OS Structures

Operating Systems

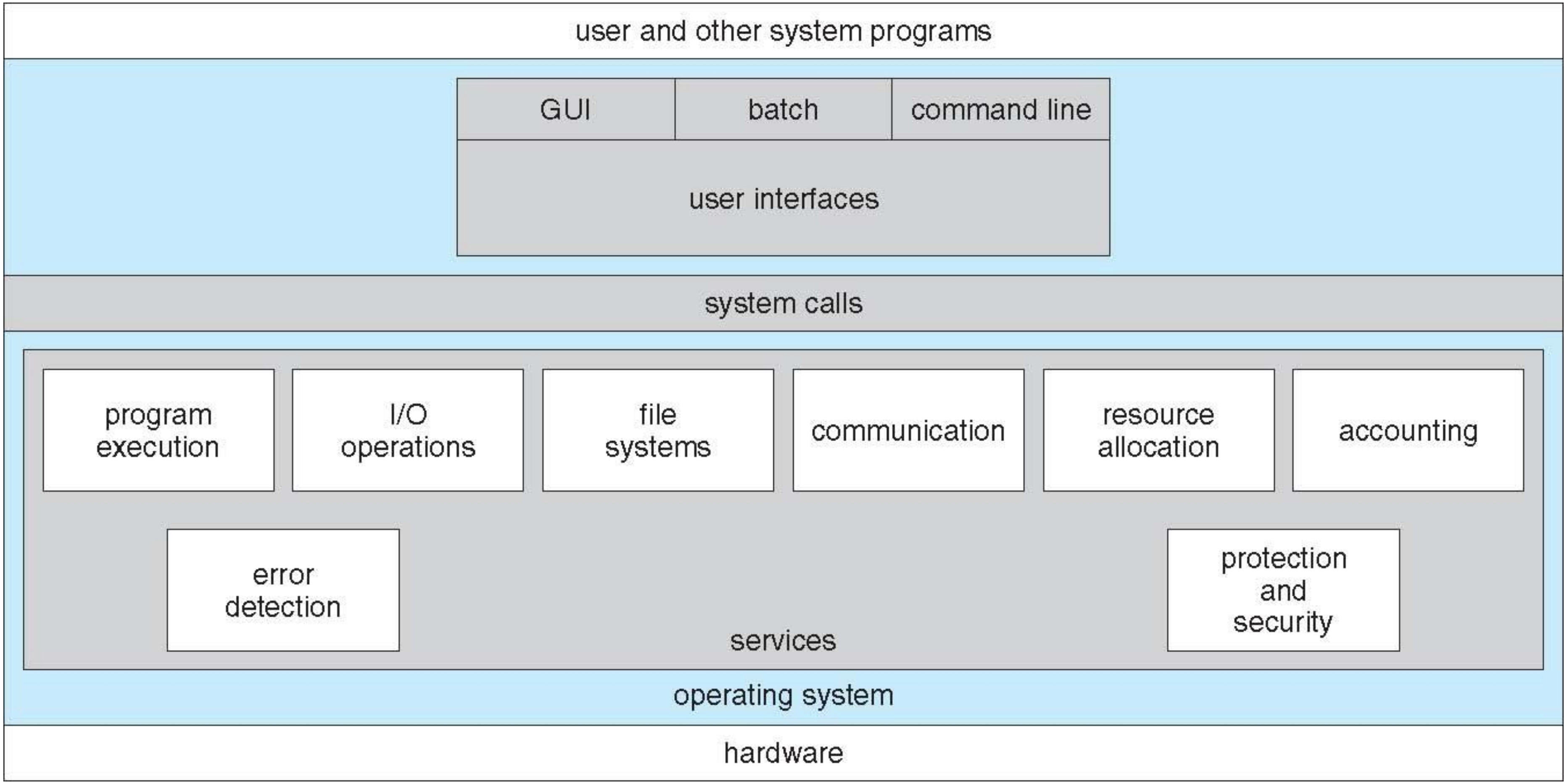
Wenbo Shen

Some slides are from OSC and Henri Casanova@Hawaii

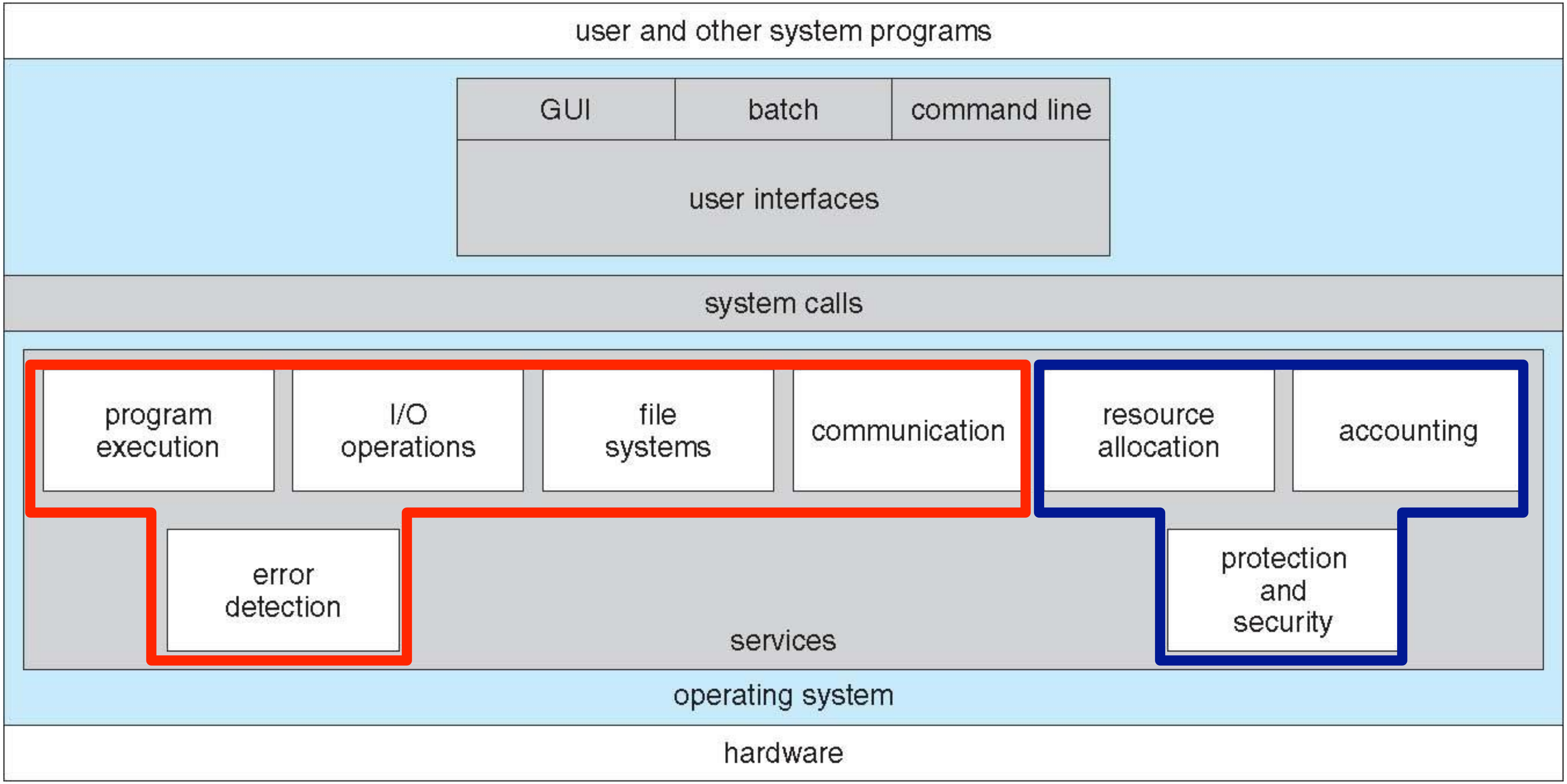
OS Structures

- Operating System Services
- User and Operating System-Interface
- System Calls
- System Services
- Linkers and Loaders
- Why Applications are Operating System Specific
- Operating-System Design and Implementation
- Operating System Structure
- Building and Booting an Operating System
- Operating System Debugging

A View of Operating System Services



OS Services and Features

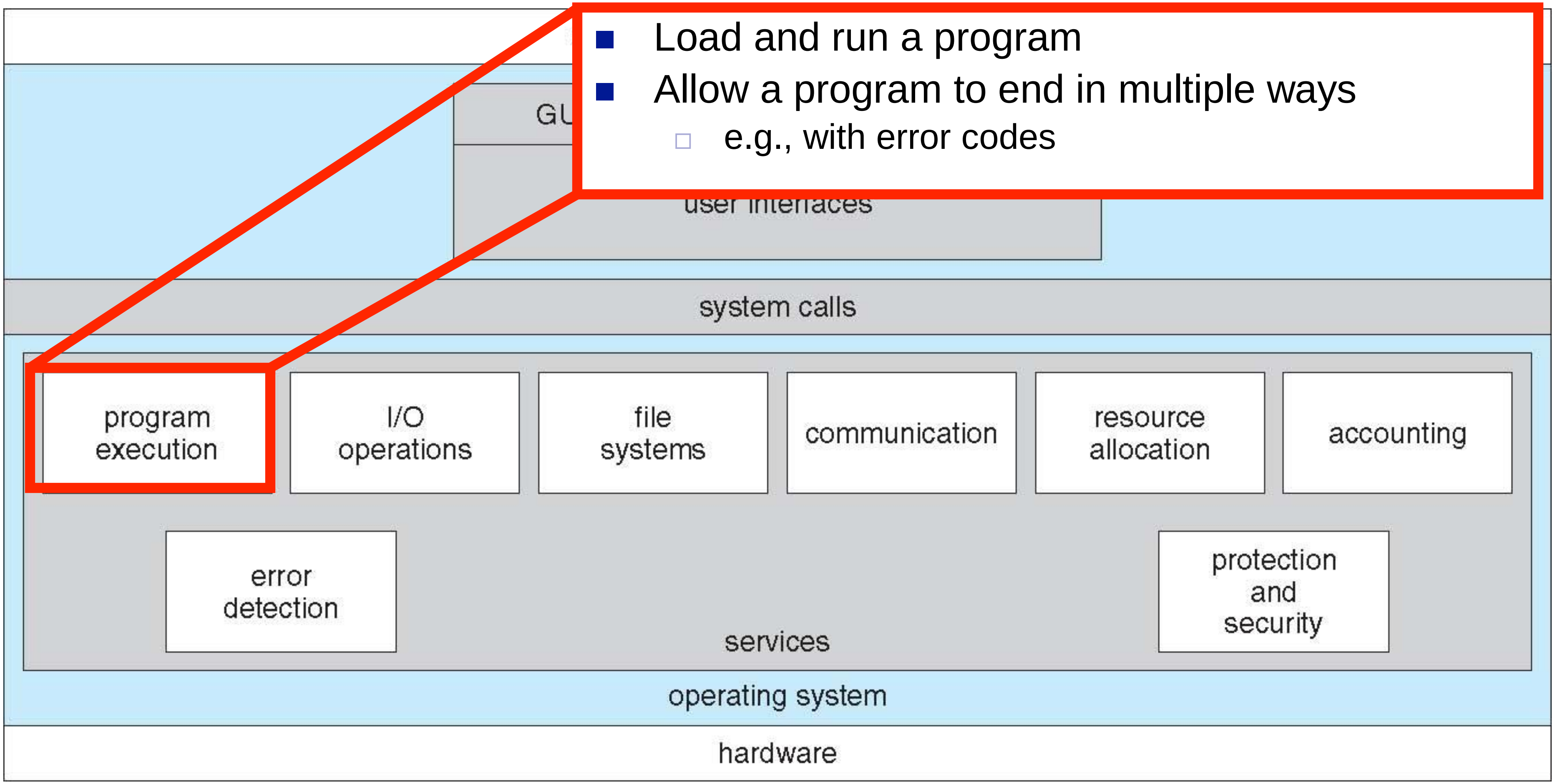


Helpful to users

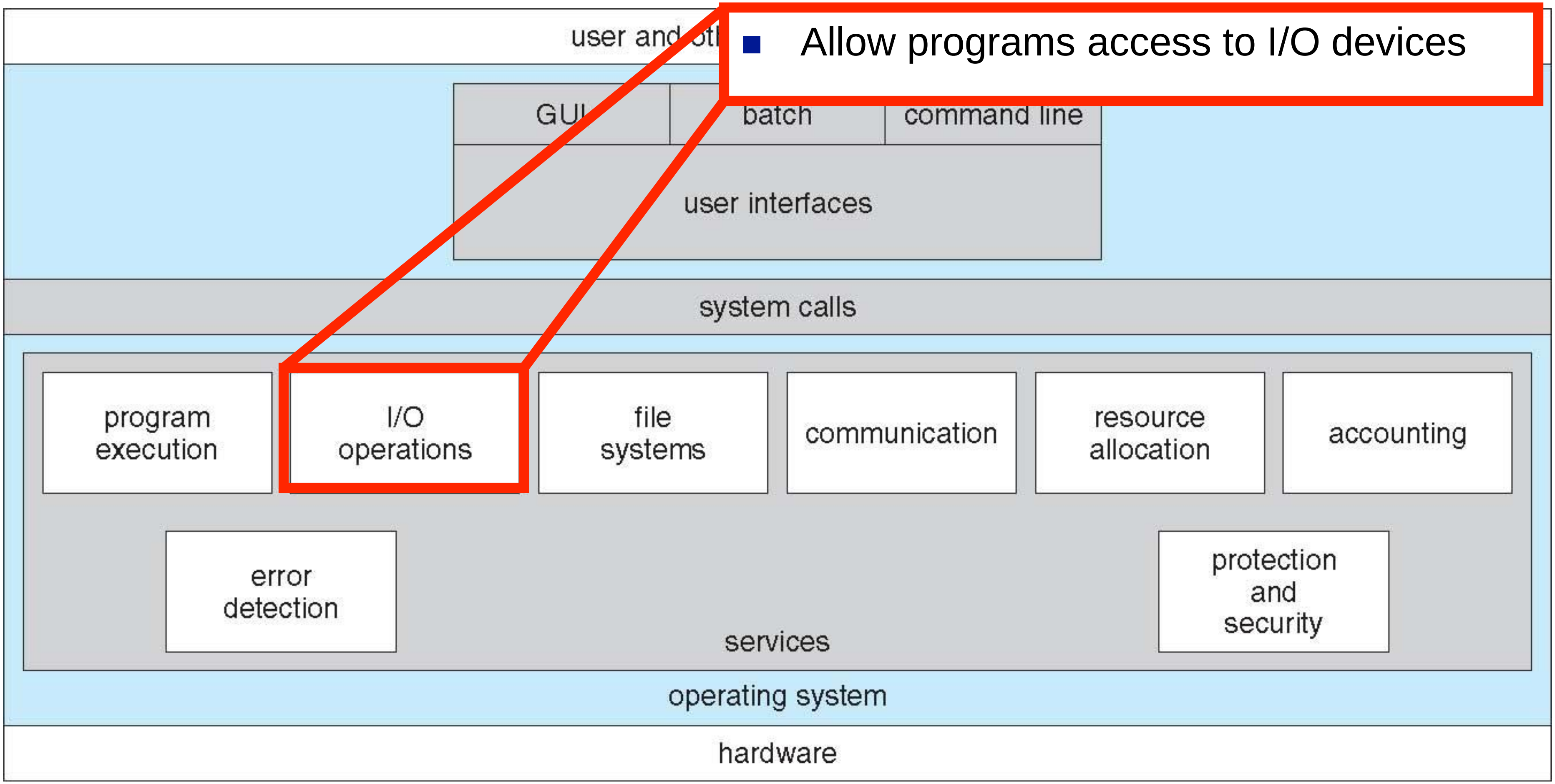
Better efficiency/operation

OS Services and Features

- Load and run a program
- Allow a program to end in multiple ways
  - e.g., with error codes

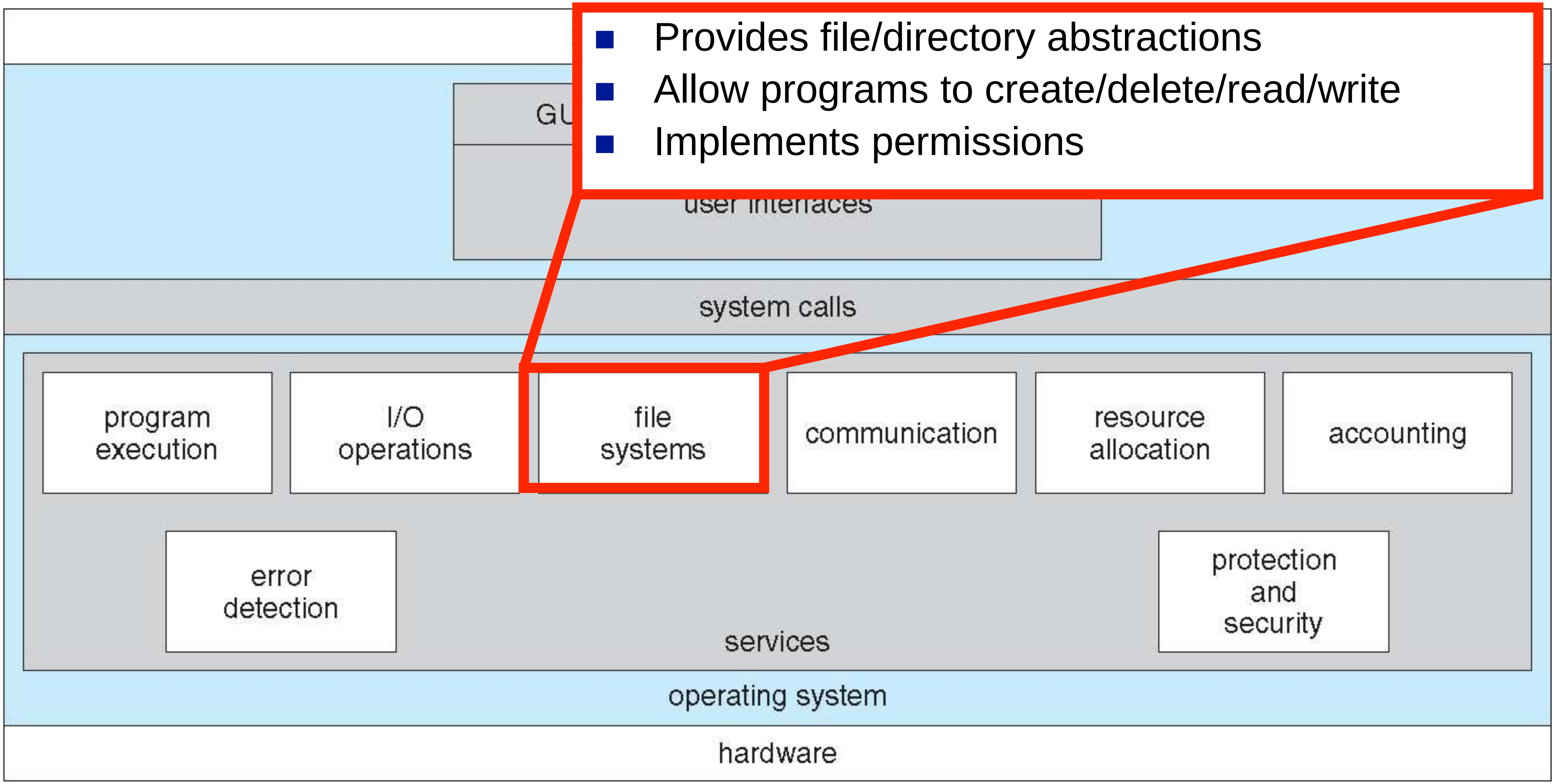


OS Services and Features



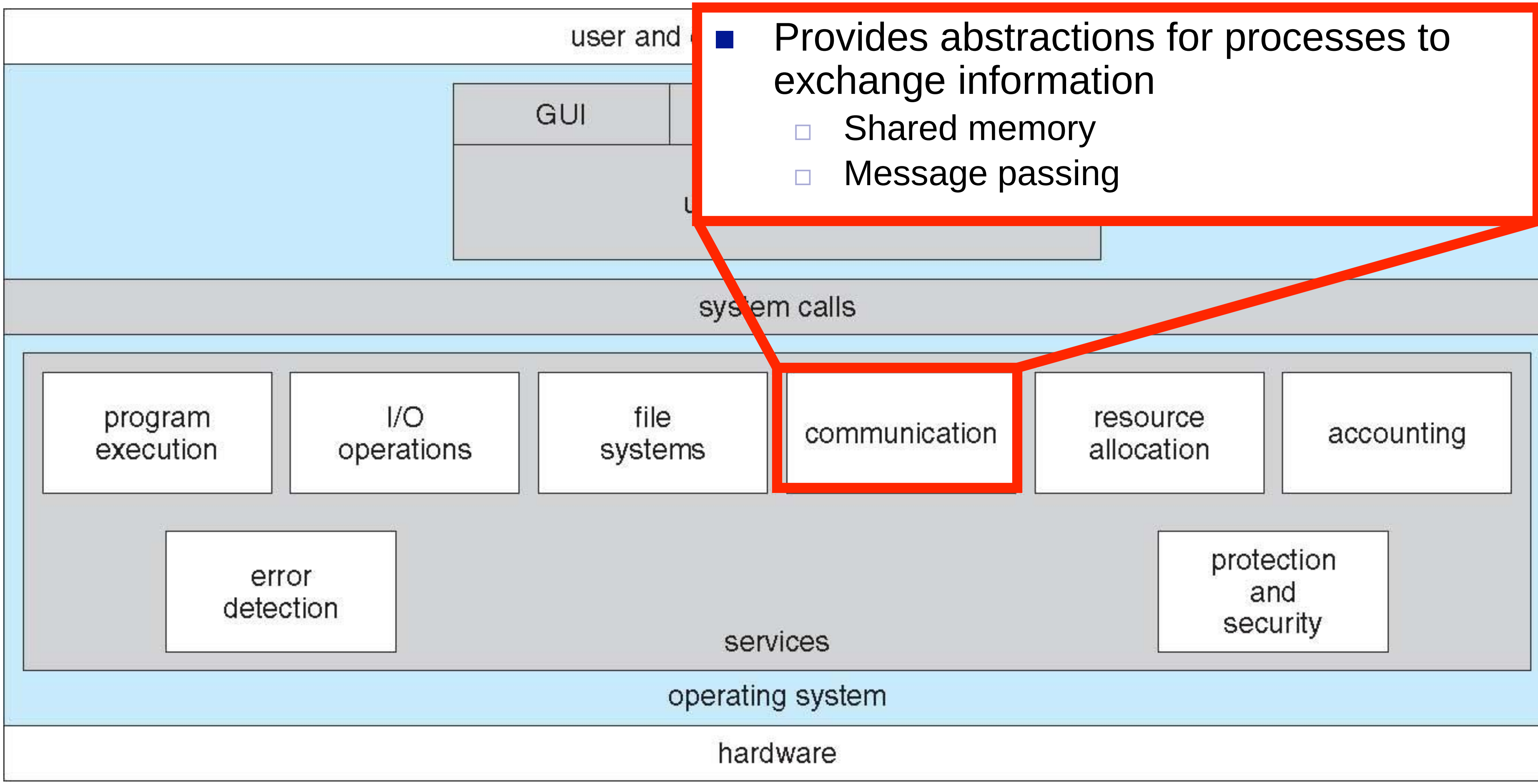
OS Services and Features

- Provides file/directory abstractions
- Allow programs to create/delete/read/write
- Implements permissions

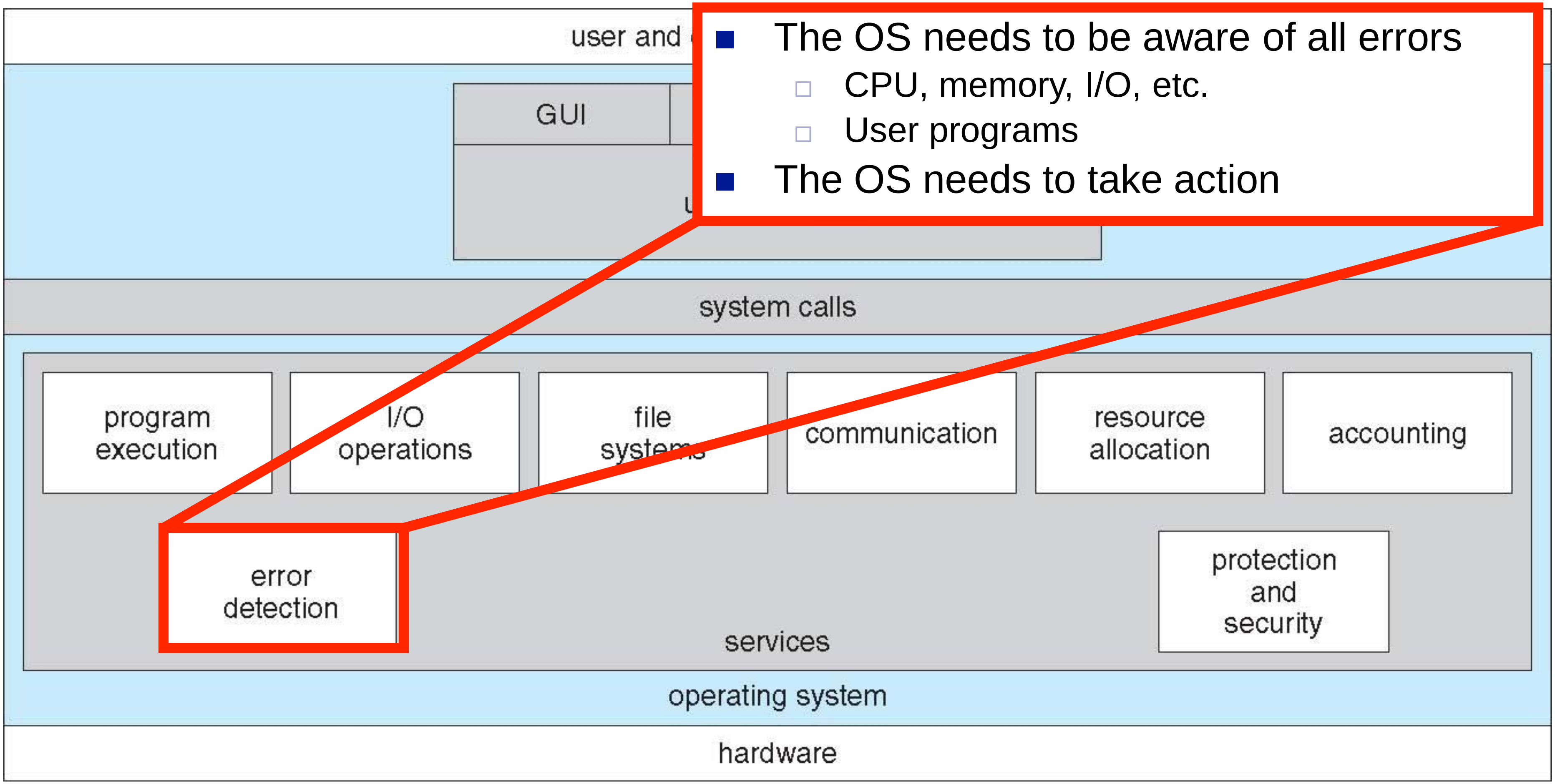


OS Services and Features

- Provides abstractions for processes to exchange information
  - Shared memory
  - Message passing

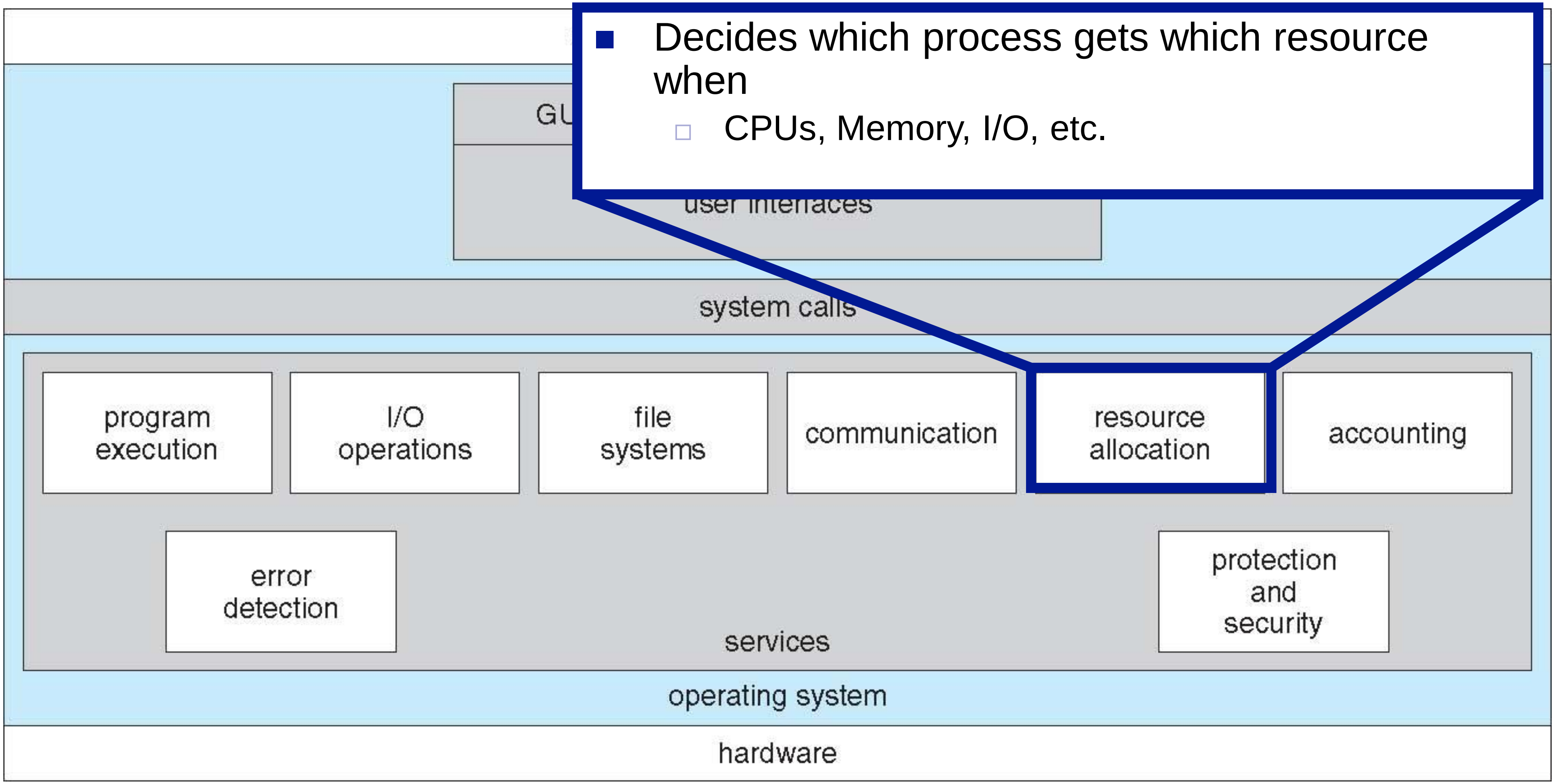


OS Services and Features



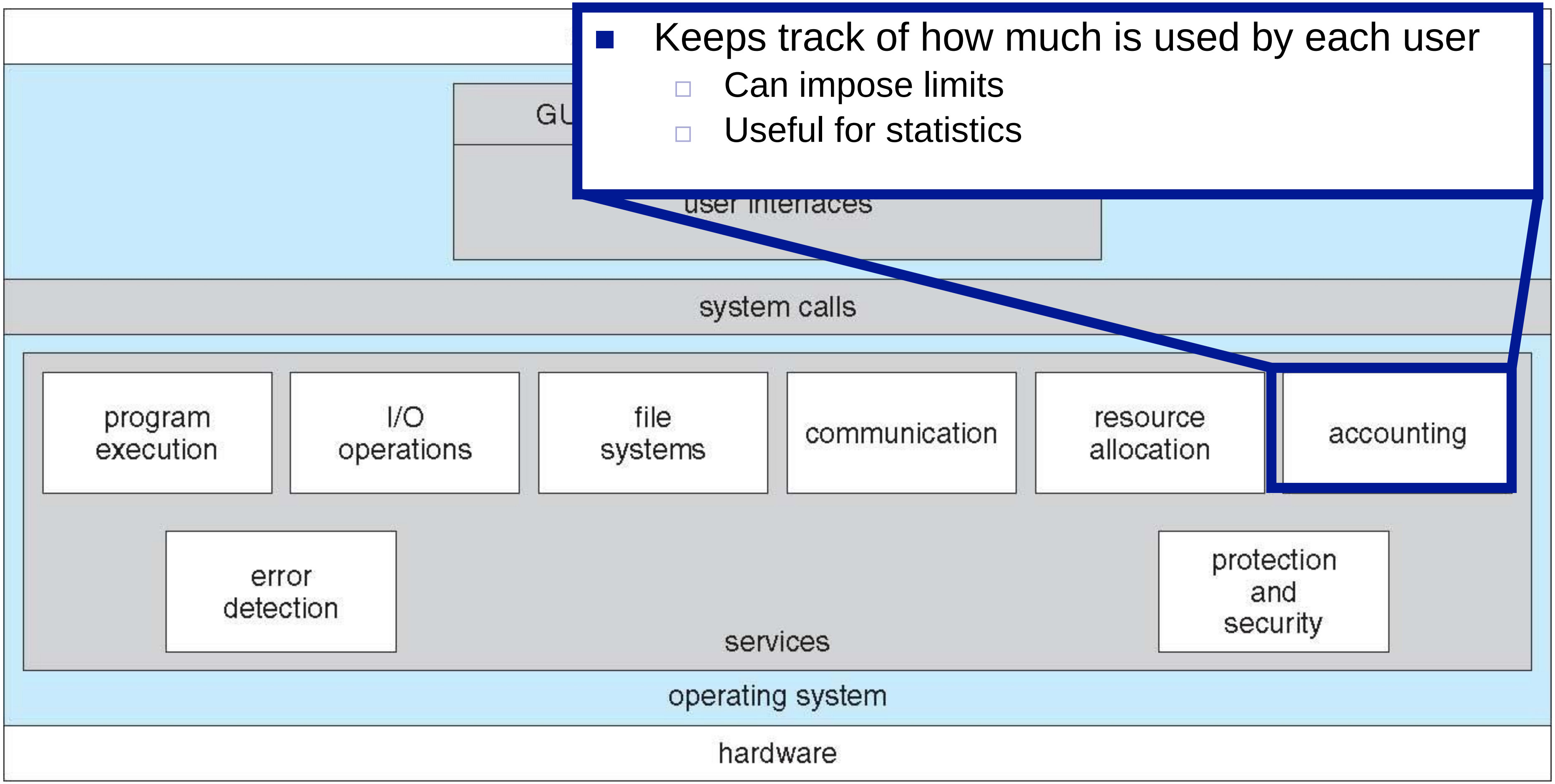
- The OS needs to be aware of all errors
  - CPU, memory, I/O, etc.
  - User programs
- The OS needs to take action

OS Services and Features



- Decides which process gets which resource when
  - CPUs, Memory, I/O, etc.

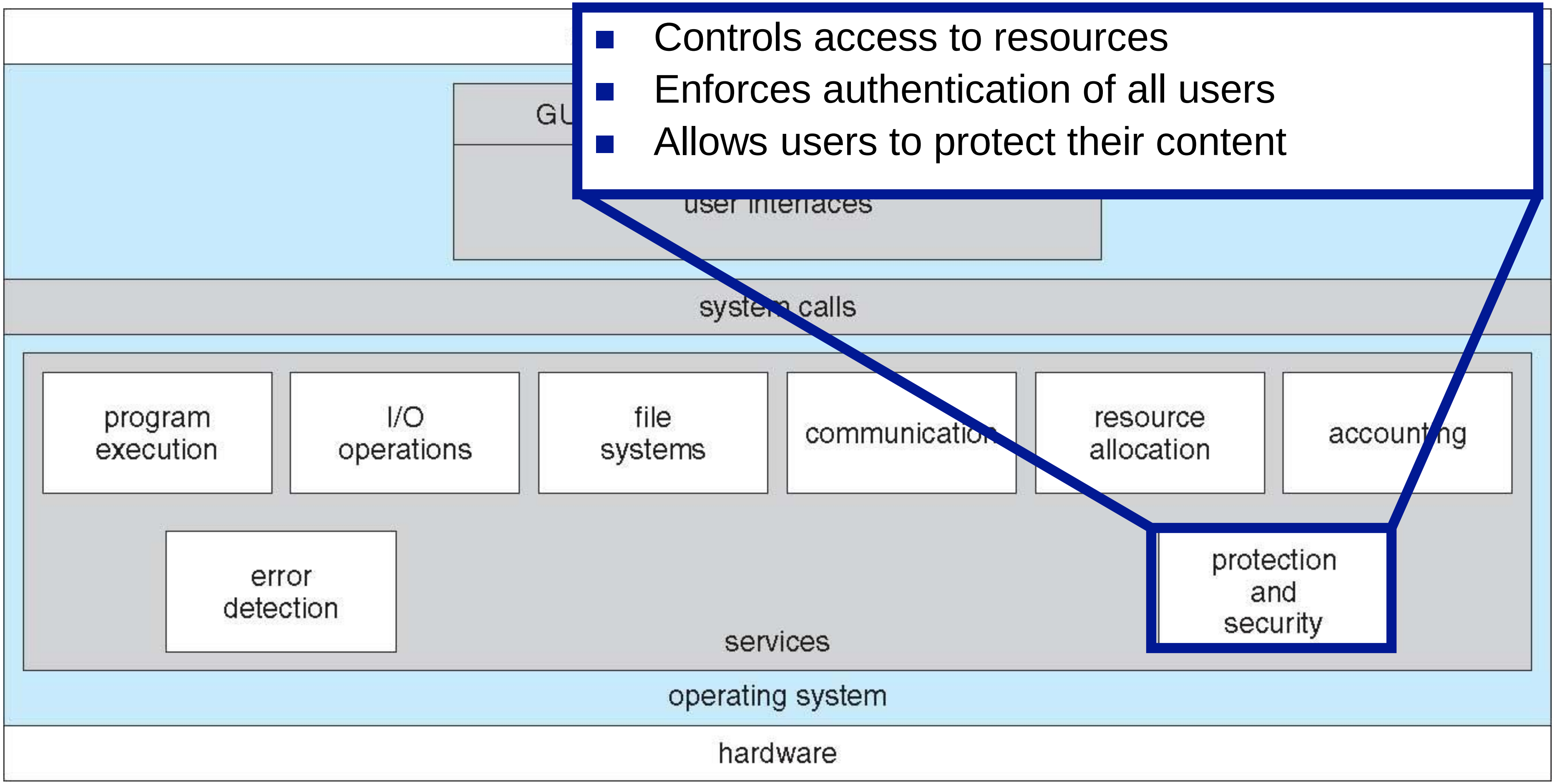
OS Services and Features



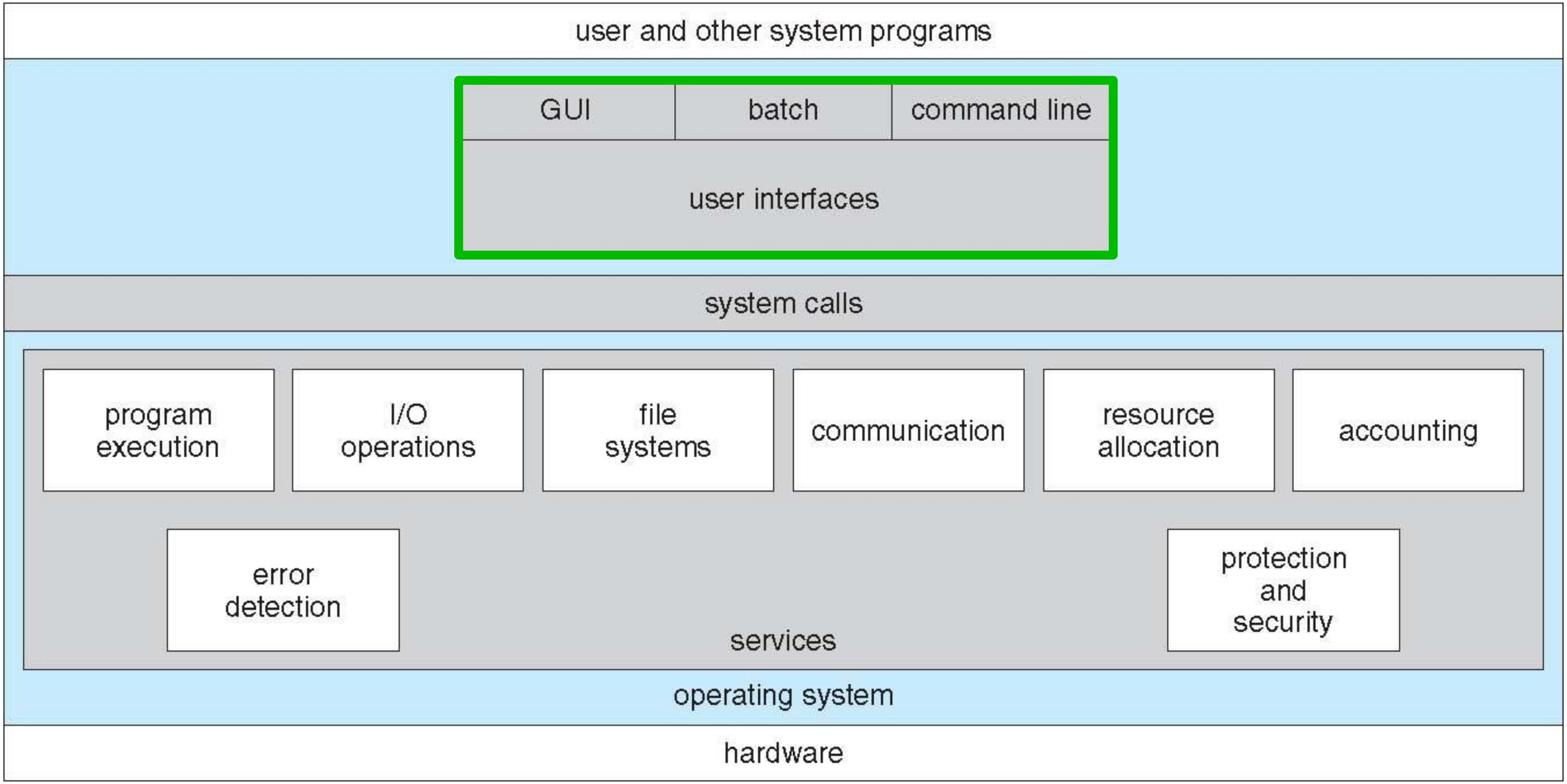
- Keeps track of how much is used by each user
  - Can impose limits
  - Useful for statistics

OS Services and Features

- Controls access to resources
- Enforces authentication of all users
- Allows users to protect their content



OS Services and Features



User Operating System Interface - CLI

CLI or **command interpreter** allows direct command entry

- Sometimes implemented in kernel, sometimes by systems program
- Sometimes multiple flavors implemented – **shells**
- Primarily fetches a command from user and executes it
- Sometimes commands built-in, sometimes just names of programs
  - If the latter, adding new features doesn't require shell modification

Bourne Shell Command Interpreter

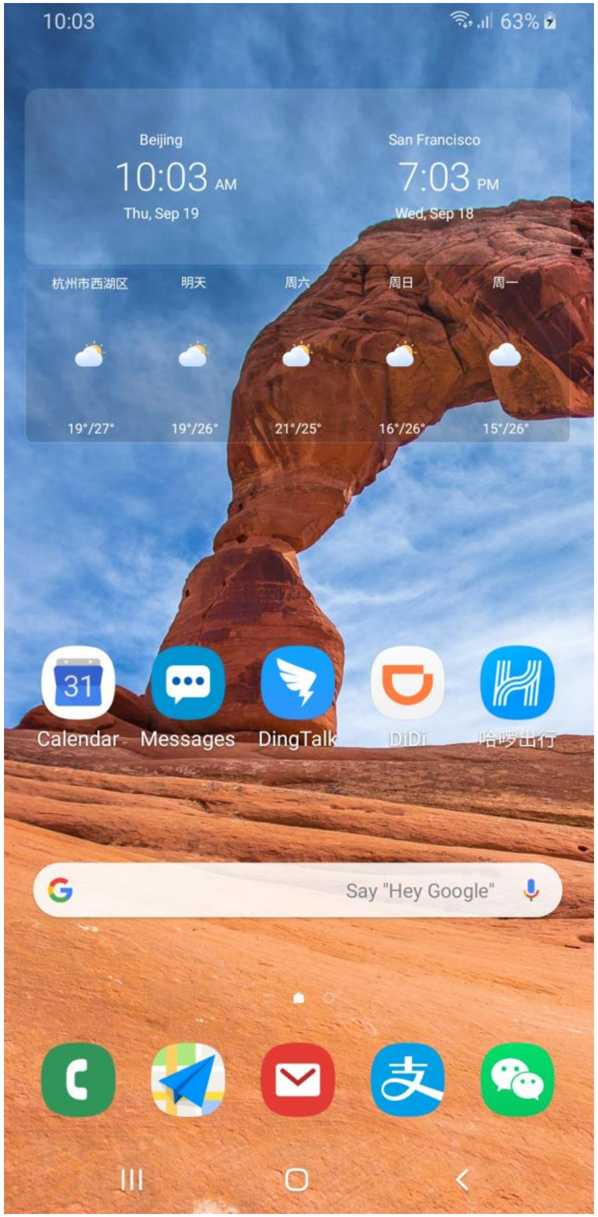
```
wenbo@parallels: ~
wenbo@parallels: ~ 107x30
7ffc75a5f000-7ffc75a80000 rw-p 00000000 00:00 0                          [stack]
7ffc75aa7000-7ffc75aaa000 r--p 00000000 00:00 0                          [vvar]
7ffc75aaa000-7ffc75aac000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                  [vsyscall]
wenbo@parallels:~$ which cat
/bin/cat
wenbo@parallels:~$ file /bin/cat
/bin/cat: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/l,
 for GNU/Linux 3.2.0, BuildID[sha1]=747e524bc20d33ce25ed4aea108e3025e5c3b78f, stripped
wenbo@parallels:~$ cat /proc/self/maps
55b793b79000-55b793b81000 r-xp 00000000 08:01 1048601                    /bin/cat
55b793d80000-55b793d81000 r--p 00007000 08:01 1048601                    /bin/cat
55b793d81000-55b793d82000 rw-p 00008000 08:01 1048601                    /bin/cat
55b794d33000-55b794d54000 rw-p 00000000 00:00 0                          [heap]
7f1974b90000-7f197555f000 r--p 00000000 08:01 662494                     /usr/lib/locale/locale-archive
7f197555f000-7f1975746000 r-xp 00000000 08:01 267596                     /lib/x86_64-linux-gnu/libc-2.27.so
7f1975746000-7f1975946000 ---p 001e7000 08:01 267596                     /lib/x86_64-linux-gnu/libc-2.27.so
7f1975946000-7f197594a000 r--p 001e7000 08:01 267596                     /lib/x86_64-linux-gnu/libc-2.27.so
7f197594a000-7f197594c000 rw-p 001eb000 08:01 267596                     /lib/x86_64-linux-gnu/libc-2.27.so
7f197594c000-7f1975950000 rw-p 00000000 00:00 0
7f1975950000-7f1975977000 r-xp 00000000 08:01 267568                     /lib/x86_64-linux-gnu/ld-2.27.so
7f1975b3c000-7f1975b60000 rw-p 00000000 00:00 0
7f1975b77000-7f1975b78000 r--p 00027000 08:01 267568                     /lib/x86_64-linux-gnu/ld-2.27.so
7f1975b78000-7f1975b79000 rw-p 00028000 08:01 267568                     /lib/x86_64-linux-gnu/ld-2.27.so
7f1975b79000-7f1975b7a000 rw-p 00000000 00:00 0
7ffc73010000-7ffc73031000 rw-p 00000000 00:00 0                          [stack]
7ffc73148000-7ffc7314b000 r--p 00000000 00:00 0                          [vvar]
7ffc7314b000-7ffc7314d000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                  [vsyscall]
wenbo@parallels:~$
```

User Operating System Interface - GUI

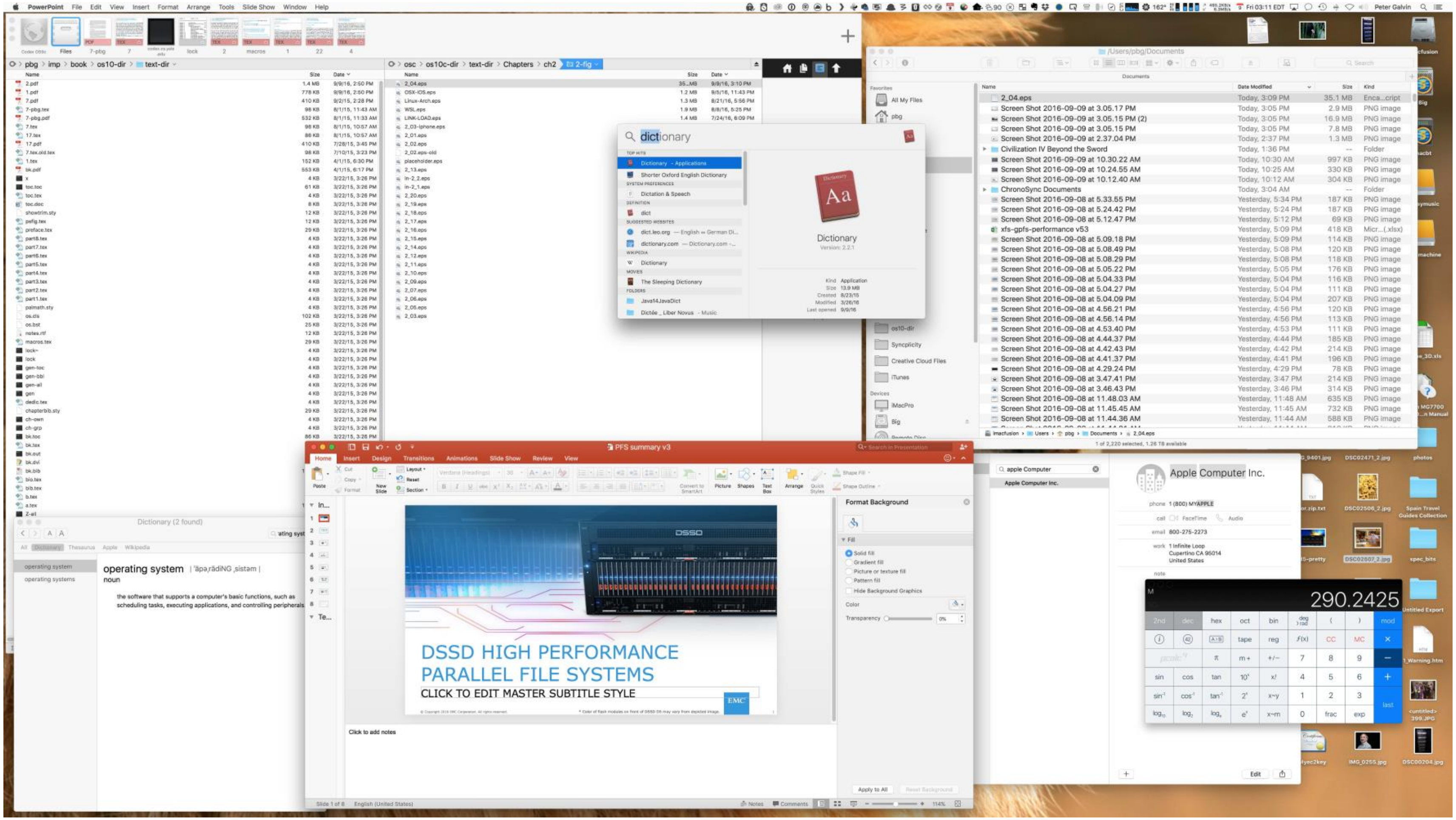
- User-friendly **desktop** metaphor interface
  - Usually mouse, keyboard, and monitor
  - **Icons** represent files, programs, actions, etc
  - Various mouse buttons over objects in the interface cause various actions (provide information, options, execute function, open directory (known as a **folder**)
  - Invented at Xerox PARC
- Many systems now 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)

Touchscreen Interfaces

- Touchscreen devices require new interfaces
  - Mouse not possible or not desired
  - Actions and selection based on gestures
  - Virtual keyboard for text entry
- Voice commands



The Mac OS X GUI

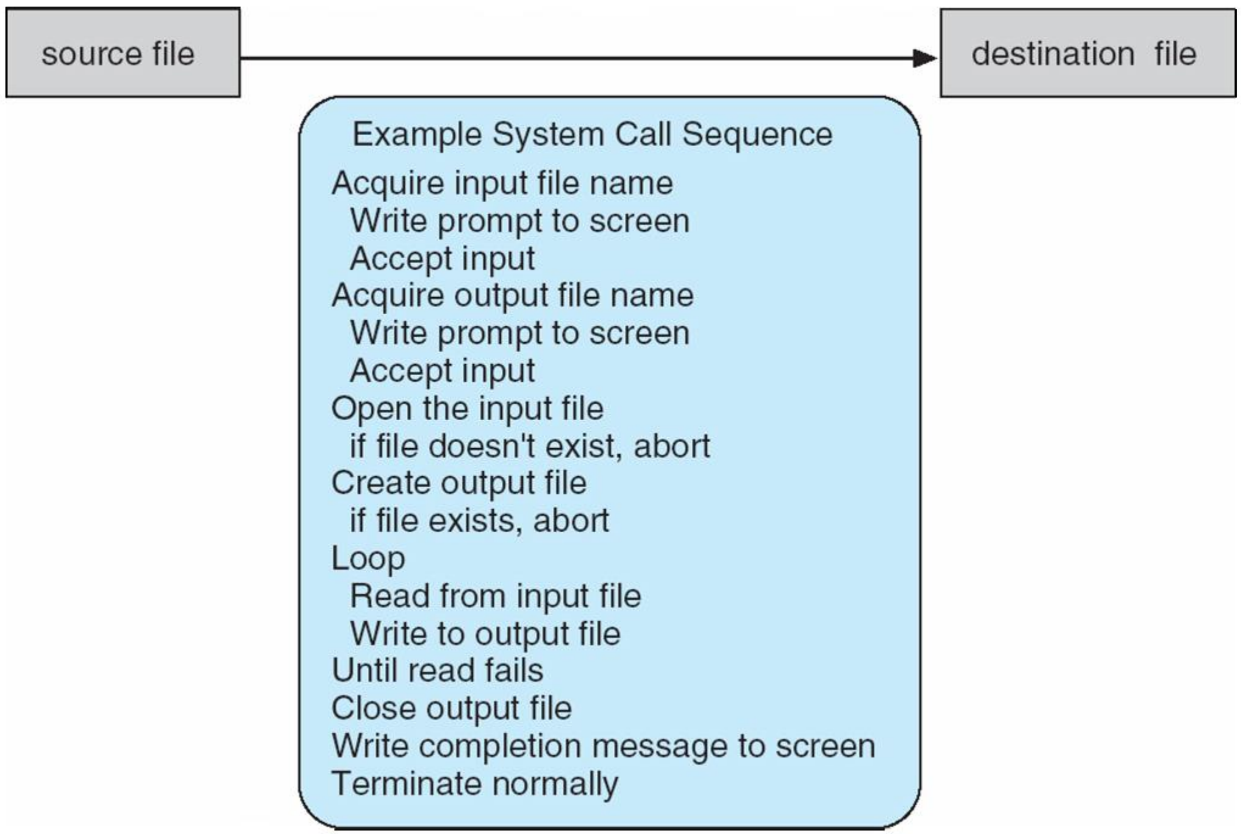


System Calls

- Programming interface to the services provided by the OS
- 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
- Three most common APIs:
  - Win32 API for Windows
  - POSIX API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X)
  - Java API for the Java virtual machine (JVM)

Example of System Calls

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



System Calls

On Linux there is a “command” called strace that gives details about which system calls were placed by a program during execution

- dtruss on Mac OSX is roughly equivalent

Let’s look at what it shows us when I copy a large file with the cp command on my Linux server

- strace -xf cp <some large file> bogus
- Let’s count the number of system calls using the wc command
- Let’s try with a tiny file and compare
- Let’s look at the system calls and see if they make sense
- Let’s try very simple commands and see...

Conclusion: there are TONS of system calls

strace can be “attached” to a running program!

- to find out, e.g., why a program is stuck!

Example of 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>

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 into which the data will be read
- `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.

Time Spent in System Calls?

- The time command is a simple way to time the execution of a program
  - Not great precision/resolution, but fine for getting a rough idea
- Time is used just like strace: place it in front of the command you want to time
- It reports three times:
  - “real” time: wall-clock time (also called elapsed time, execution time, run time, etc.)
  - “user” time: time spent in user code (user mode)
  - “system” time”: time spent in system calls (kernel mode)
- Let’s try it

System Call Implementation

Typically, a number associated with each system call

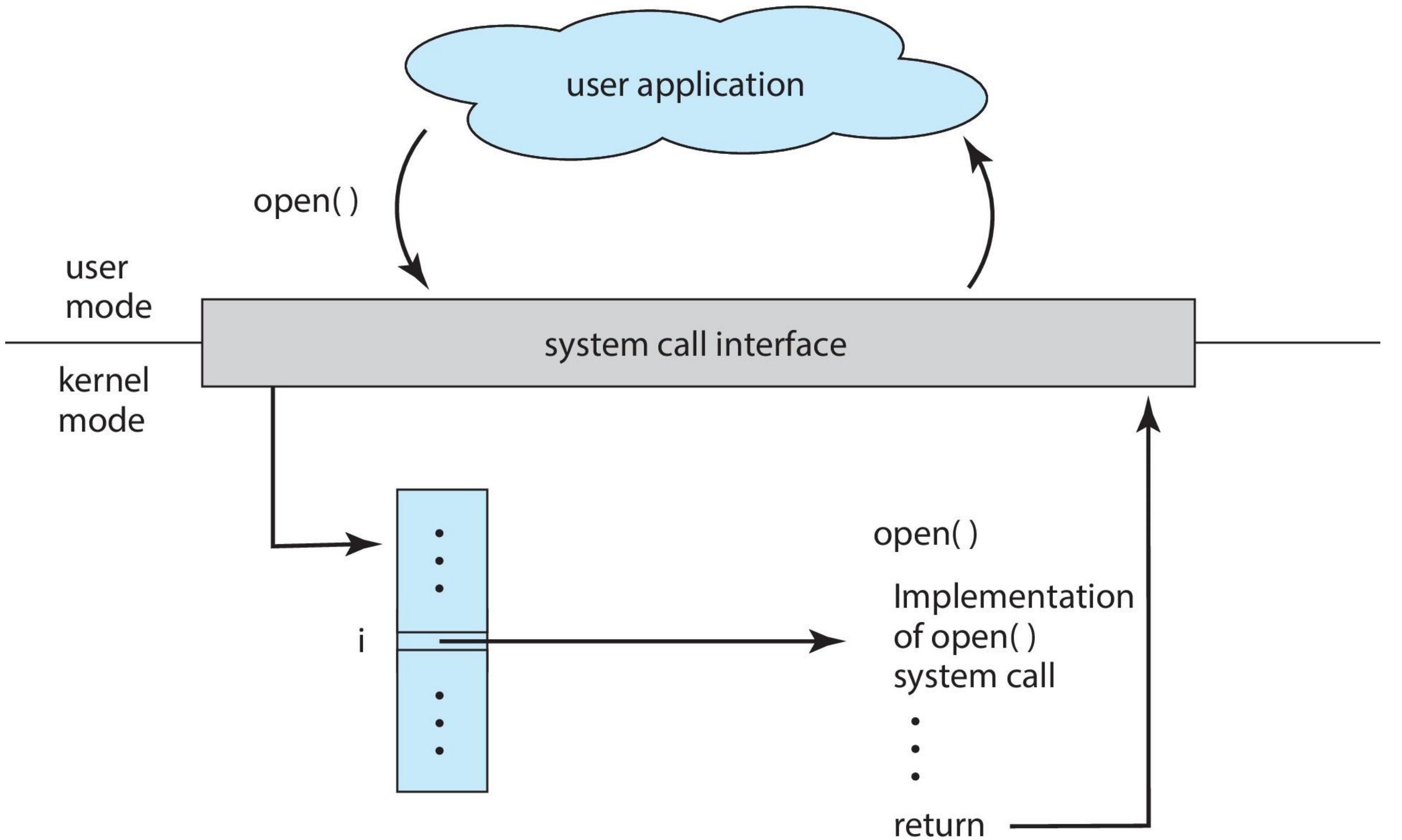
- **System-call interface** maintains a table indexed according to these numbers

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)

API – System Call – OS Relationship



The Hidden Syscall Table

Let's look a bit inside the Linux Kernel

Linux kernel v5.3, ARM64

arch/arm64/kernel/sys.c

```
#undef __SYSCALL
#define __SYSCALL(nr, sym)      [nr] = (syscall_fn_t)__arm64_##sym,

const syscall_fn_t sys_call_table[__NR_syscalls] = {
        [0 ... __NR_syscalls - 1] = (syscall_fn_t)sys_ni_syscall,
#include <asm/unistd.h>
};
```

```
#define __NR_setxattr 5
__SYSCALL(__NR_setxattr, sys_setxattr)
#define __NR_lsetxattr 6
__SYSCALL(__NR_lsetxattr, sys_lsetxattr)
#define __NR_fsetxattr 7
__SYSCALL(__NR_fsetxattr, sys_fsetxattr)
#define __NR_getxattr 8
__SYSCALL(__NR_getxattr, sys_getxattr)
#define __NR_lgetxattr 9
__SYSCALL(__NR_lgetxattr, sys_lgetxattr)
#define __NR_fgetxattr 10
__SYSCALL(__NR_fgetxattr, sys_fgetxattr)
#define __NR_listxattr 11
__SYSCALL(__NR_listxattr, sys_listxattr)
```

The Hidden Syscall Table

Let’s look a bit inside the Linux Kernel

Linux kernel v5.3, ARM64

arch/arm64/kernel/sys.i

```
const syscall_fn_t sys_call_table[294] = {
[0] = (syscall_fn_t)__arm64_sys_io_setup,
[1] = (syscall_fn_t)__arm64_sys_io_destroy,
[2] = (syscall_fn_t)__arm64_sys_io_submit,
[3] = (syscall_fn_t)__arm64_sys_io_cancel,
[4] = (syscall_fn_t)__arm64_sys_io_getevents,
[5] = (syscall_fn_t)__arm64_sys_setxattr,
[6] = (syscall_fn_t)__arm64_sys_lsetxattr,
[7] = (syscall_fn_t)__arm64_sys_fsetxattr,
[8] = (syscall_fn_t)__arm64_sys_getxattr,
[9] = (syscall_fn_t)__arm64_sys_lgetxattr,
[10] = (syscall_fn_t)__arm64_sys_fgetxattr,
[11] = (syscall_fn_t)__arm64_sys_listxattr,
[12] = (syscall_fn_t)__arm64_sys_llistxattr,
[13] = (syscall_fn_t)__arm64_sys_flistxattr,
[14] = (syscall_fn_t)__arm64_sys_removexattr,
[15] = (syscall_fn_t)__arm64_sys_lremovexattr,
[16] = (syscall_fn_t)__arm64_sys_fremovexattr,
[17] = (syscall_fn_t)__arm64_sys_getcwd,
[18] = (syscall_fn_t)__arm64_sys_lookup_dcookie,
[19] = (syscall_fn_t)__arm64_sys_eventfd2,
```

The Hidden Syscall Table

Let’s look a bit inside the Linux Kernel

Linux kernel v5.3, ARM64

In Homework 1, also find system call table for:

- ARM32
- RISC-V(32 bit)
- RISC-V(64 bit)
- X86(32 bit)
- X86_64

Details on c.zju.edu.cn Homework 1

System Call Parameter Passing

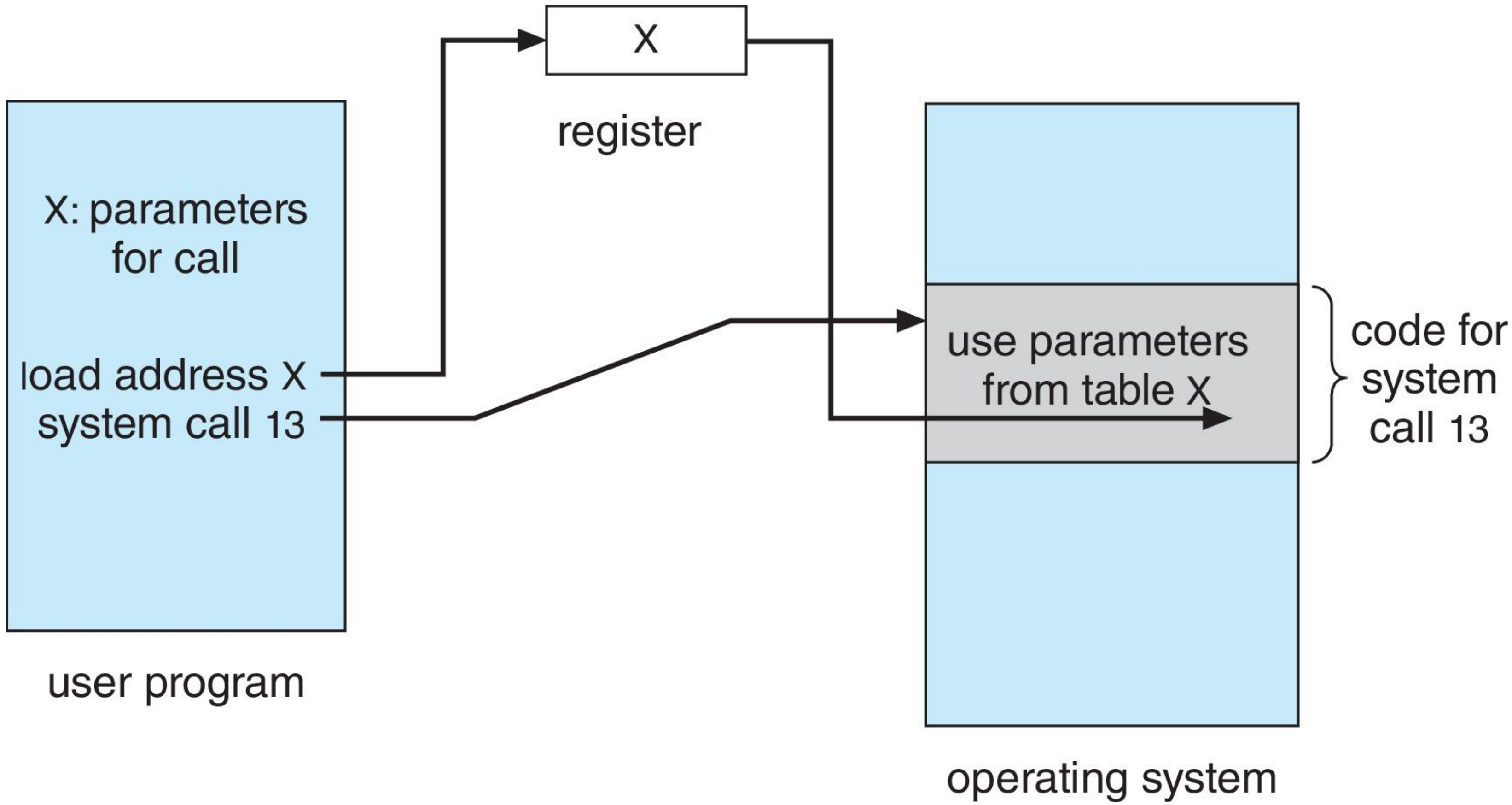
Often, more information is required than simply identity of desired system call

- Exact type and amount of information vary according to OS and call

Three general methods used to pass parameters to the OS

- Simplest: pass the parameters in registers
  - In some cases, may be more parameters than registers
- 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

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()`
	`ExitProcess()`
	`WaitForSingleObject()`
		`fork()`
		`exit()`
		`wait()`

File management	`CreateFile()`
	`ReadFile()`
	`WriteFile()`
	`CloseHandle()`
		`open()`
		`read()`
		`write()`
		`close()`

Device management	`SetConsoleMode()`
	`ReadConsole()`
	`WriteConsole()`
		`ioctl()`
		`read()`
		`write()`

Information maintenance	`GetCurrentProcessID()`
	`SetTimer()`
	`Sleep()`
		`getpid()`
		`alarm()`
		`sleep()`

Communications	`CreatePipe()`
	`CreateFileMapping()`
	`MapViewOfFile()`
		`pipe()`
		`shm_open()`
		`mmap()`

Protection	`SetFileSecurity()`
	`InitlializeSecurityDescriptor()`
	`SetSecurityDescriptorGroup()`
		`chmod()`
		`umask()`
		`chown()`

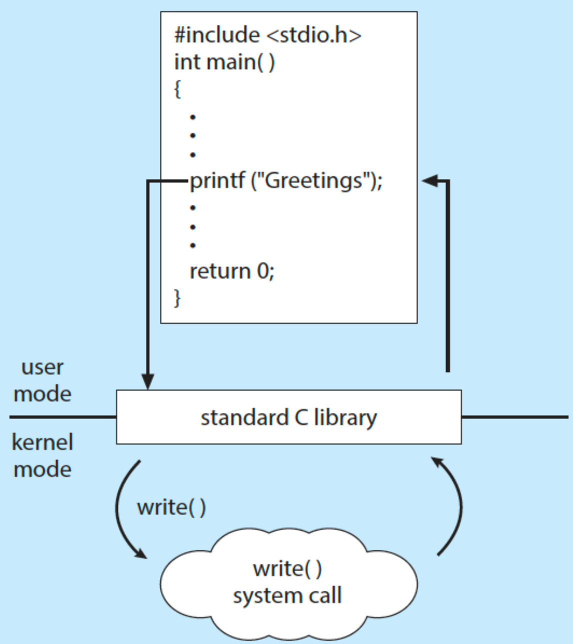
Standard C Library Example

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

https://code.woboq.org/userspace/glibc/stdio-common/printf.c.html

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:



System Services

System programs provide a convenient environment for program development and execution. They can be divided into:

- File manipulation
- Status information sometimes stored in a file
- Programming language support
- Program loading and execution
- Communications
- Background services
- Application programs

Most users' view of the operation system is defined by system programs, not the actual system calls

System Services (cont.)

Provide a convenient environment for program development and 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

- Some ask the system for info - date, time, amount of available memory, disk space, number of users
- Others provide detailed performance, logging, and debugging information
- Typically, these programs format and print the output to the terminal or other output devices
- Some systems implement a **registry** - used to store and retrieve configuration information

System Services (Cont.)

File modification

- Text editors to create and modify files
- Special commands to search contents of files or perform transformations of the text

Programming-language support - Compilers, assemblers, debuggers and interpreters sometimes provided

Program loading and execution- Absolute loaders, relocatable loaders, linkage editors, and overlay-loaders, debugging systems for higher-level and machine language

Communications - Provide the mechanism for creating virtual connections among processes, users, and computer systems

- Allow users to send messages to one another's screens, browse web pages, send electronic-mail messages, log in remotely, transfer files from one machine to another

System Services (Cont.)

- **Background Services**
  - Launch at boot time
    - Some for system startup, then terminate
    - Some from system boot to shutdown
  - Provide facilities like disk checking, process scheduling, error logging, printing
  - Run in user context not kernel context
  - Known as **services**, **subsystems**, **daemons**
- **Application programs**
  - Don’t pertain to system
  - Run by users
  - Not typically considered part of OS
  - Launched by command line, mouse click, finger poke

Linkers and Loaders

Source code compiled into object files designed to be loaded into any physical memory location – **relocatable object file**

Linker combines these into single binary **executable** file

- Also brings in libraries

Program resides on secondary storage as binary executable

Must be brought into memory by **loader** to be executed

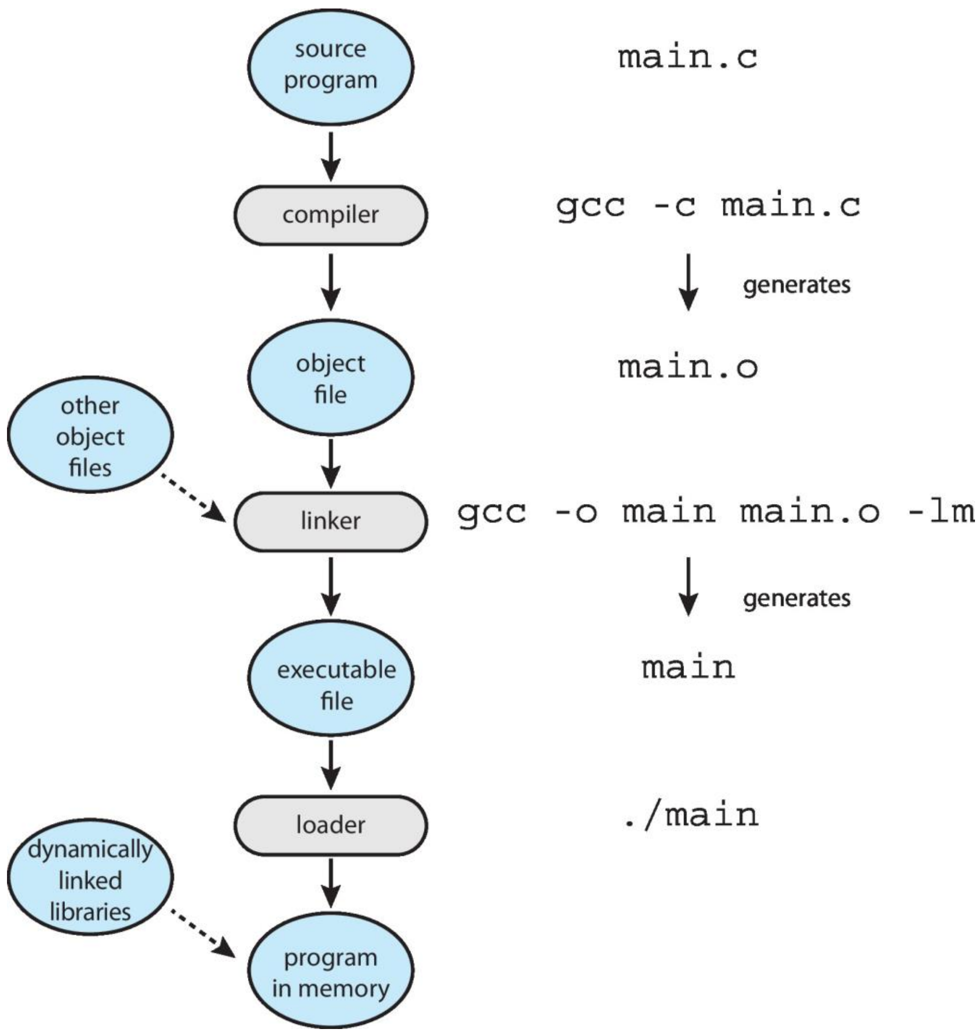
- **Relocation** assigns final addresses to program parts and adjusts code and data in program to match those addresses

Modern general purpose systems don't link libraries into executables

- Rather, **dynamically linked libraries** (in Windows, **DLLs**) are loaded as needed, shared by all that use the same version of that same library (loaded once)

Object, executable files have standard formats, so operating system knows how to load and start them

The Role of the Linker and Loader



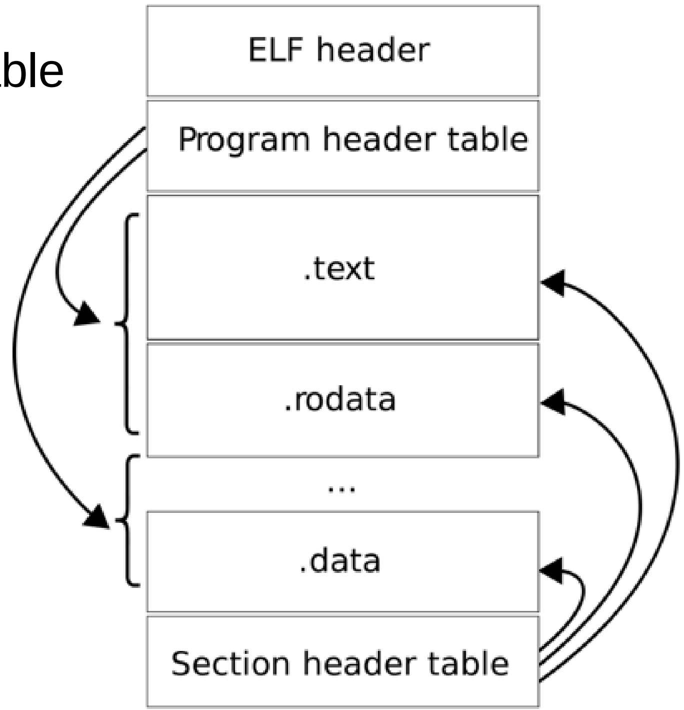
ELF binary basics

What's main（a.out）

- Executable and Linkable Format - ELF
- Program header table and section header table
  - For Linker and Loader
- .text: code
- .rodata: initialized read-only data
- .data: initialized data
- .bss: uninitialized data

Quiz

- Where does static variable go?
- Static const?



ELF binary basics

Dump all sections

readelf -S a.out

```
Wenbos-MacBook-Pro:c-hello wenbo$ readelf -S a.out
There are 31 section headers, starting at offset 0x1708:

Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
  [ 0]                   NULL             0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .interp           PROGBITS         0000000000000238  00000238
       000000000000001c  0000000000000000   A       0     0     1
  [ 2] .note.ABI-tag     NOTE             0000000000000254  00000254
       0000000000000020  0000000000000000   A       0     0     4

  [13] .text             PROGBITS         0000000000000550  00000550
       00000000000001a2  0000000000000000  AX       0     0     16
  [14] .fini             PROGBITS         00000000000006f4  000006f4
       0000000000000009  0000000000000000  AX       0     0     4
  [15] .rodata           PROGBITS         0000000000000700  00000700
       0000000000000010  0000000000000000   A       0     0     4

  [23] .data             PROGBITS         0000000000002000  00001000
       0000000000000010  0000000000000000  WA       0     0     8
  [24] .tm_clone_table   PROGBITS         0000000000002010  00001010
       0000000000000000  0000000000000000  WA       0     0     8
  [25] .bss              NOBITS           0000000000002010  00001010
       0000000000000001  0000000000000000  WA       0     0     1
```

ELF binary basics

Read header

readelf -h a.out

```
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              DYN (Shared object file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x550
  Start of program headers:          64 (bytes into file)
  Start of section headers:          5896 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         9
  Size of section headers:           64 (bytes)
  Number of section headers:         31
  Section header string table index: 30
```

ELF binary basics

Run a program

```
wenbo@wenbo-ThinkPad:~$ cat /proc/self/maps
559a61b8d000-559a61b95000 r-xp 00000000 08:05 3145753                    /bin/cat
559a61d94000-559a61d95000 r--p 00007000 08:05 3145753                    /bin/cat
559a61d95000-559a61d96000 rw-p 00008000 08:05 3145753                    /bin/cat
559a63860000-559a63881000 rw-p 00000000 00:00 0                          [heap]
7f8fc2690000-7f8fc305f000 r--p 00000000 08:05 4725339                    /usr/lib/locale/locale-archive
7f8fc305f000-7f8fc3246000 r-xp 00000000 08:05 2626368                    /lib/x86_64-linux-gnu/libc-2.27.so
7f8fc3246000-7f8fc3446000 ---p 001e7000 08:05 2626368                    /lib/x86_64-linux-gnu/libc-2.27.so
7f8fc3446000-7f8fc344a000 r--p 001e7000 08:05 2626368                    /lib/x86_64-linux-gnu/libc-2.27.so
7f8fc344a000-7f8fc344c000 rw-p 001eb000 08:05 2626368                    /lib/x86_64-linux-gnu/libc-2.27.so
7f8fc344c000-7f8fc3450000 rw-p 00000000 00:00 0
7f8fc3450000-7f8fc3477000 r-xp 00000000 08:05 2626340                    /lib/x86_64-linux-gnu/ld-2.27.so
7f8fc3635000-7f8fc3659000 rw-p 00000000 00:00 0
7f8fc3677000-7f8fc3678000 r--p 00027000 08:05 2626340                    /lib/x86_64-linux-gnu/ld-2.27.so
7f8fc3678000-7f8fc3679000 rw-p 00028000 08:05 2626340                    /lib/x86_64-linux-gnu/ld-2.27.so
7f8fc3679000-7f8fc367a000 rw-p 00000000 00:00 0
7ffd416e0000-7ffd41701000 rw-p 00000000 00:00 0                          [stack]
7ffd4178b000-7ffd4178e000 r--p 00000000 00:00 0                          [vvar]
7ffd4178e000-7ffd41790000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                  [vsyscall]
```

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
  - Written in interpreted language like Python, Ruby, and interpreter available on multiple operating systems
  - App written in language that includes a VM containing the running app (like Java)
  - Use standard language (like C), compile separately on each operating system to run on each
- **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 (Cont.)

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

- A very important principle
- Allows policy changes without changed implemented mechanism

Door Example

- Entrance policy with regular door lock (mechanism) BAD
- Smart door lock (mechanism) GOOD

Scheduling

- Scheduling policy and how to pick the next (mechanism)

Implementation

Much variation

- Early OSes in assembly language
- Then system programming languages like Algol, PL/1
- Now C, C++

Actually usually a mix of languages

- Lowest levels in assembly
- Main body in C
- Systems programs in C, C++, scripting languages like PERL, Python, shell scripts

More high-level language easier to **port** to other hardware

- But slower

Emulation can allow an OS to run on non-native hardware

Operating System Structure

General-purpose OS is very large program

Various ways to structure ones

- Simple structure – MS-DOS
- More complex -- UNIX
- Layered – an abstrcation
- Microkernel -Mach

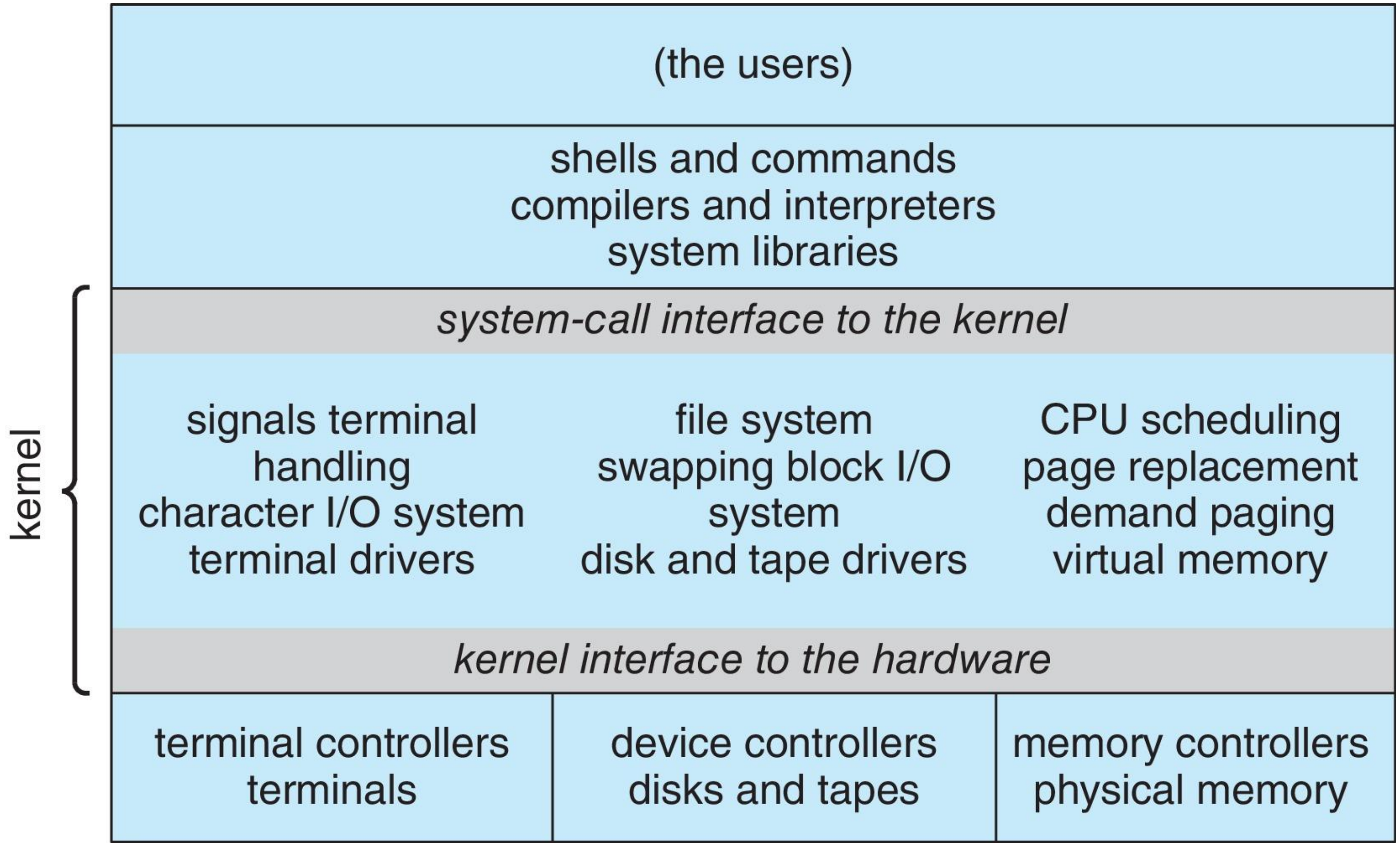
Monolithic Structure – Original UNIX

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

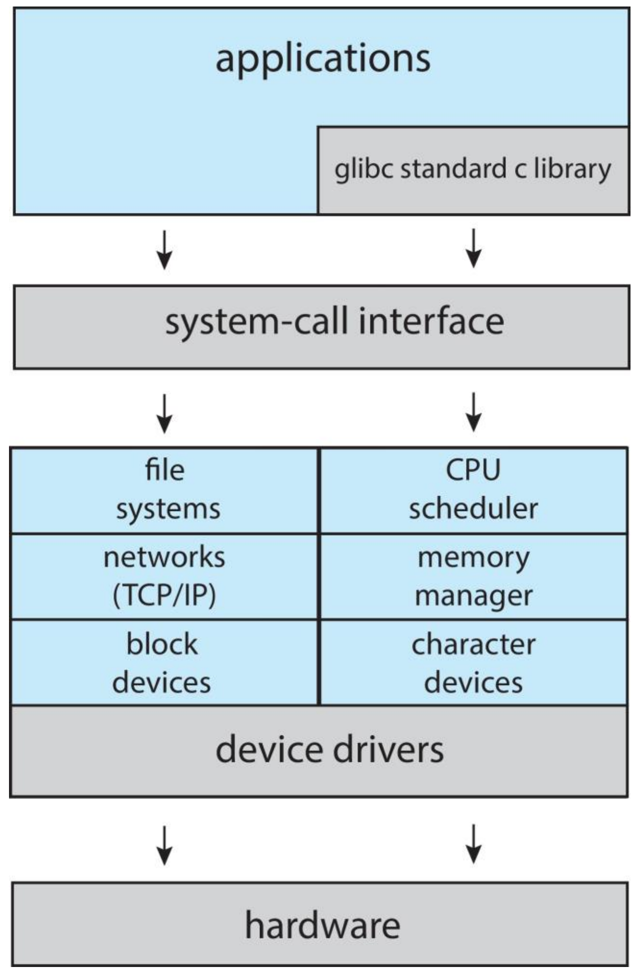
Traditional UNIX System Structure

Beyond simple but not fully layered

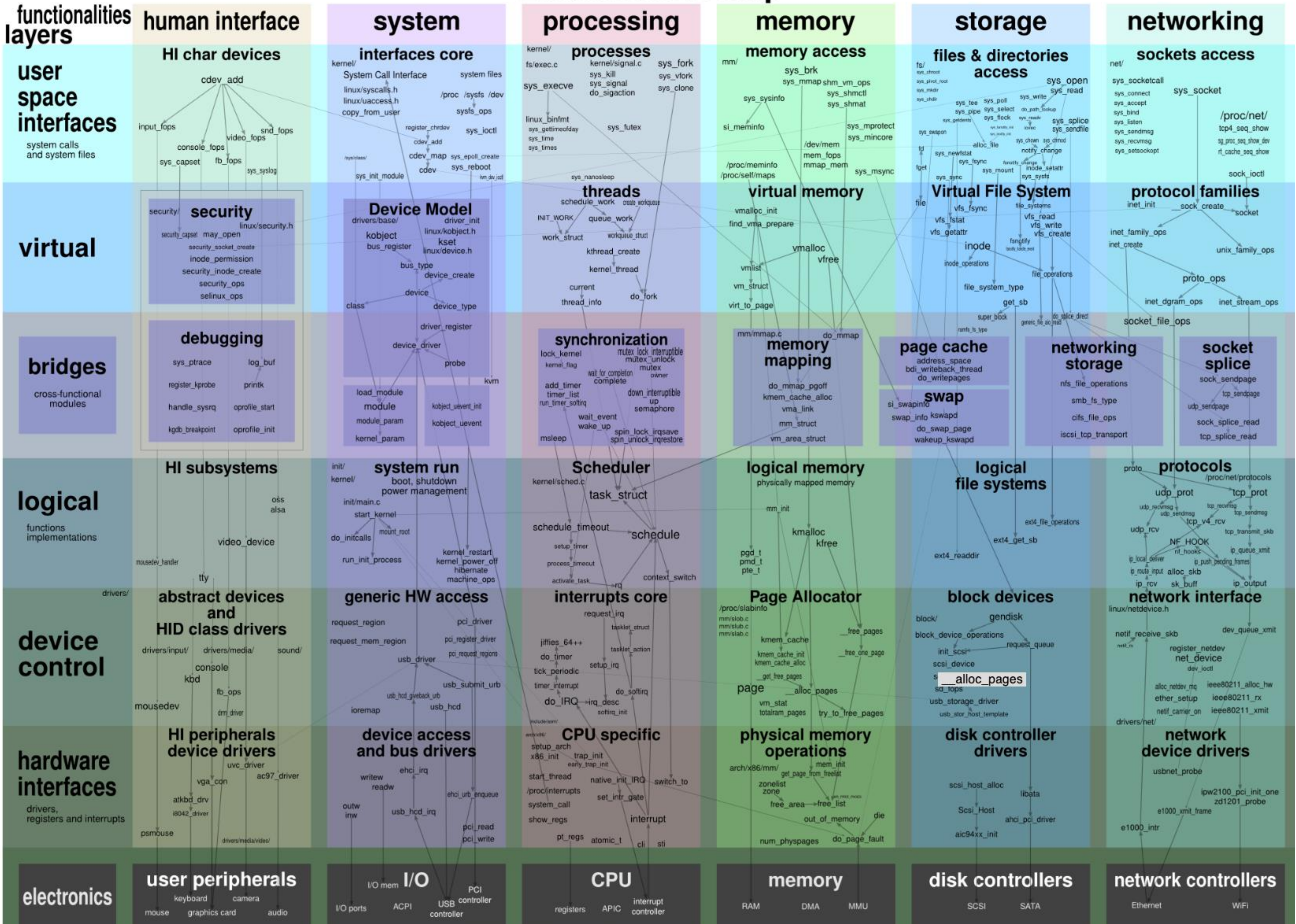


Linux System Structure

Monolithic supports loadable kernel module



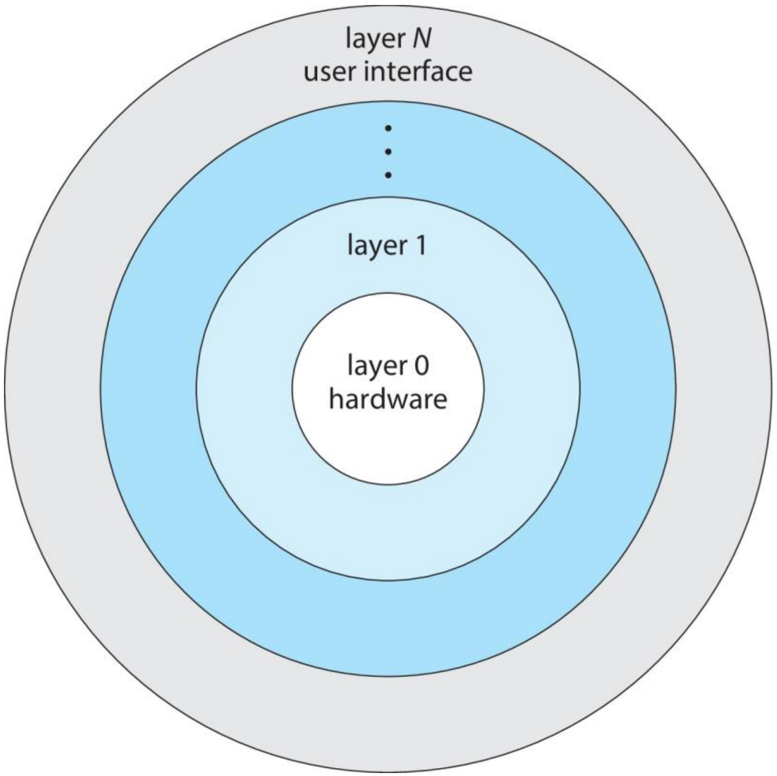
Linux kernel map



Layered 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 only lower-level layers



Microkernels

Moves as much from the kernel into user space

Mach example of **microkernel**

- Mac OS X kernel (**Darwin**) partly based on Mach

Communication takes place between user modules using **message passing**

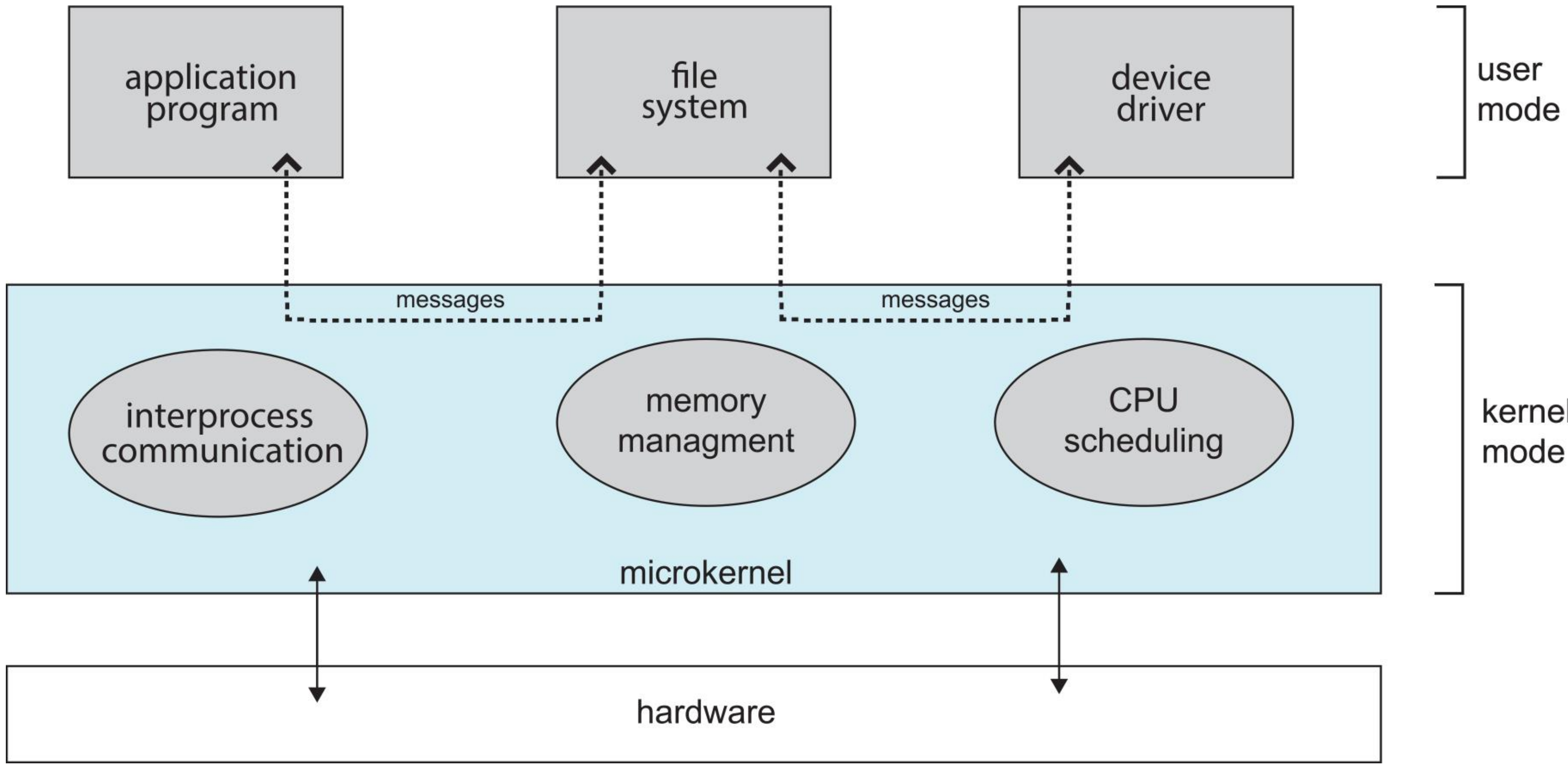
Benefits:

- 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

Microkernel System Structure



Modules

Many modern operating systems implement **loadable kernel modules** (**LKMs**)

- 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

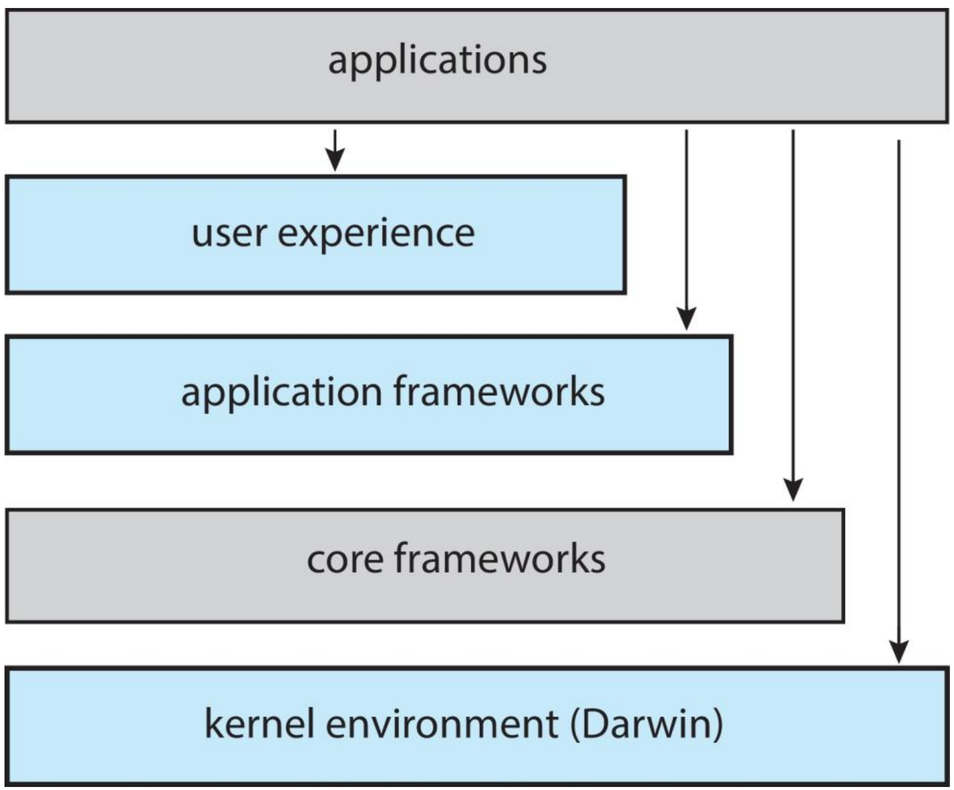
Overall, similar to layers but with more flexible

- Linux, Solaris, etc

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** UI plus **Cocoa** programming environment
  - Below is kernel consisting of Mach microkernel and BSD Unix parts, plus I/O kit and dynamically loadable modules (called **kernel extensions**)

macOS and iOS Structure



Android

Developed by Open Handset Alliance (mostly Google)

- Open Source

Similar stack to IOS

Based on Linux kernel but modified

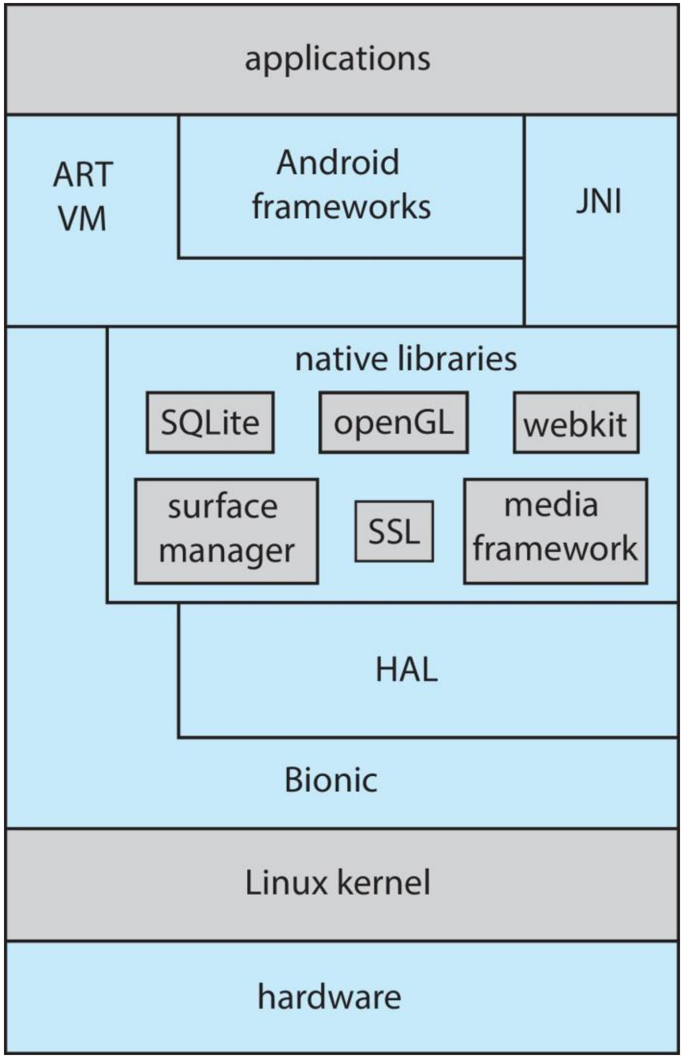
- Provides process, memory, device-driver management
- Adds power management

Runtime environment includes core set of libraries and Dalvik virtual machine

- Apps developed in Java plus Android API
  - Java class files compiled to Java bytecode then translated to executable than runs in Dalvik VM

Libraries include frameworks for web browser (webkit), database (SQLite), multimedia, smaller libc

Android Architecture



Building and Booting an Operating System

Operating systems generally designed to run on a class of systems with variety of perpherals

Commonly, operating system already installed on purchased computer

- But can build and install some other operating systems
- If generating an operating system from scratch
  - Write the operating system source code
  - Configure the operating system for the system on which it will run
  - Compile the operating system
  - Install the operating system
  - Boot the computer and its new operating system

Building and Booting Linux

Download Linux source code (http://www.kernel.org)

Configure kernel via "`make menuconfig`"

Compile the kernel using "`make`"

- Produces `vmlinuz`, the kernel image
- Compile kernel modules via "`make modules`"
- Install kernel modules into `vmlinuz` via "`make modules_install`"
- Install new kernel on the system via "`make install`"

System Boot

- When power initialized on system, execution starts at a fixed memory location
- Operating system must be made available to hardware so hardware can start it
  - Small piece of code – **bootstrap loader**, **BIOS**, stored in **ROM** or **EEPROM** locates the kernel, loads it into memory, and starts it
  - Sometimes two-step process where **boot block** at fixed location loaded by ROM code, which loads bootstrap loader from disk
  - Modern systems replace BIOS with **Unified Extensible Firmware Interface** (**UEFI**)
- Common bootstrap loader, **GRUB**, allows selection of kernel from multiple disks, versions, kernel options
- Kernel loads and system is then **running**
- Boot loaders frequently allow various boot states, such as single user mode

Operating-System Debugging

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."

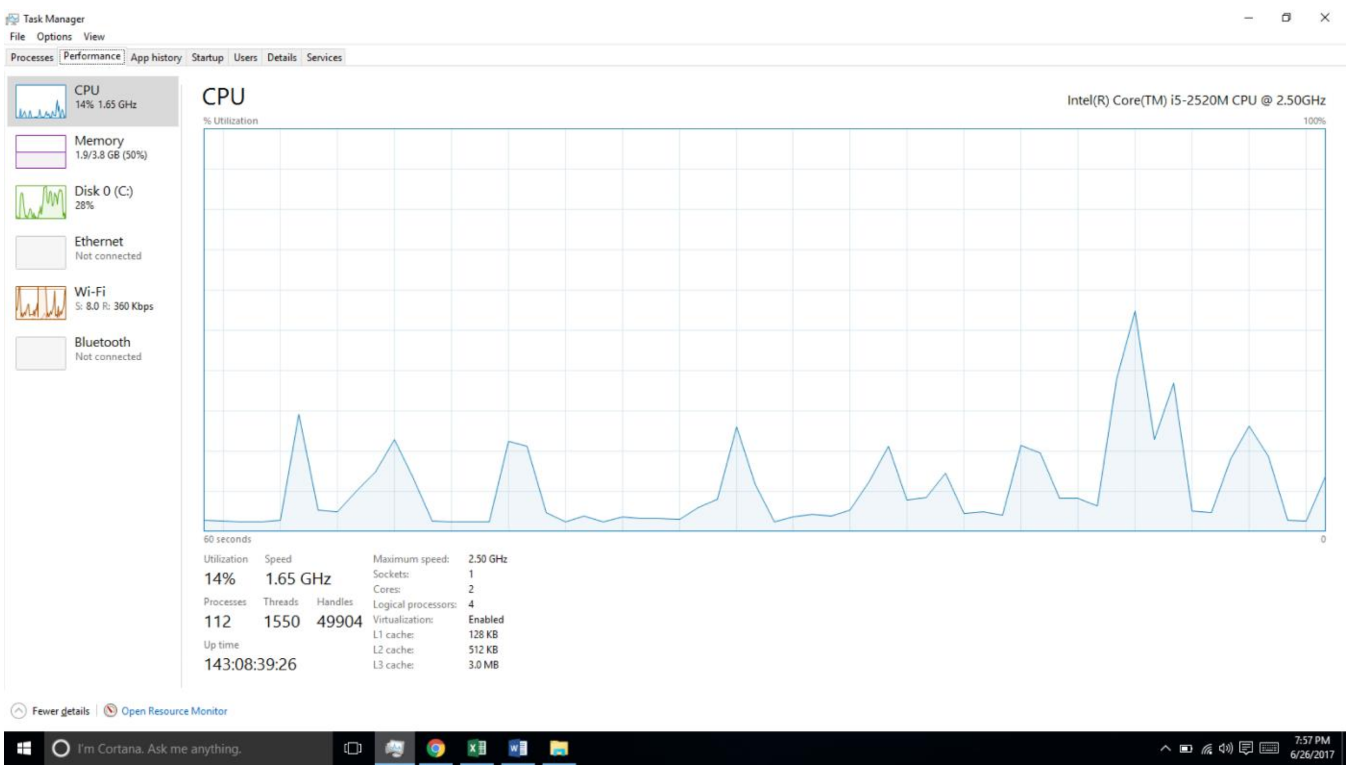
- OS generate **log files** containing error information
- Failure of an application can generate **core dump** file capturing memory of the process
- Operating system failure can generate **crash dump** file containing kernel memory
- Beyond crashes, performance tuning can optimize system performance
  - Sometimes using ***trace listings*** of activities, recorded for analysis
  - **Profiling** is periodic sampling of instruction pointer to look for statistical trends

Performance Tuning

Improve performance by removing bottlenecks

OS must provide means of computing and displaying measures of system behavior

For example, “top” program or Windows Task Manager



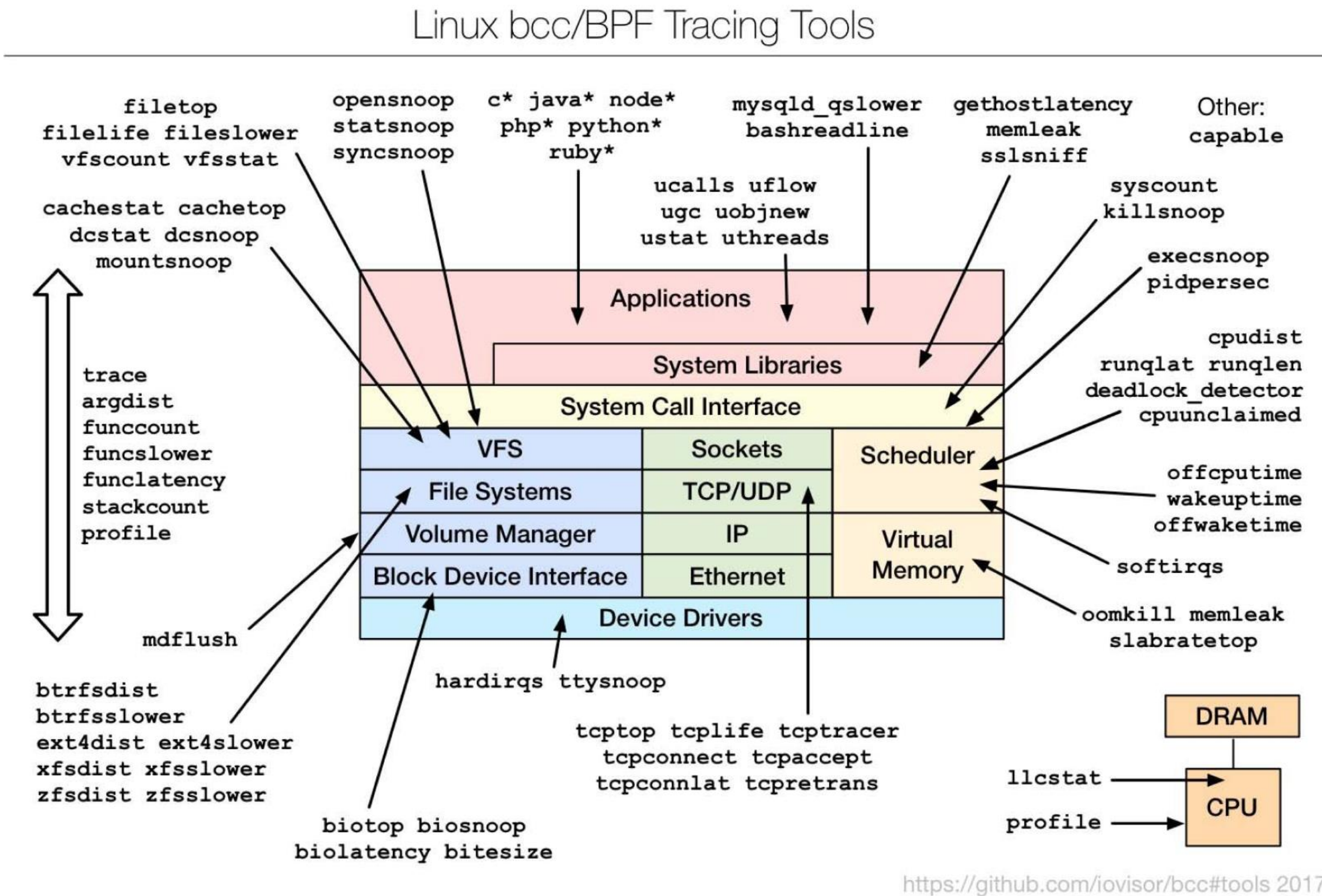
Tracing

Collects data for a specific event, such as steps involved in a system call invocation

Tools include

- strace – trace system calls invoked by a process
- gdb – source-level debugger
- perf – collection of Linux performance tools
- tcpdump – collects network packets

Linux bcc/BPF Tracing Tools



Takeaway

- System calls
  - Kernel interfaces provided to user space
  - Syscall table
  - Strace
- OS Structures
  - Monolithic vs microkernel
- Linkers and Loaders
  - Deep more by yourself
  - Static linking vs dynamic linking
  - Loaders
- Separation of policy from mechanism

End of Chapter 2