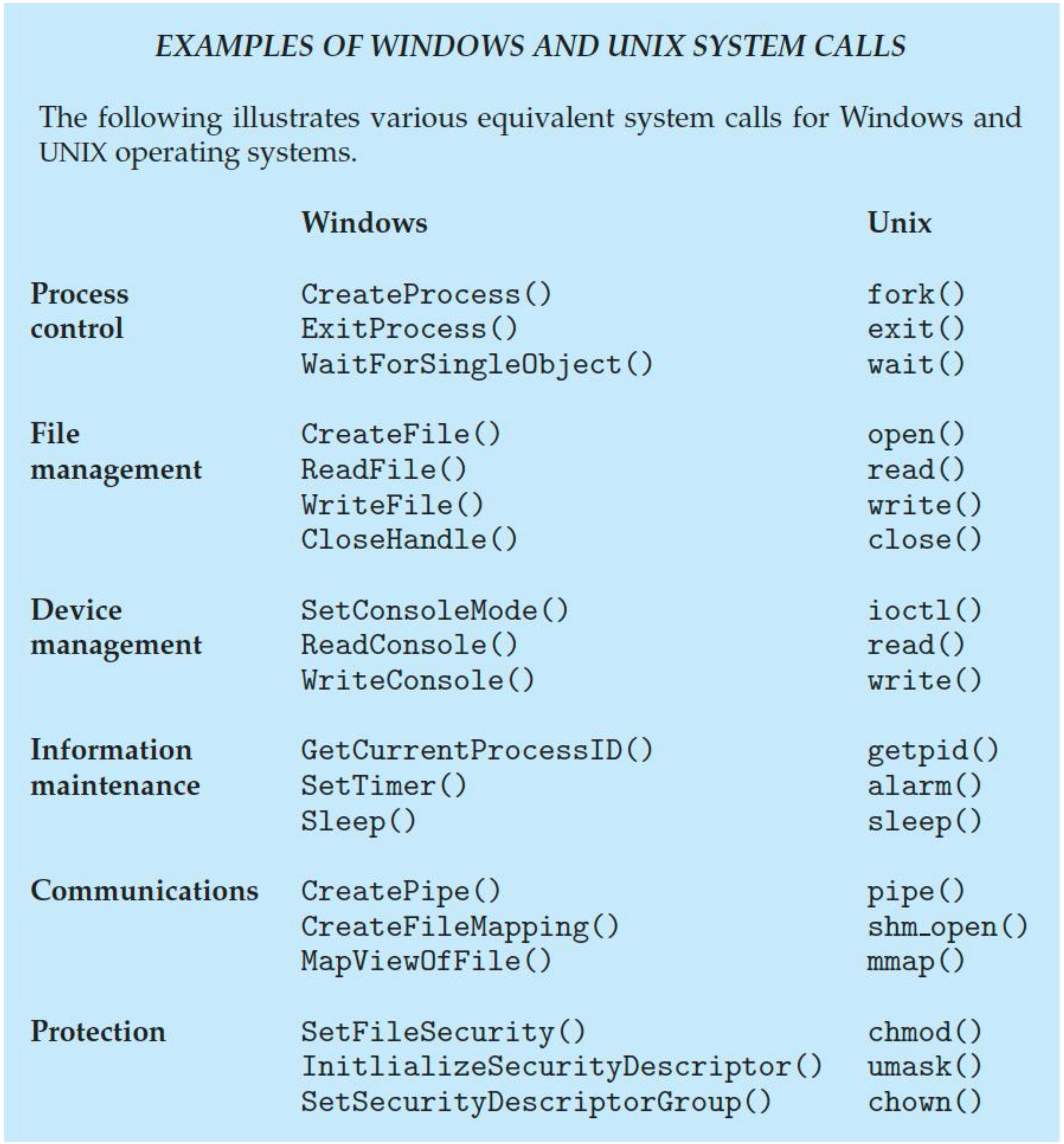
Apps are based on OS API (syscalls) and objects

- Window manager
  - Manage devices and screens (read/write/mmap)
  - Inter process communication (send, recv)
- Task manager
  - Access to process objects provided by the os (readdir/read)
  - Refer to man proc, info proc * in gdb
- Antivirus software
  - File static scan (read)
  - Active Defense (ptrace)
  - Other more complex security mechanisms...

Apps are based on OS API (syscalls) and objects

- Window manager
  - Manage devices and screens (read/write/mmap)
  - Inter process communication (send, recv)
- Task manager
  - Access to process objects provided by the os (readdir/read)
  - Refer to man proc, info proc * in gdb
- Antivirus software
  - File static scan (read)
  - Active Defense (ptrace)
  - Other more complex security mechanisms...

Examples of Windows and Unix System Calls



EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS

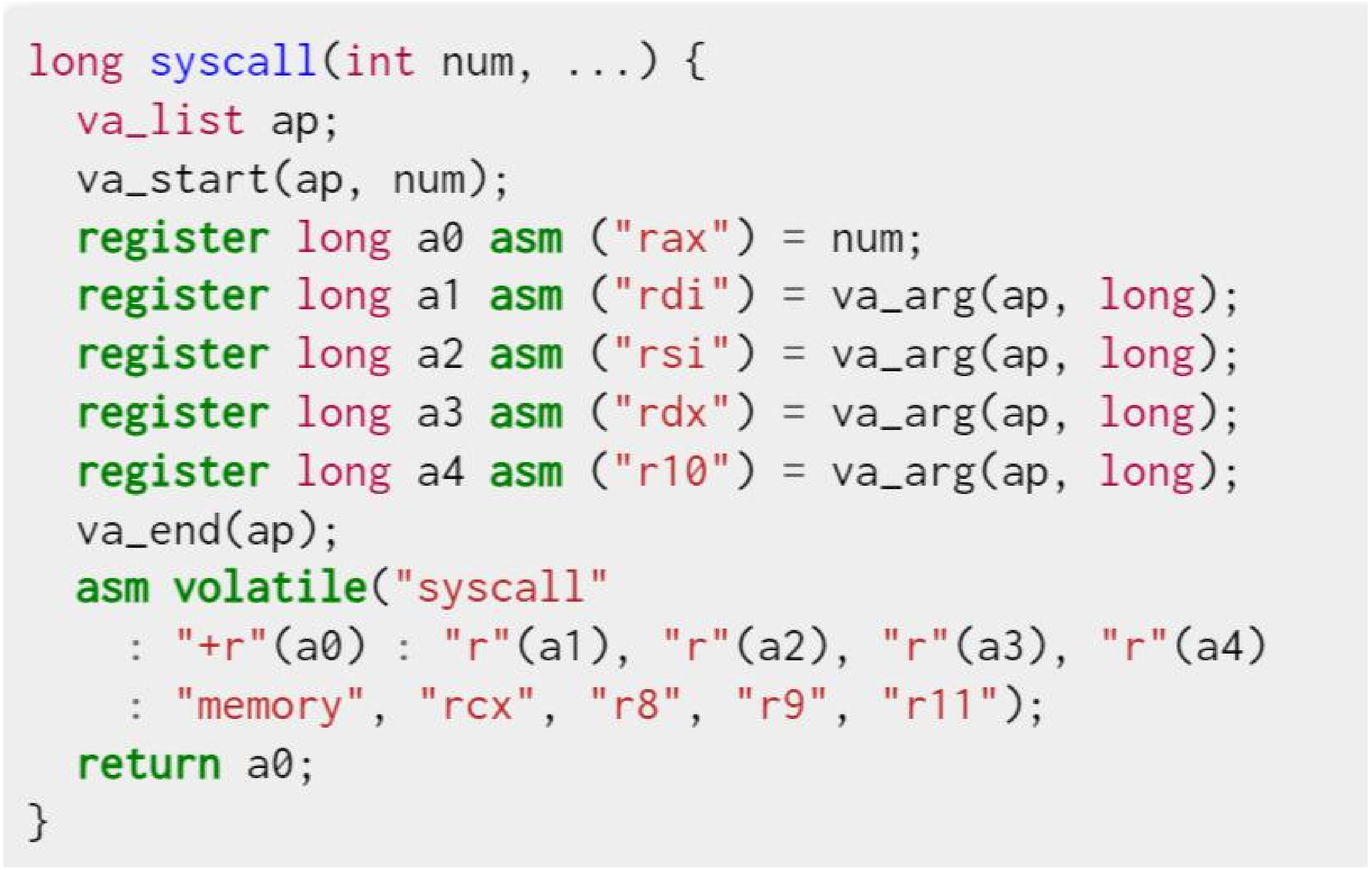
The following illustrates various equivalent system calls for Windows and UNIX operating systems.

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

syscall (2), syscalls (2)

Why libc?

- Can we use only “syscall”?
- It is possible (definitely doable), but not very friendly.



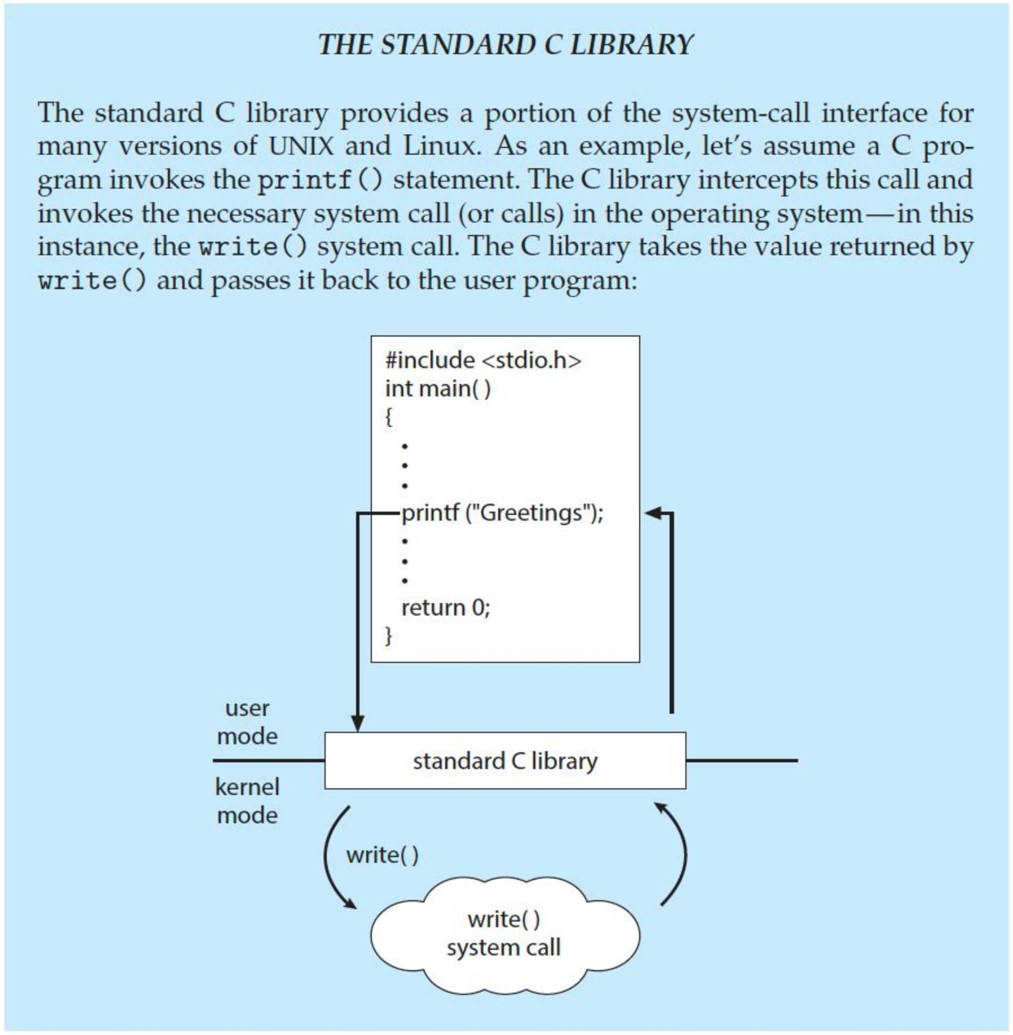
```c
long syscall(int num, ...) {
  va_list ap;
  va_start(ap, num);
  register long a0 asm ("rax") = num;
  register long a1 asm ("rdi") = va_arg(ap, long);
  register long a2 asm ("rsi") = va_arg(ap, long);
  register long a3 asm ("rdx") = va_arg(ap, long);
  register long a4 asm ("r10") = va_arg(ap, long);
  va_end(ap);
  asm volatile("syscall"
    : "+r"(a0) : "r"(a1), "r"(a2), "r"(a3), "r"(a4)
    : "memory", "rcx", "r8", "r9", "r11");
  return a0;
}
```

Difference between system call and function call

- System call: a call into kernel code, typically performed by executing an interrupt
- Function call, if calling a system call (via library), switches into kernel mode from user mode

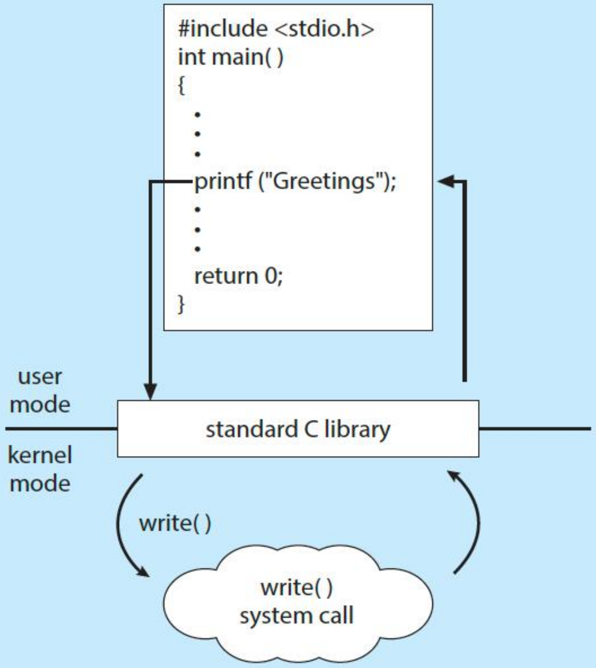
Standard C Library Example

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



THE STANDARD C LIBRARY

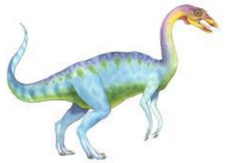
The standard C library provides a portion of the system-call interface for many versions of UNIX and Linux. As an example, let's assume a C program invokes the `printf()` statement. The C library intercepts this call and invokes the necessary system call (or calls) in the operating system—in this instance, the `write()` system call. The C library takes the value returned by `write()` and passes it back to the user program:



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

Standard C Library Example

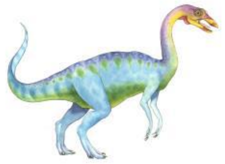
- In the main function of your C program you call the printf function.
  - The code (microprocessor instructions) for printf itself is in the C Standard Library.
    - This C Standard Library code is linked (glued) with your own code when you create the .exe executable program at compile time (more on linking later).
- The code of printf itself calls the write system call.
  - The write system call is a C function which is part of the code of the kernel, which is always in memory.
  - When printf calls the write system call, the CPU switches from user mode to kernel mode and at the same time the CPU jumps from executing the code of printf to executing the code of the write system call from the kernel.
- When the write system call is finished, the reverse process happens: the CPU switches from kernel mode to user mode and at the same time jumps back from executing the code of the write system call in the kernel to executing the rest of the code of printf in the C Standard Library. When printf itself is finished the CPU goes back to execute the rest of the main function of your program.



Standard C Library Example

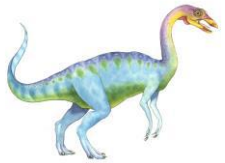
- Note: the write system call exists only on Unix/Linux; on Microsoft Windows the write system call does not exist and the code of printf itself uses the WriteConsole system call instead.
  - So the code of printf itself is different on different operating systems.
    - In fact the code of the whole **C Standard Library** itself **is different on different operating systems**!
  - Your own code that uses printf does not need to change though because all these implementations of printf on different operating systems are required by the C language to all work the same way

    (**even though internally they use different system calls**).
  - Every time you move your code **to a different operating system**, you do not need to change your source code that uses printf, but **you must recompile** it to use the code of printf for the new operating system which is in the C Standard Library that comes with the new operating system.



Standard C Library Example

- **Advantages of using C Standard Library functions**, like printf, scanf, ...
  - It makes your code **portable**: you do not need to change your code when you move your code to a new kind of computer.
    - You only need to recompile the code to use the new C Standard Library for the new computer.
  - Functions from the C Standard Library are usually **easier to use** than system calls.
- **Advantages of using system calls** in your code directly, like write, read, …
  - System calls are usually **more powerful** than functions from the C Standard Library: you can create processes, share memory between processes, etc. These advanced features are not available in the C Standard Library.
  - It's **a little bit faster** than using a library function (since internally the library function uses a system call anyway).
- In practice, for normal programmers, code portability is the most important issue, so use functions from the C Standard Library as much as possible in your own code!



Standard C Library Example

```
#include <stdio.h>
#define SIZE 1024

int main(void) {
  char prompt[] = "Type a command: ";
  char buf[SIZE];

  // Ask the user to type a command:
  printf(prompt);

  // Read a string from kb using fgets.
  // Note that fgets also puts into the array buf
  // the '\n' character typed by the user
  // when the user presses the Enter key):
  fgets(buf, SIZE, stdin);
  // Replace the Enter key with '\0':
  for(int i = 0; i < SIZE; i++) {
    if(buf[i] == '\n' || buf[i] == '\r') {
      buf[i] = '\0';
      break;
    }
  }

  // print the command:
  printf("command: %s\n", buf);
  return 0;
}
```

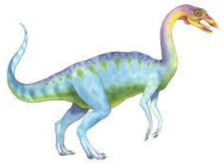
scanf("%s", buf);

\r:carriage return, ASCII: 13
\n:new line, ASCII: 10

"ENTER KEY"
Linux: \n
Windows: \r\n
Mac: \r

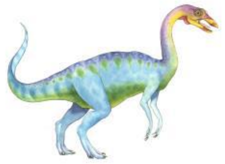
System calls C Example

```
#include <unistd.h>
#include <string.h>
#define SIZE 1024
int main(void) {
          char prompt[] = "Type a command: ";
          char buf[SIZE];

          // Ask the user to type a command:
          write(1, prompt, strlen(prompt));

          // Read from the standard input the command typed by the user (note
          // that read puts into the array buf the ’\n’ character typed
          // by the user when the user presses the Enter key on the keyboard):
          read(0, buf, SIZE);

          // Replace the Enter key typed by the user with ’\0’:
          for(int i = 0; i < SIZE; i++) {
                    if(buf[i] == '\n' || buf[i] == '\r') {
                              buf[i] = '\0';
                              break;
                    }
          }

          // output the user typed command:
          write(1, buf, strlen(buf));
          write(1, "\n", 1);

          return 0;
}
```



Why Applications are Operating System Specific

- Apps compiled on one system usually not executable on other operating systems
- Each operating system provides its own **unique system calls**
  - Own file formats, etc.
- Apps can be multi-operating system
  1. Written in interpreted language like **Python, Ruby**, and interpreter available on multiple operating systems
  2. App written in language that includes a **VM** containing the running app (like **Java**)
  3. Use standard language (like C), compile separately on each operating system to run on each OS
- **Application Binary Interface** (**ABI**) is architecture equivalent of API, defines how different components of binary code can interface for a given operating system on a given architecture, CPU, etc.



Operating System Design and Implementation

- Design and Implementation of OS not "solvable", but some **approaches** have proven successful
- Internal structure of different Operating Systems can vary widely
- Start the design by defining **goals** and specifications
- Affected by choice of **hardware**, type of system
- **User** goals and **System** goals
  - User goals – operating system should be convenient to use, easy to learn, reliable, safe, and fast
  - System goals – operating system should be easy to design, implement, and maintain, as well as flexible, reliable, error-free, and efficient

Operating System Design and Implementation

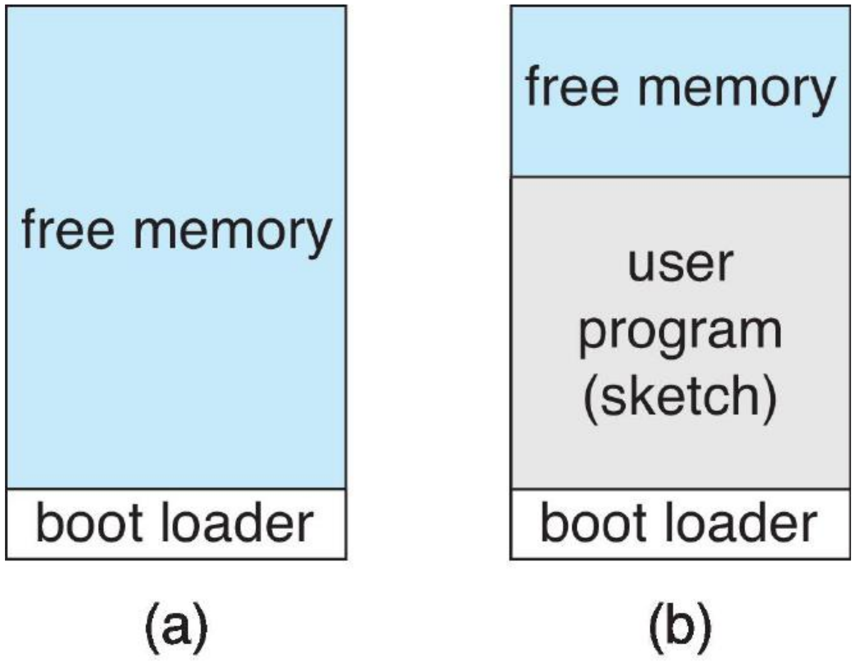
- Important principle to separate

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

- Mechanisms determine how to do something, policies decide what will be done
- The separation of policy from mechanism is a very important principle, it allows maximum flexibility if policy decisions are to be changed later (example – timer)
- Specifying and designing an OS is **highly creative task** of **software engineering**

Arduino

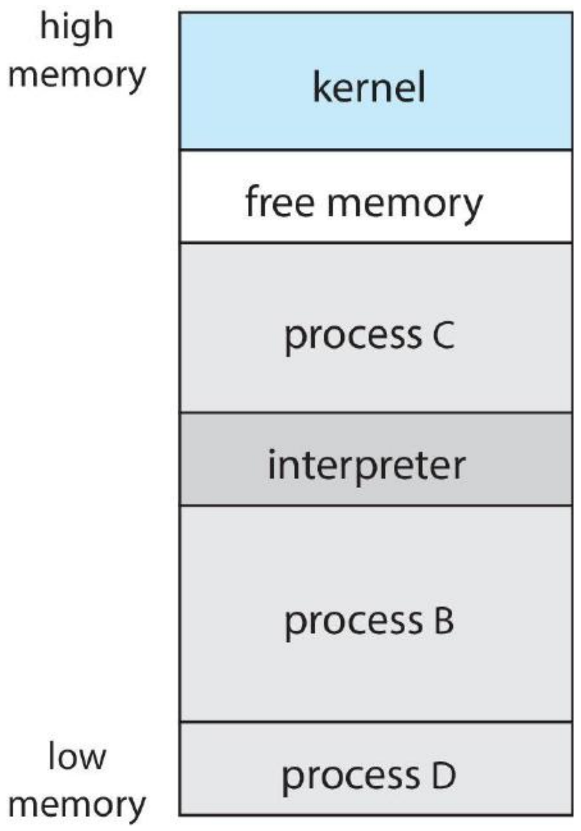
- Single-tasking
- No operating system
- Programs (sketch) loaded via USB into flash memory
- Single memory space
- Boot loader loads program
- Program exit -> shell reloaded



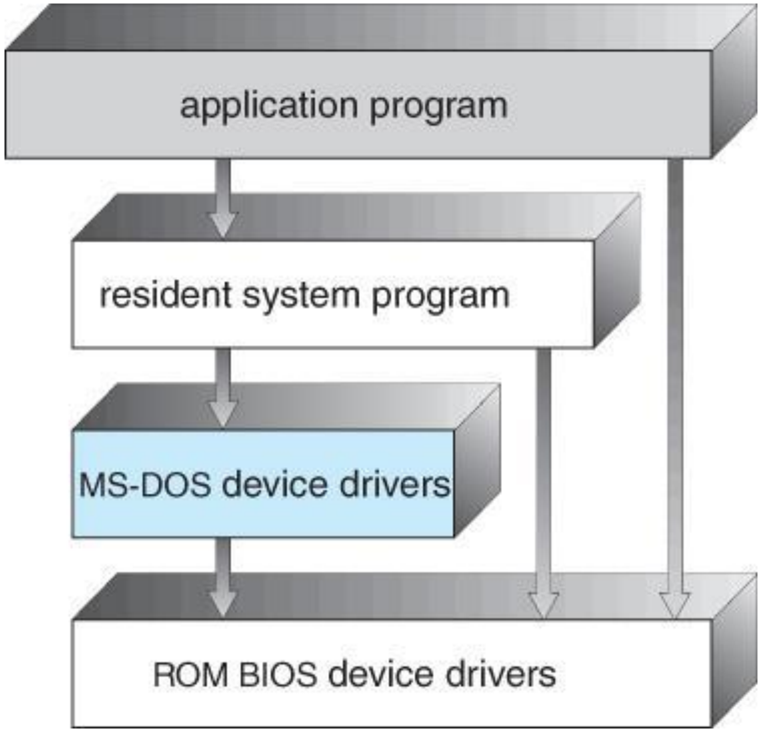
At system startup running a program

FreeBSD

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

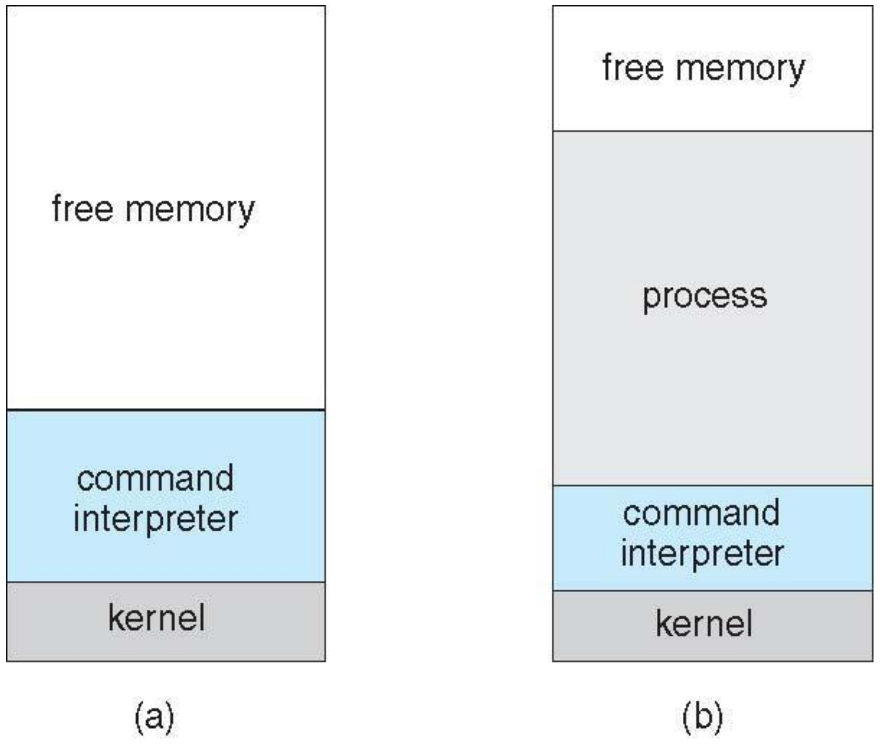


DOS OS


application program
resident system program
MS-DOS device drivers
ROM BIOS device drivers

DOS OS

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



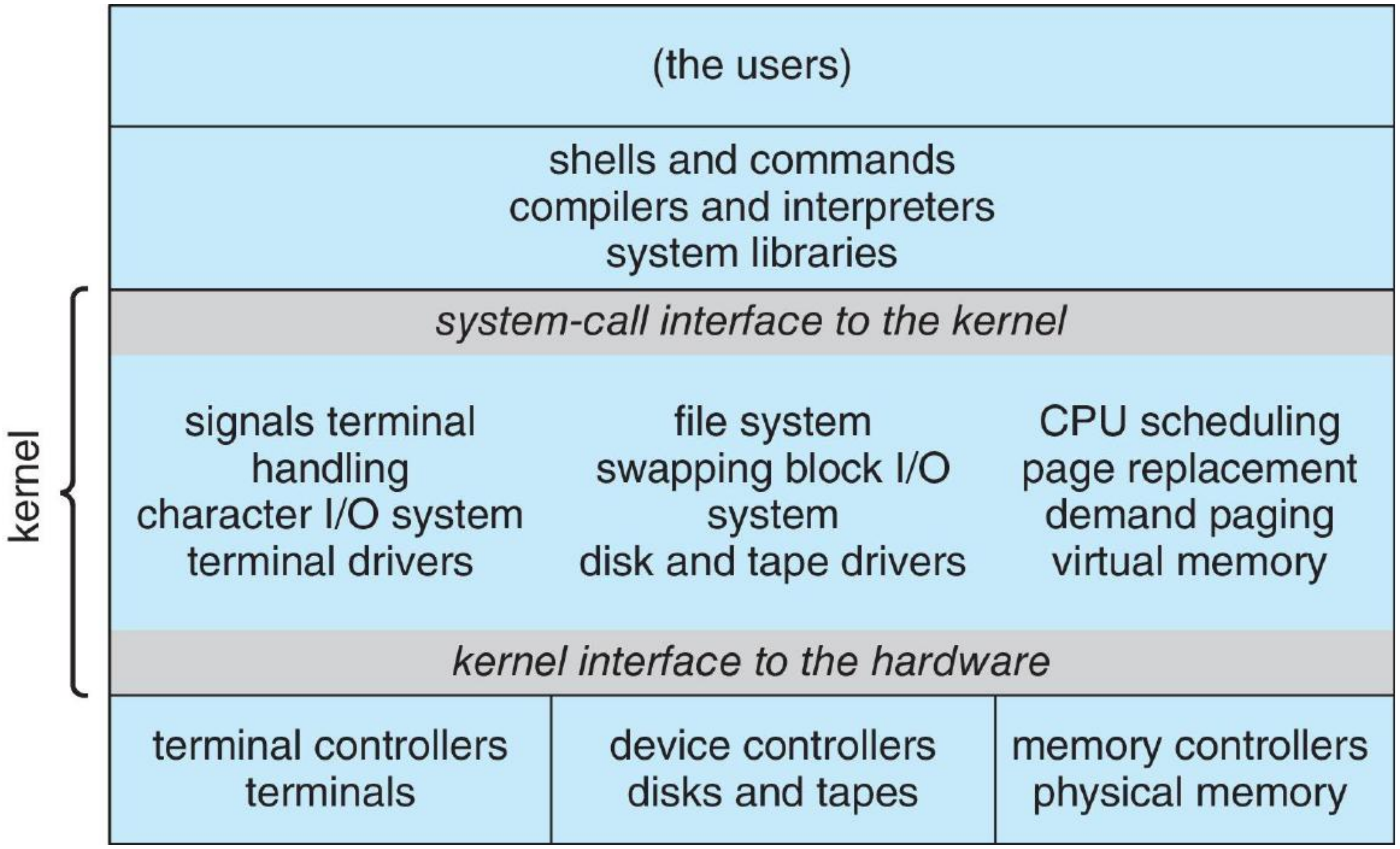
Monolithic Structure – Original UNIX

UNIX – limited by hardware functionality, the original UNIX operating system had limited structuring. The UNIX OS consists of two separable parts

- **Systems programs**
- **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; a large number of functions for one level

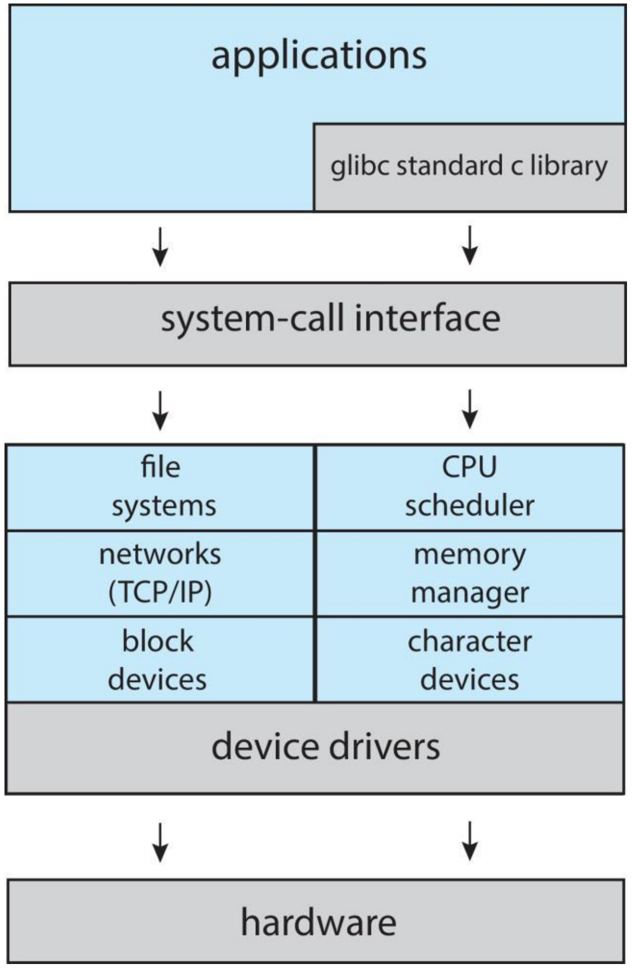
Traditional UNIX System Structure

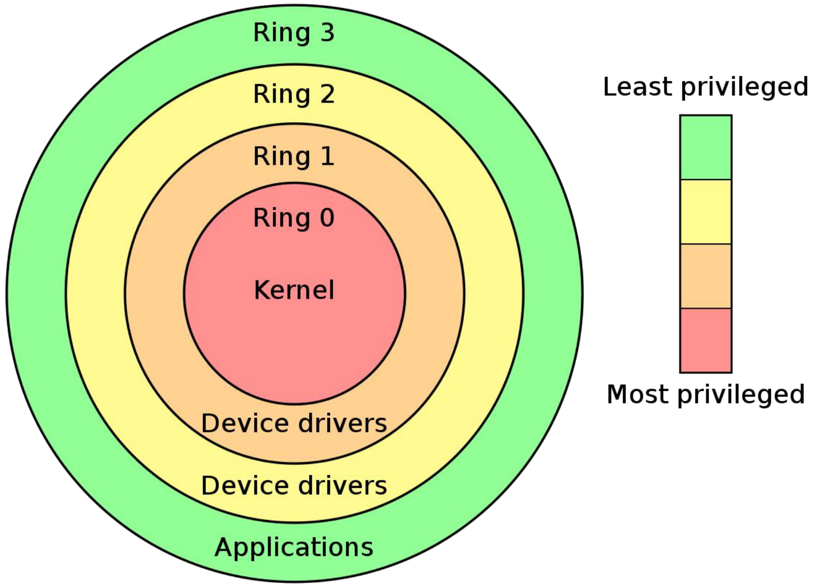
Beyond simple but not fully layered

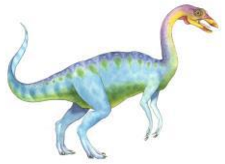


Linux System Structure

Monolithic plus **modular** design





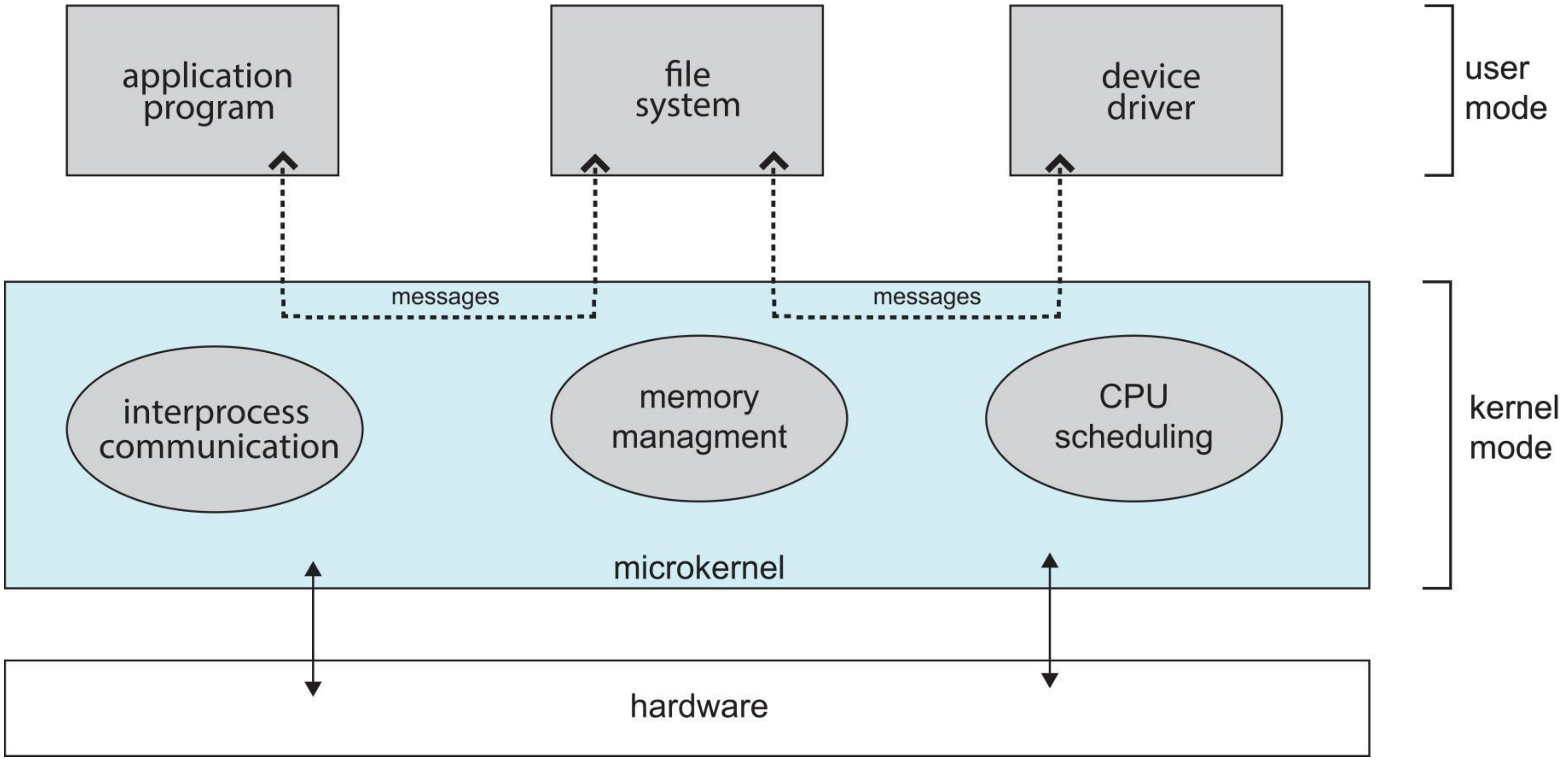


Microkernels

- Moves as much from the kernel into user space
- **Mach[1]** example of **microkernel**
  - Mac OS X kernel (**Darwin**) partly based on Mach
- 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

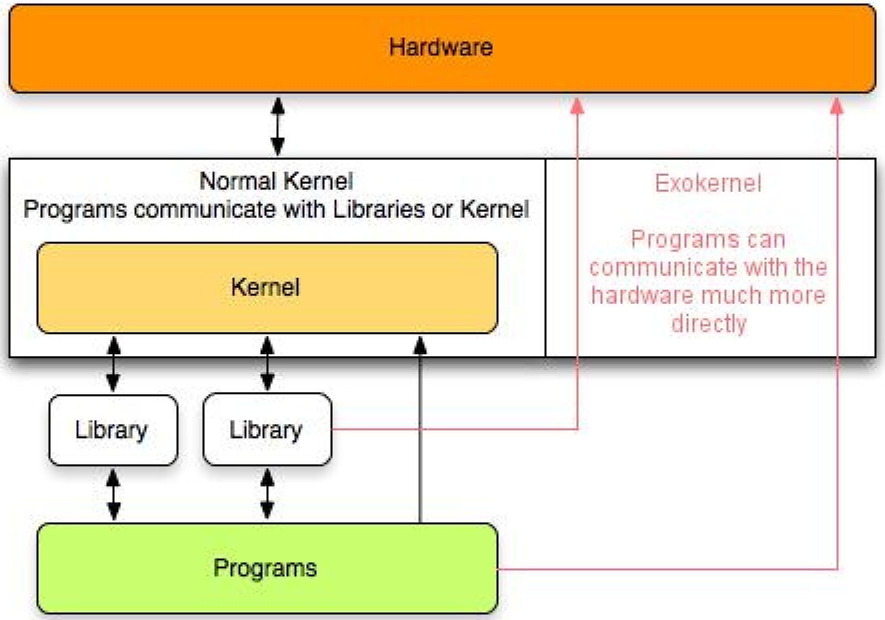
[1]Mach is a kernel developed at Carnegie Mellon University to support operating system research, primarily distributed and parallel computing. Mach is often mentioned as one of the earliest examples of a microkernel.

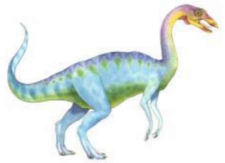
Microkernel System Structure



Exokernel architecture

- Proposed by MIT
- Application-oriented OS
  - Gives them all control
- Research

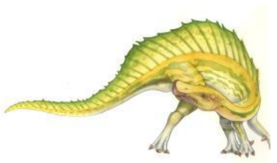

Hardware
Normal Kernel
Programs communicate with Libraries or Kernel
Kernel
Exokernel
Programs can communicate with the hardware much more directly
Library
Library
Programs



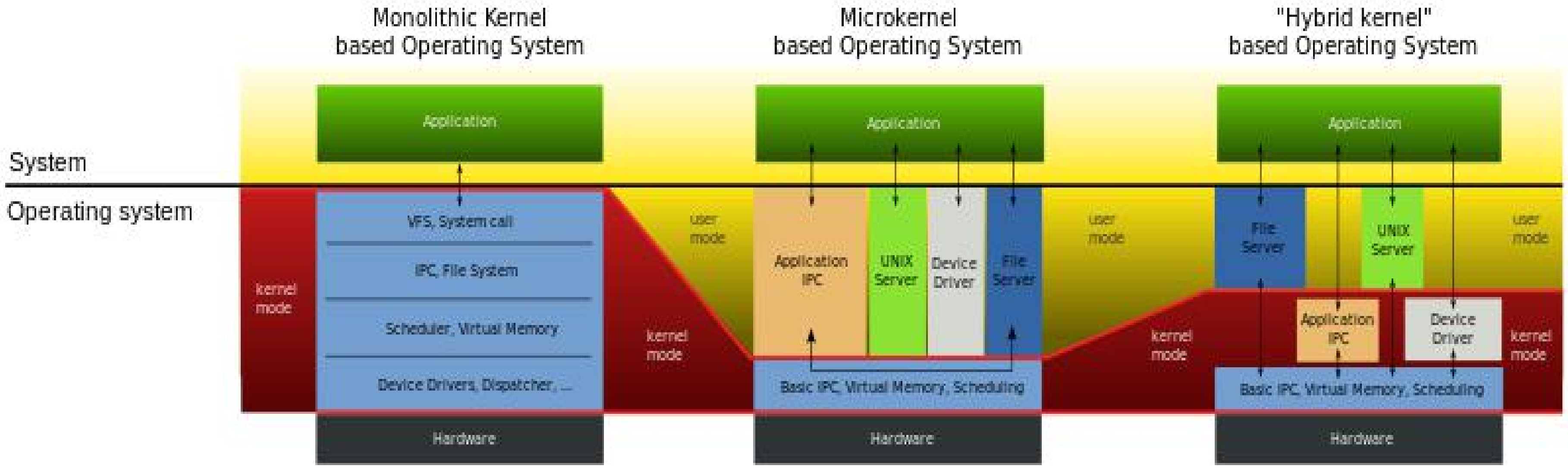
Hybrid Systems

- Most modern operating systems are actually not one pure model
  - **Hybrid** combines multiple approaches to address performance, security, usability needs
  - **Linux** and Solaris kernels in kernel address space, so **monolithic**, **plus modular** for dynamic loading of functionality
  - Windows mostly **monolithic, plus microkernel** for different subsystem ***personalities***
- Apple Mac OS X hybrid, layered, **Aqua[1]** UI plus **Cocoa[2]** programming environment
  - Below is kernel consisting of Mach microkernel and BSD Unix parts, plus I/O kit and dynamically loadable modules (called **kernel extensions**)

[1]Aqua is the graphical user interface (GUI) and visual theme of Apple's macOS operating system. It was originally based around the theme of water, with droplet-like components and a liberal use of reflection effects and translucency. Its goal is to "incorporate color, depth, translucence, and complex textures into a visually appealing interface" in macOS applications. At its introduction, Steve Jobs noted that "one of the design goals was when you saw it you wanted to lick it".
[2] **Cocoa** is Apple's native object-oriented application programming interface (API) for their operating system macOS.



Comparison



Monolithic Kernel based Operating System
Microkernel based Operating System
"Hybrid kernel" based Operating System
System
Operating system
Application
VFS, System call
IPC, File System
Scheduler, Virtual Memory
Device Drivers, Dispatcher, ...
Hardware
kernel mode
user mode
Application IPC
UNIX Server
Device Driver
File Server
Basic IPC, Virtual Memory, Scheduling

Questions

- Is the operating system using the CPU at all times?
- After booting and initialization, what are the circumstances that would cause OS code to use the CPU?

End of Chapter 3