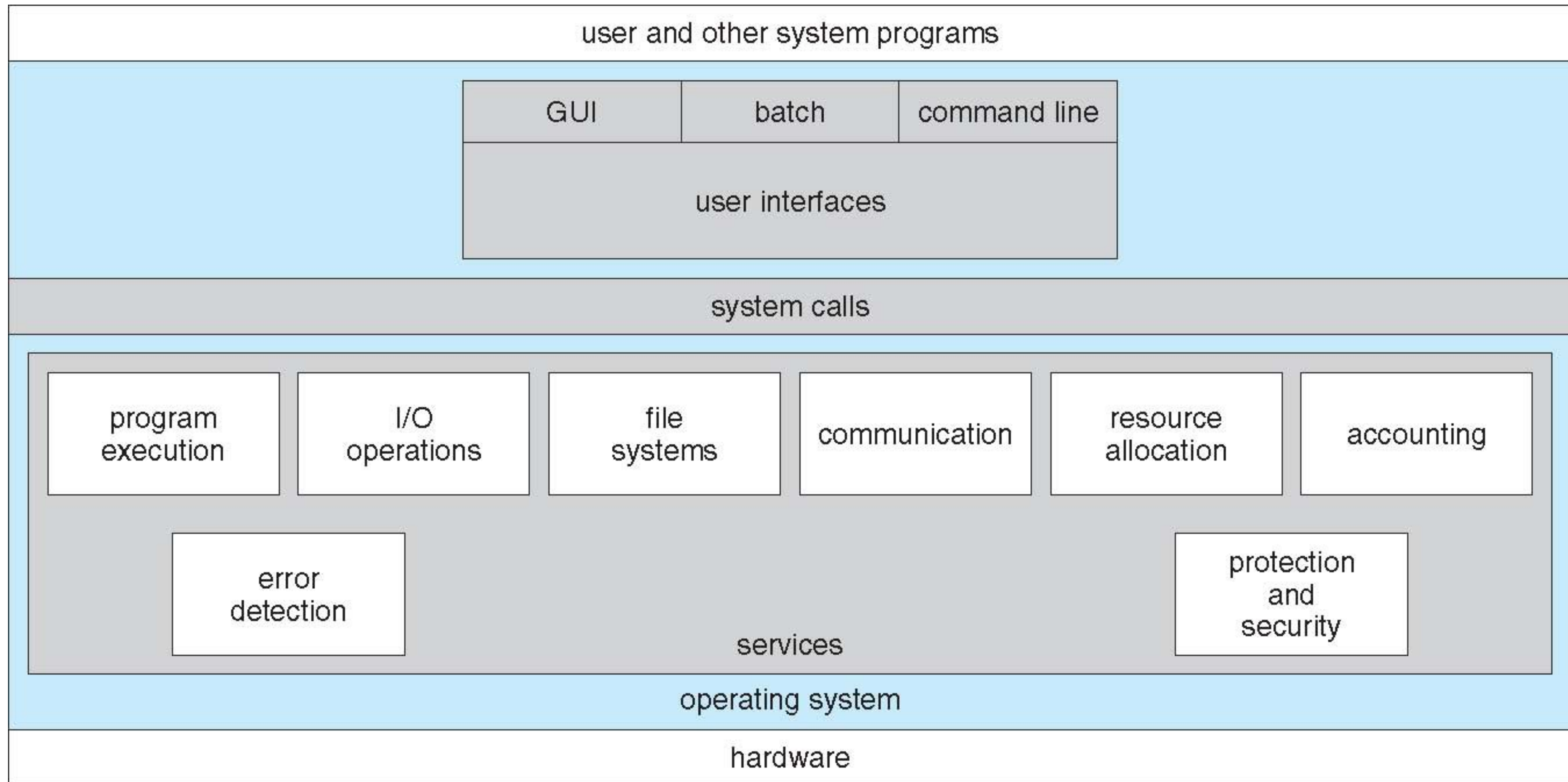
OS Structures

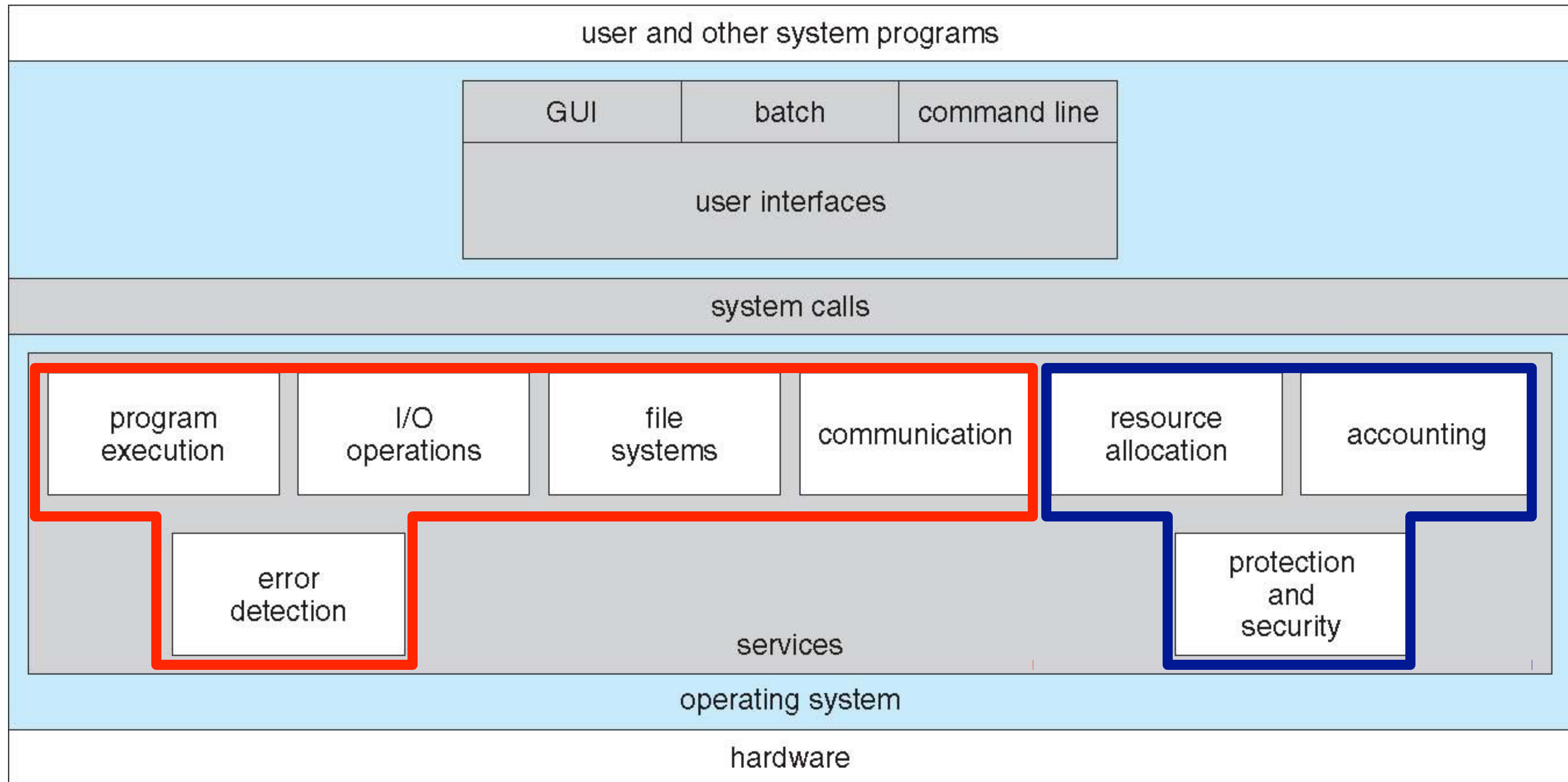
ICS332

Operating Systems

OS Services and Features



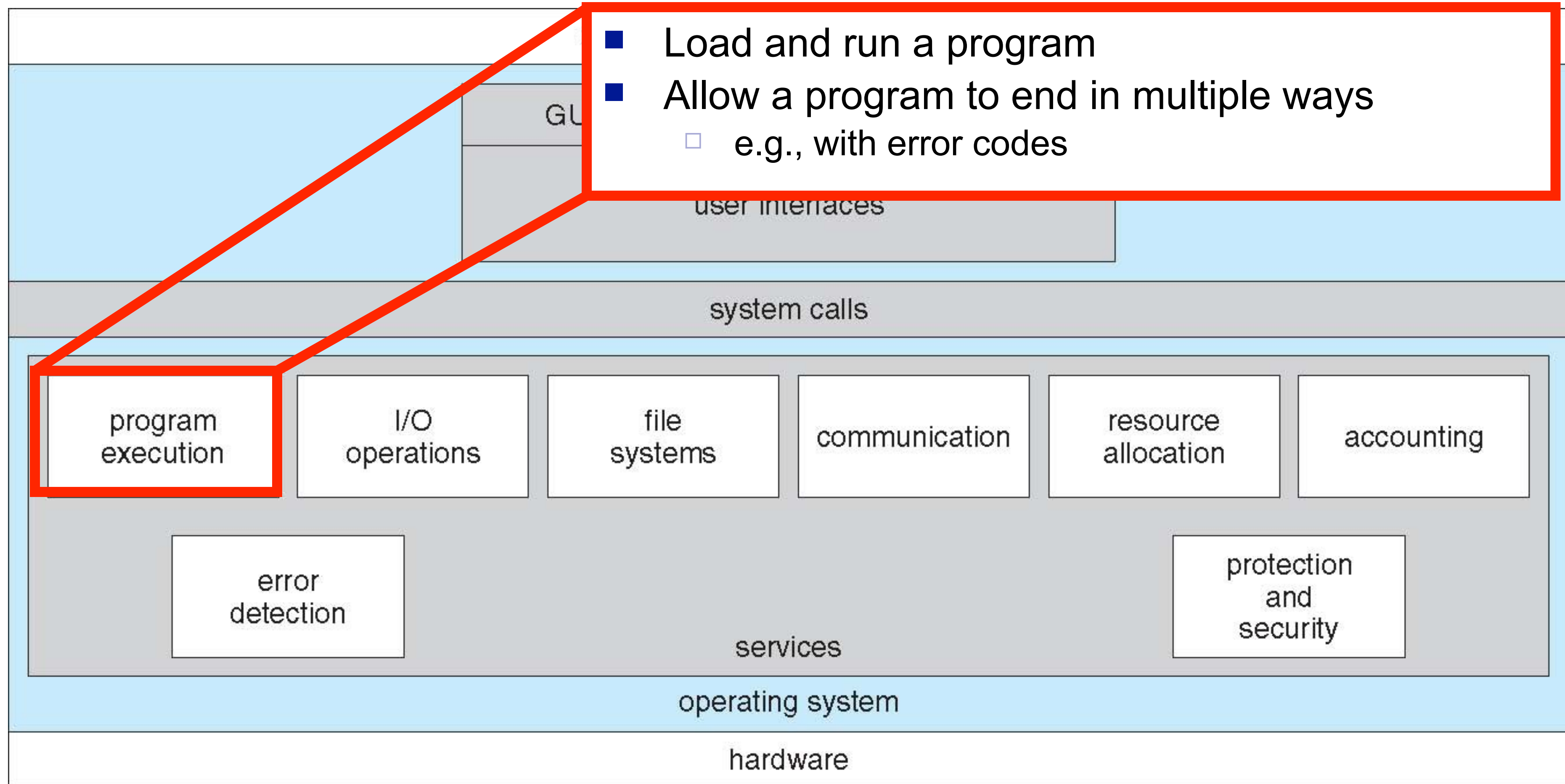
OS Services and Features



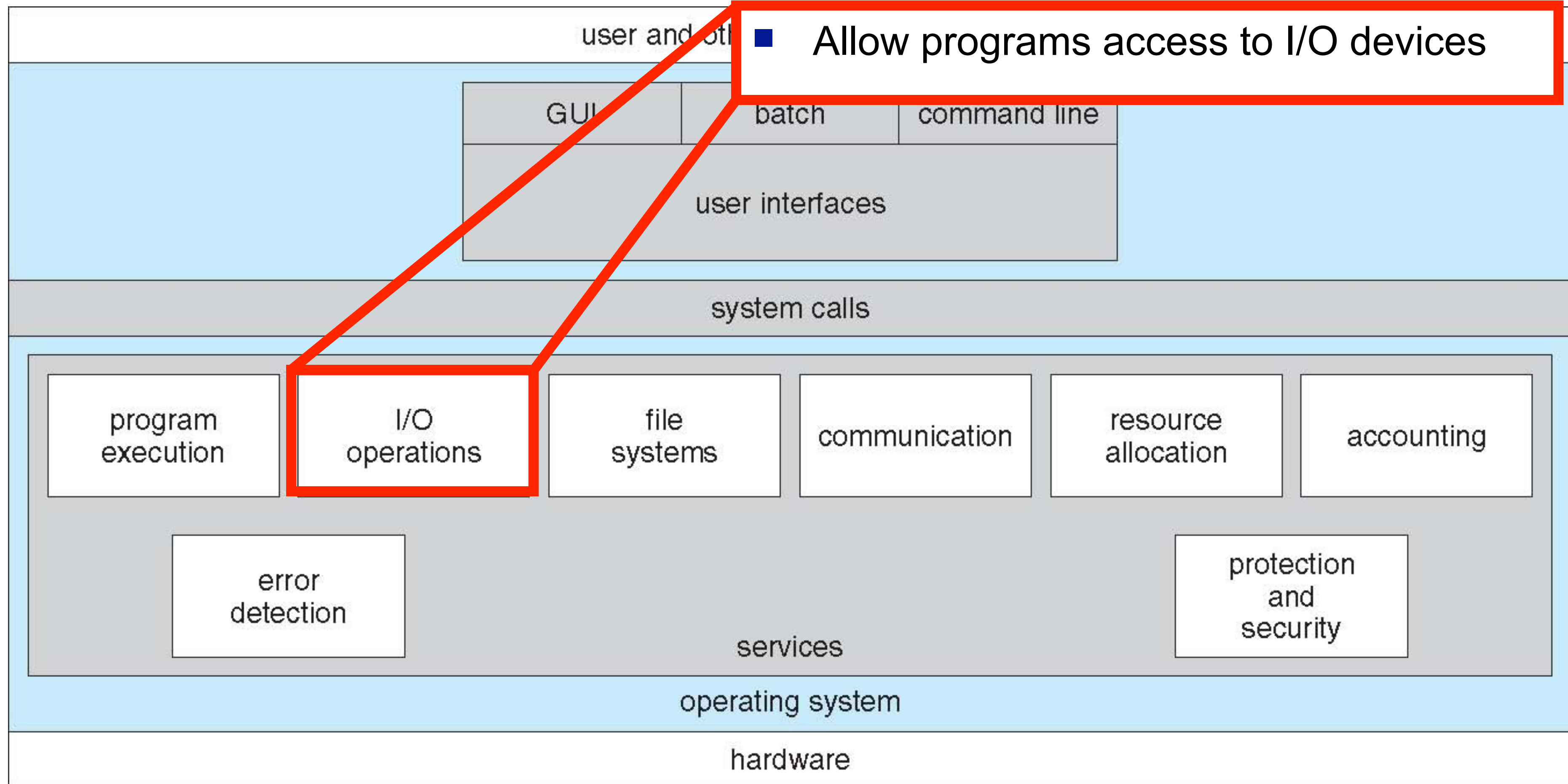
Helpful to users

Better efficiency/operation

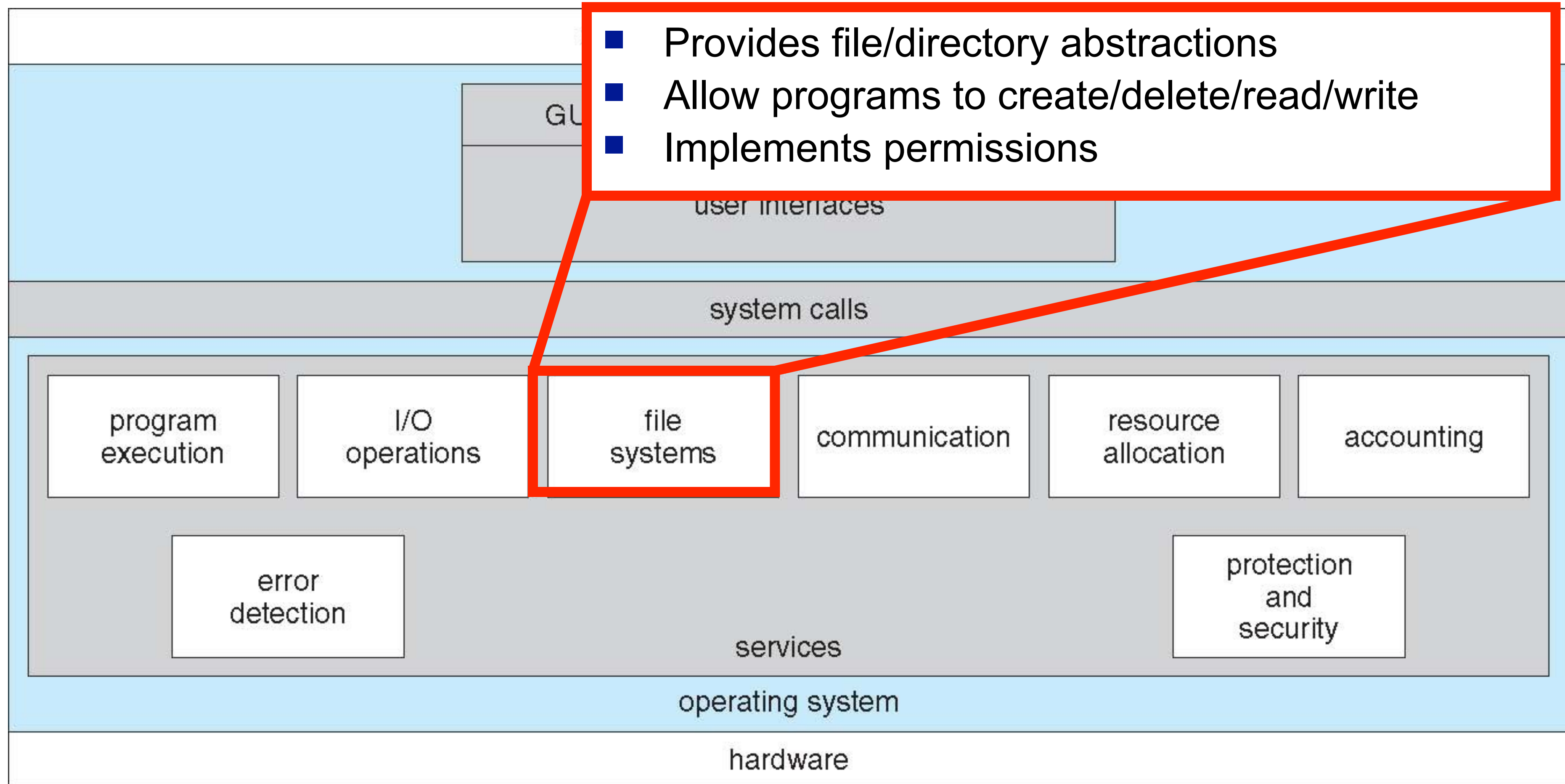
OS Services



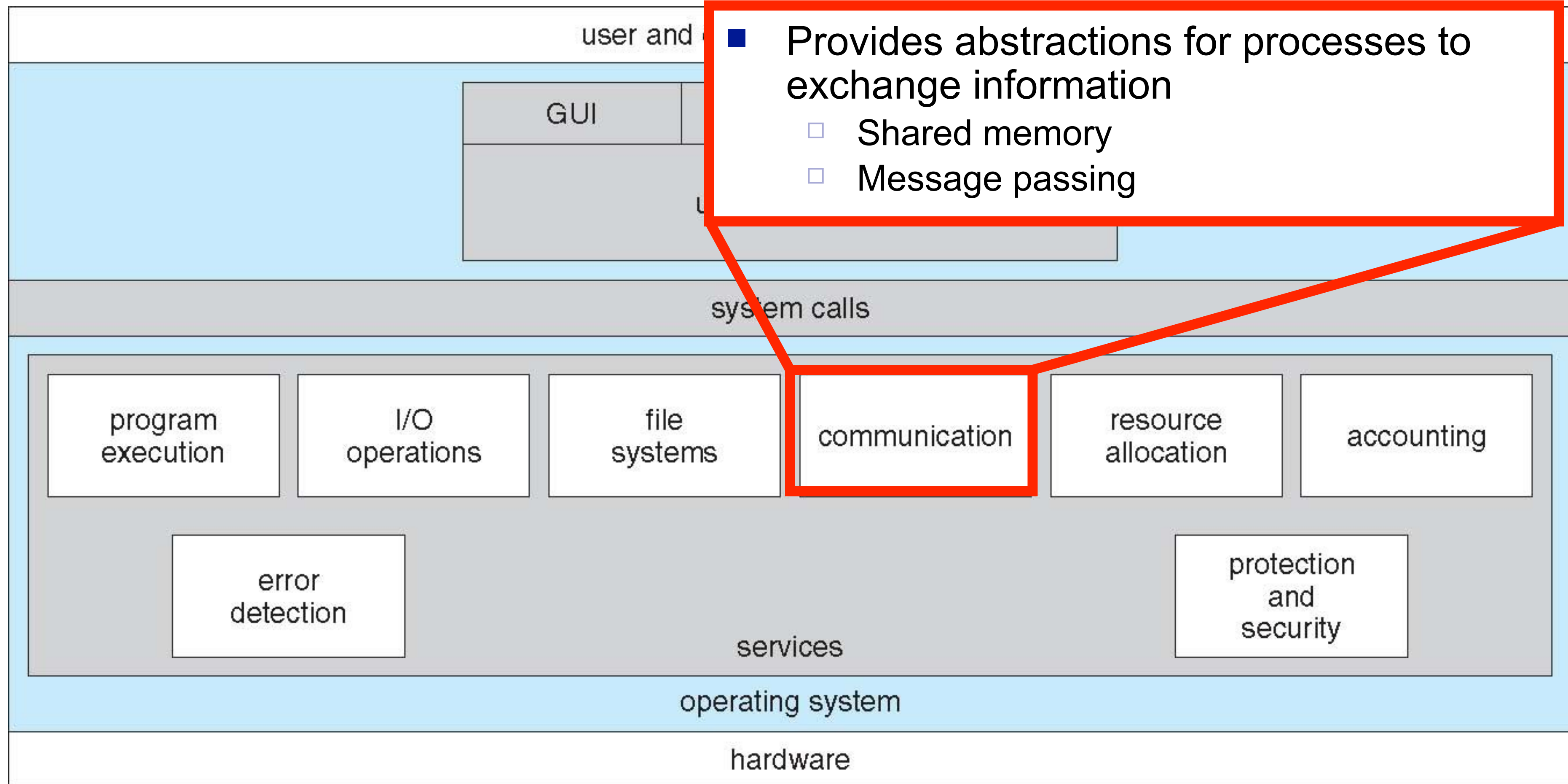
OS Services



OS Services



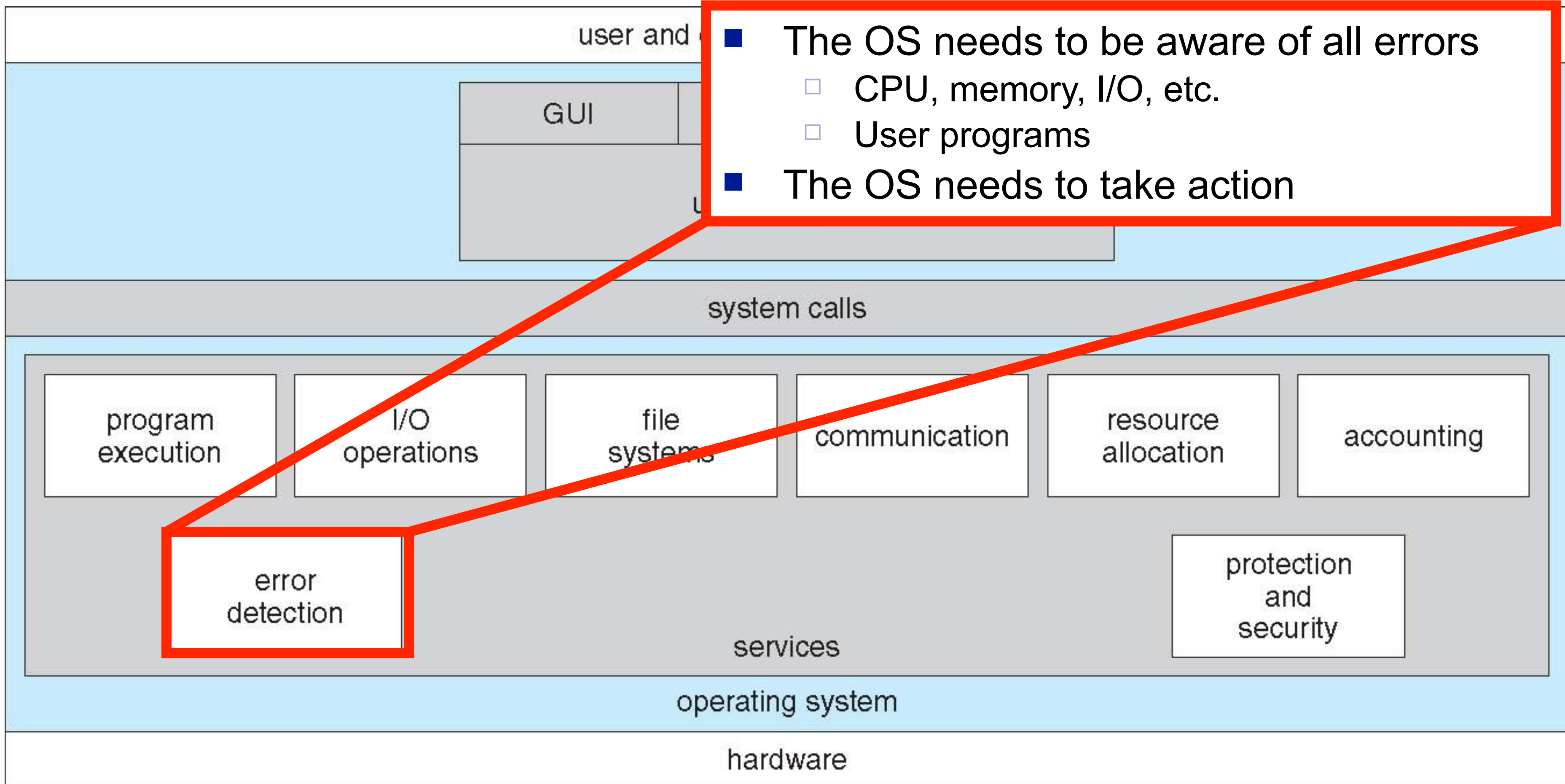
OS Services



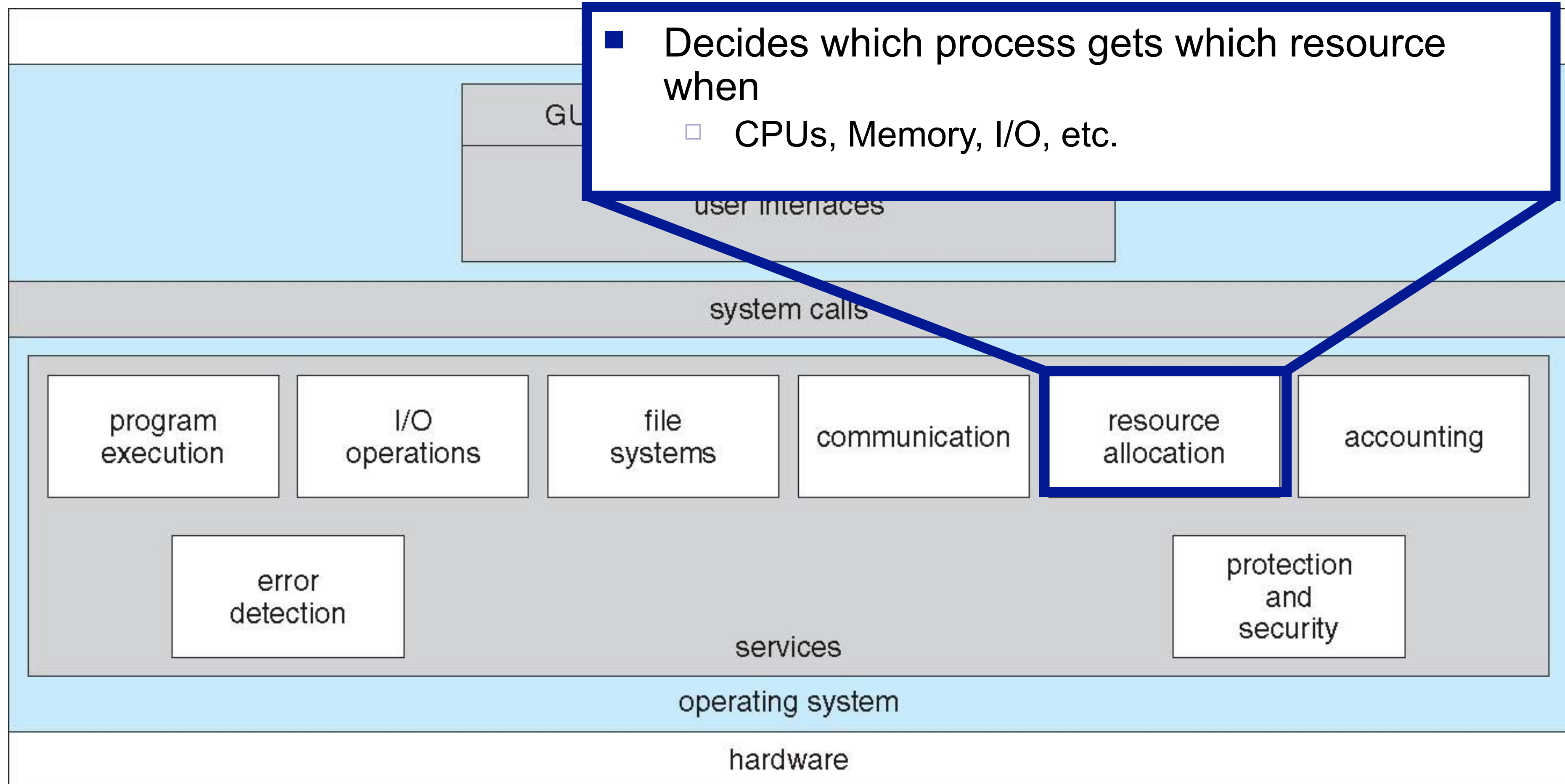
- Provides abstractions for processes to exchange information
  - Shared memory
  - Message passing

OS Services

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

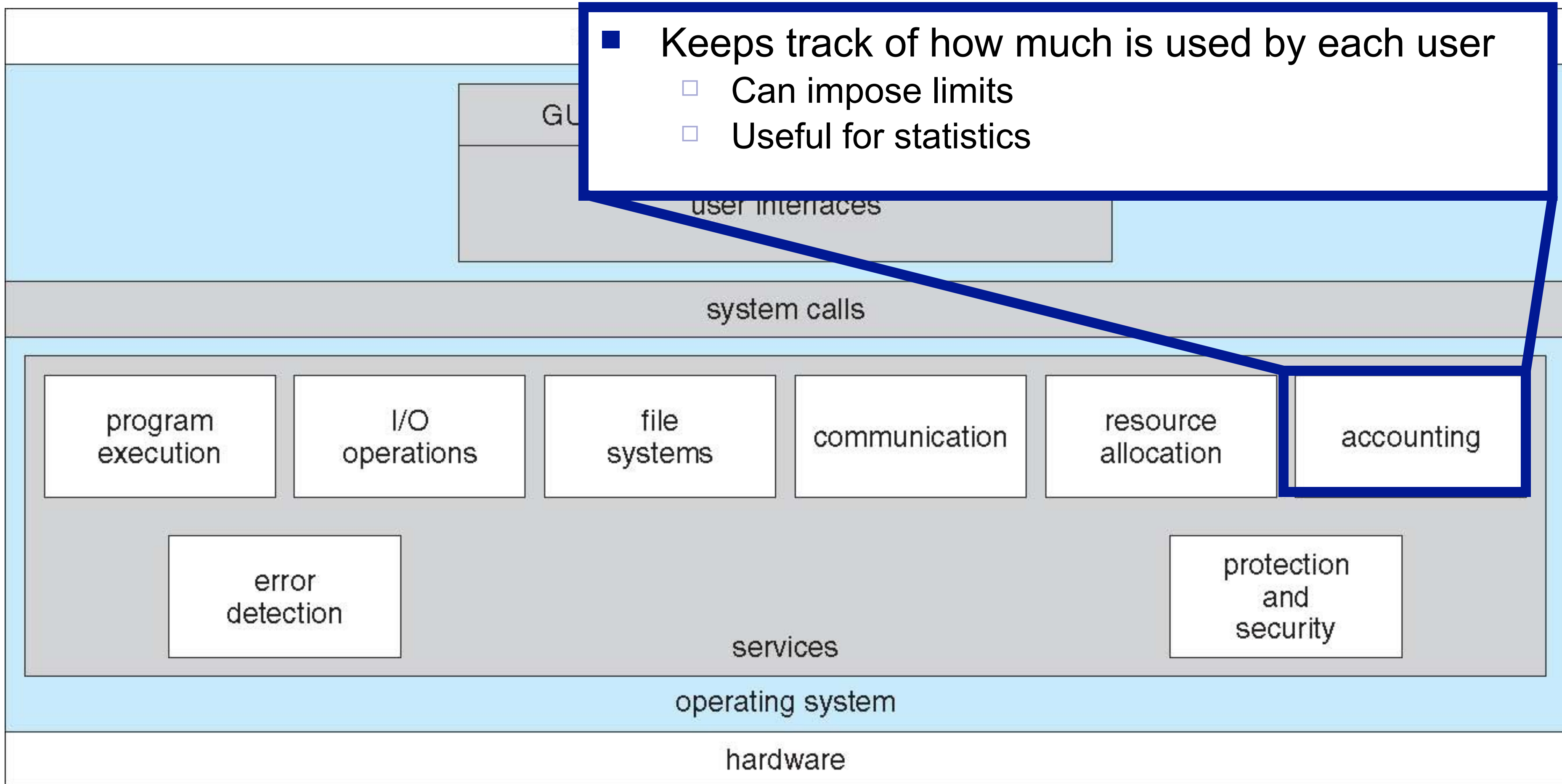


OS Features

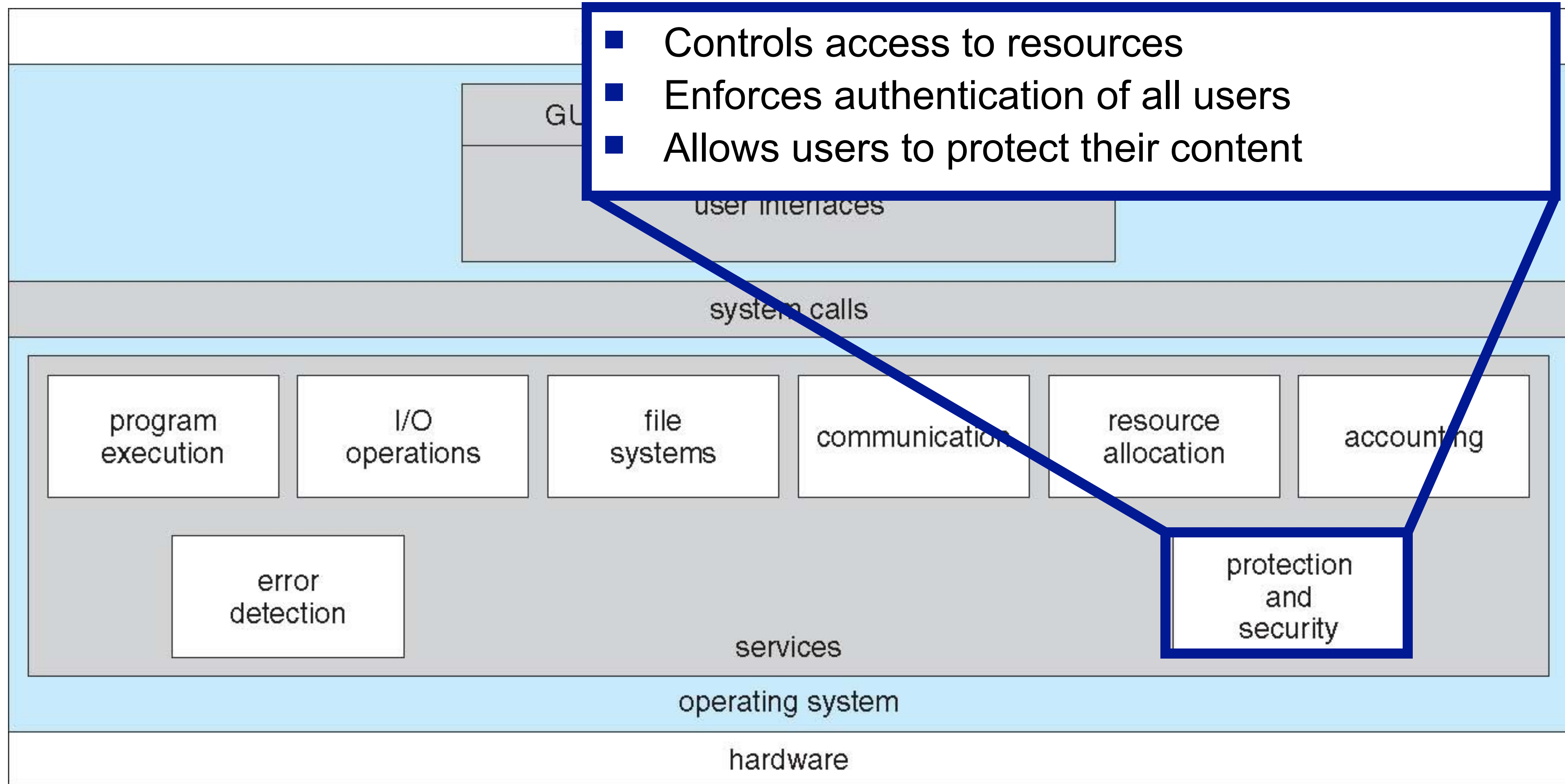


- Decides which process gets which resource when
  - CPUs, Memory, I/O, etc.

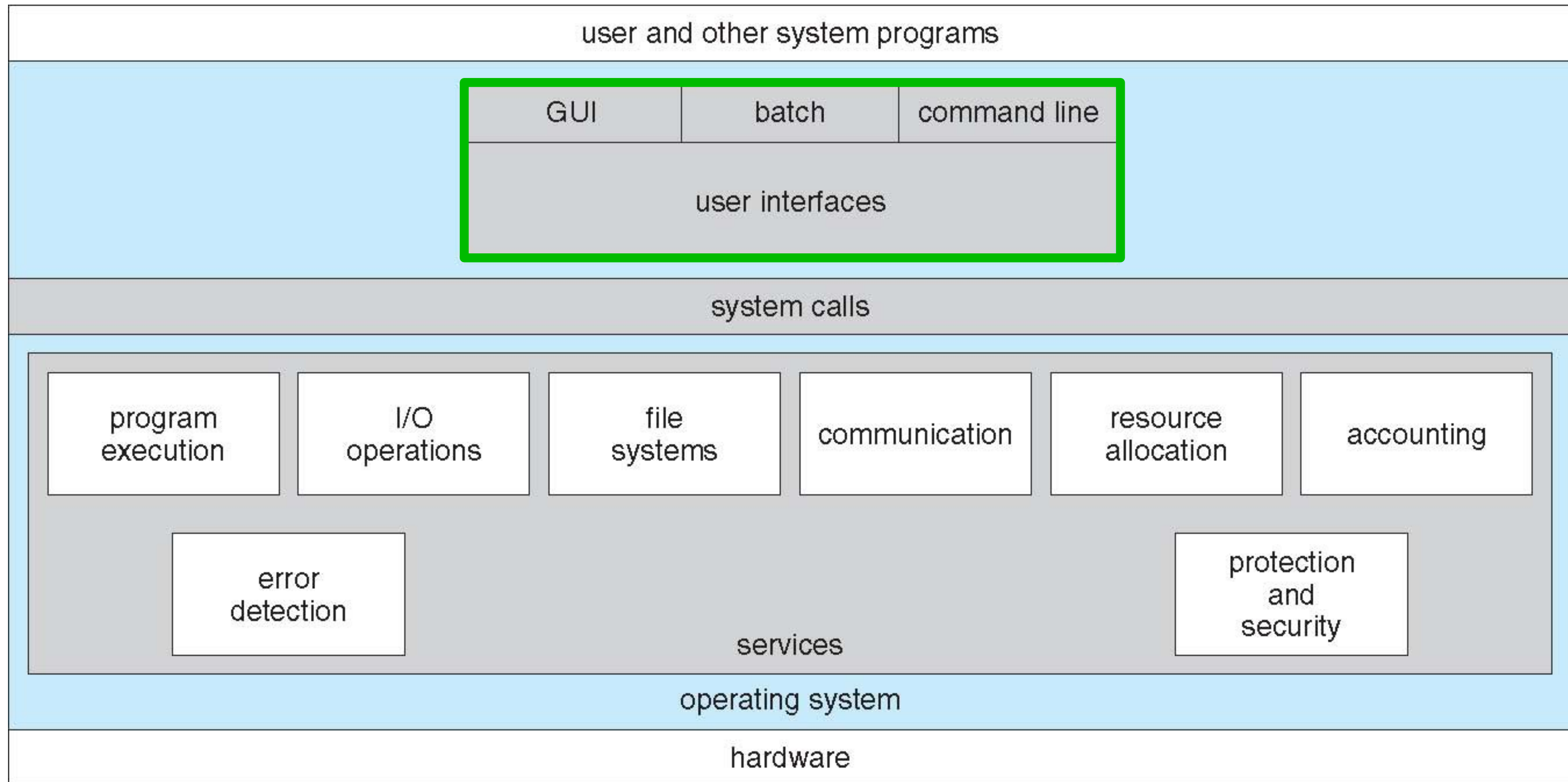
OS Features



OS Features



OS Services and Features



OS Interfaces: The CLI (aka Shell)

- Most OSes come with a **command-line interpreter** (CLI), typically called the **shell**
  - There are many UNIX Shells (bash, ksh, csh, tcsh, etc.)
  - Type “echo $SHELL” in a terminal to see which one you’re using
- The user types commands, and the shell interprets them
- The Shell implements some commands, meaning that the source code of the Shell contains the code of the commands
  - e.g., cd, bg, exit,
  - Let's see them all by doing a “man bash” (searching for the last occurrence of “BUILTIN”)
- The Shell cannot implement all commands (i.e., contain their code):
  - It would become huge
  - Adding a command would mean modifying the shell, leading users to do countless updates
- Instead, most Shells simply call system programs
  - In fact, the shell doesn’t understand (most) “commands”

OS Interfaces: The Shell (CLI)

- Example in UNIX: “rm file.txt” in fact executes the “/bin/rm” program that knows how to remove a file
  - “rm” is not a UNIX command, it’s the name of a program
- Adding a new “command” to the shell then becomes very simple
  - And we can all add our own
  - They are just programs that we think of as “commands”
  - In fact, we could write a program, call it “rm”, put its executable in /bin/, and we have a new rm “command”
- The terms “command” and “system program” are often used interchangeably
  - But it is important to remember that “rm” and “cd” are very different animals
- *type* built-in (type -t rm; type -t ll; type -t cd)

System Programs

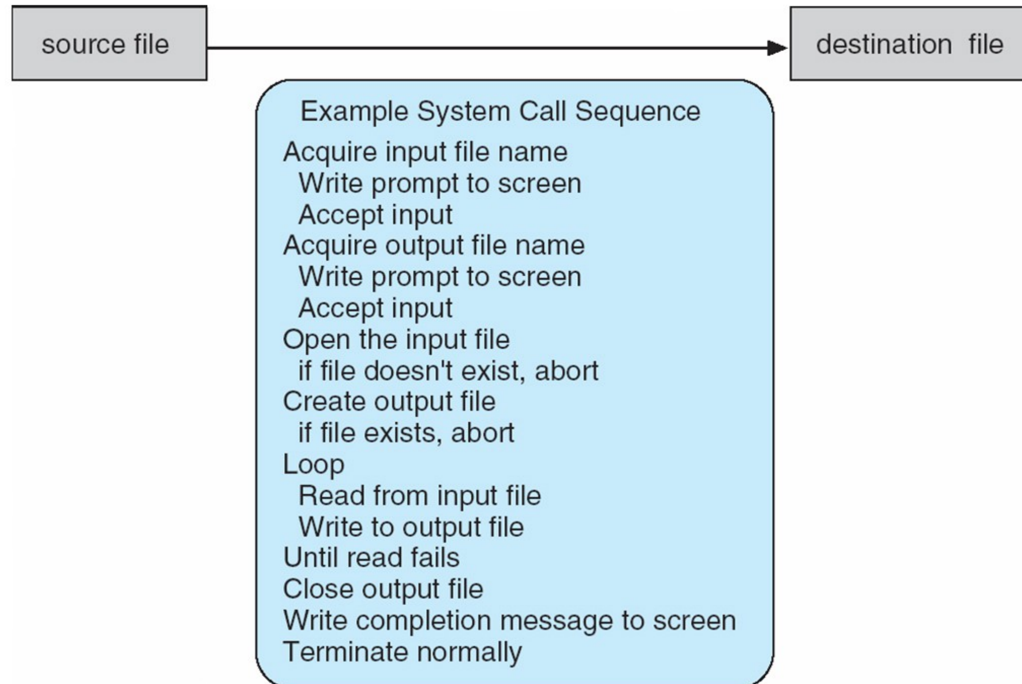
- Some system programs are simple wrappers around system calls (see later)
  - e.g., /bin/sleep
- Some are very complex
  - e.g., /bin/ls
- The term “system program” is in fact rather vague
- Some are thought of as commands, and some as applications
  - Do you think of the javac compiler as a command, an application or as a system program?
- System programs are not part of the “OS” per se, but many of them are always installed with it
  - The term “OS” is in fact rather vague also
    - What is often meant is “Kernel”

OS Interfaces: Graphical (GUI)

- Graphical interfaces appeared in the early 1970s
  - Xerox PARC research
- Popularized by Apple's Macintosh (1980s)
- Many UNIX users prefer the command-line for many operations, while most Windows users prefer the GUI
  - Mac OS used not to provide a command-line interface, but Mac OS X does: Terminal.
- Question: is the GUI part of the OS or not?
  - More general question: what's part of the OS?

System Calls

- System calls are the (lowest-level) interface to OS services
- Almost all useful programs need to call OS services
  - Could be more or less hidden to the programmer
  - Called directly (assembly), somewhat directly (C, C++), or more indirectly (Java)
- Even simple programs can use many system calls
  - Example from the book on page 63: a program that copies data from one file to another



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
  - (Create a file with dd) ; strace -xf cp <source> <target>

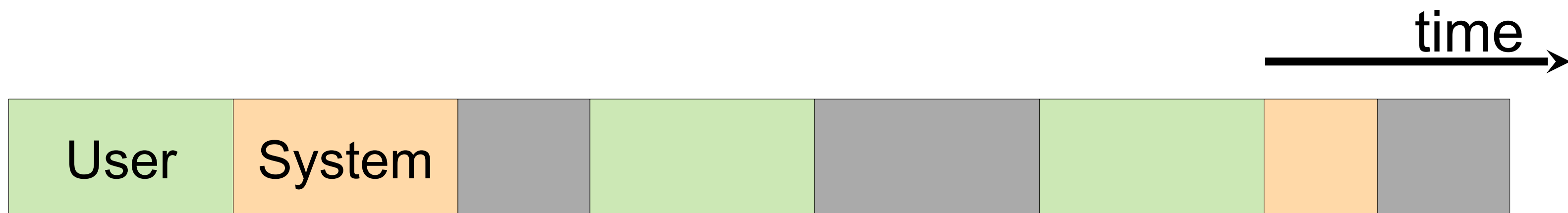
    Let's count the number of system calls using the wc command
  - Let's try with a tiny file and compare
  - Let's look at the system calls and see if they make sense
  - Let's try very simple commands and see... e.g.
- Conclusion: there are TONS of system calls
- strace can be "attached" to a running program!
  - to find out, e.g., why a program is stuck!

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 in on a *ls -R* command...

Time Spent in System Calls?

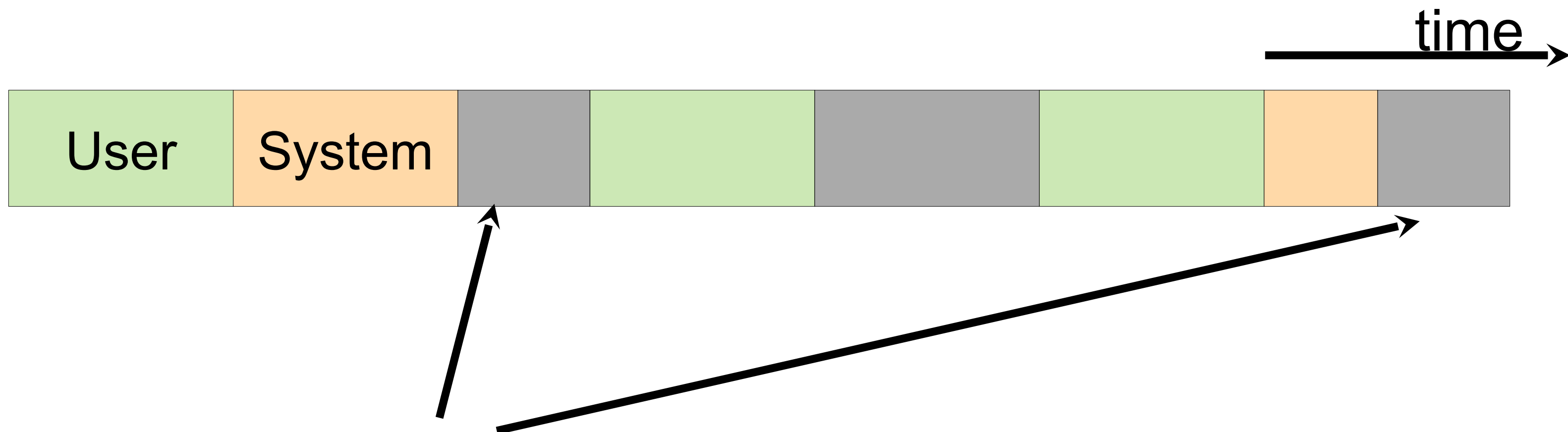
- The sum of the user time and the system time is not necessarily equal to the elapsed time
- Typical execution of a program:



- Any idea what those gray zones are?

Time Spent in System Calls?

- The sum of the user time and the system time is not necessarily equal to the elapsed time
- Typical execution of a program:

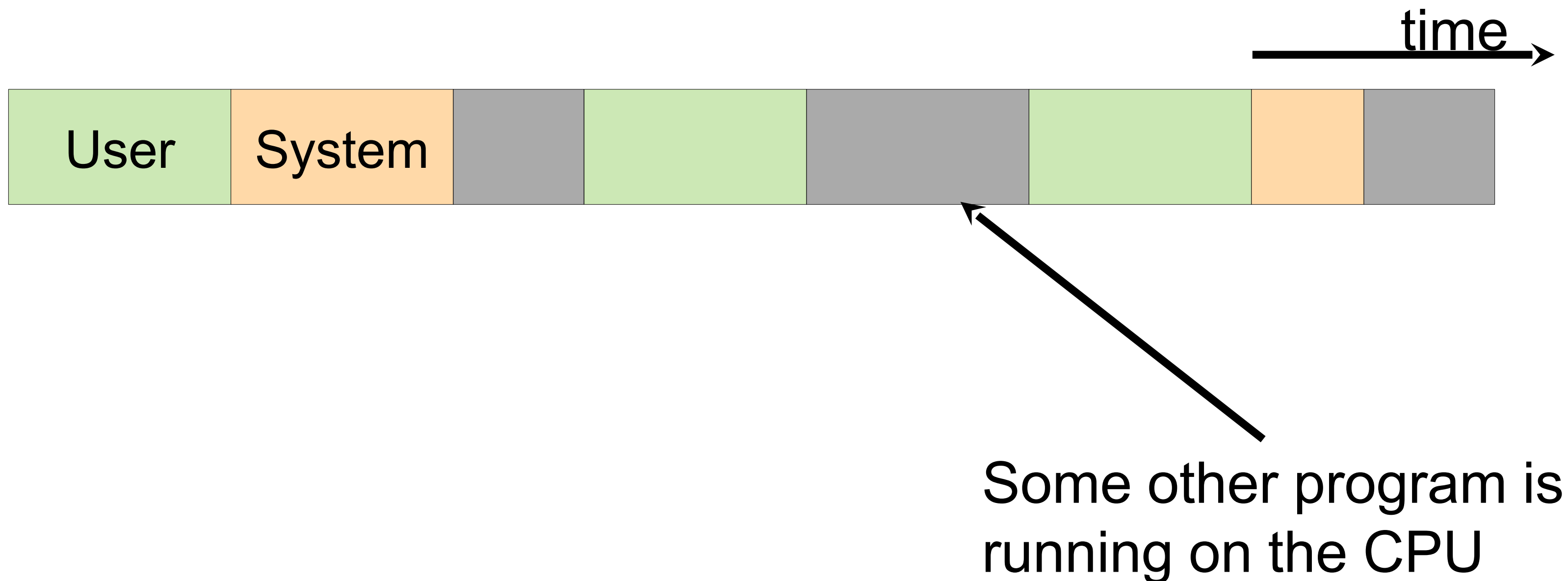


The program is waiting for some device (disk, network, keyboard), as requested by a system call

Probably more realistic

Time Spent in System Calls?

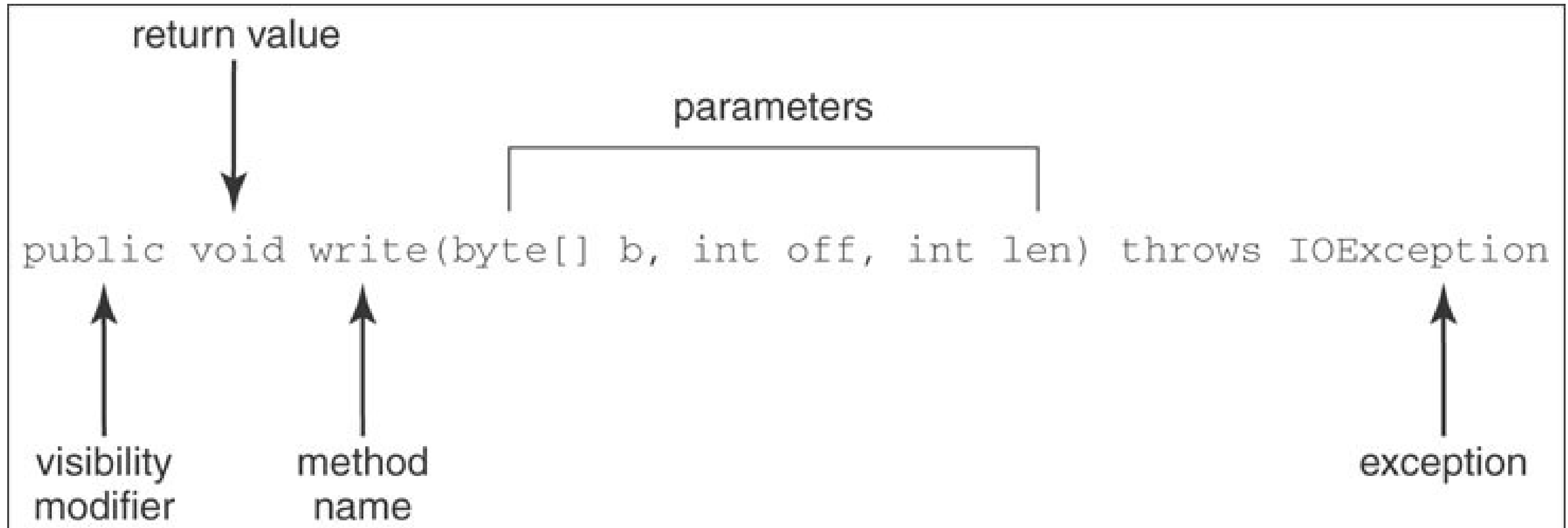
- The sum of the user time and the system time is not necessarily equal to the elapsed time
- Typical execution of a program:



APIs

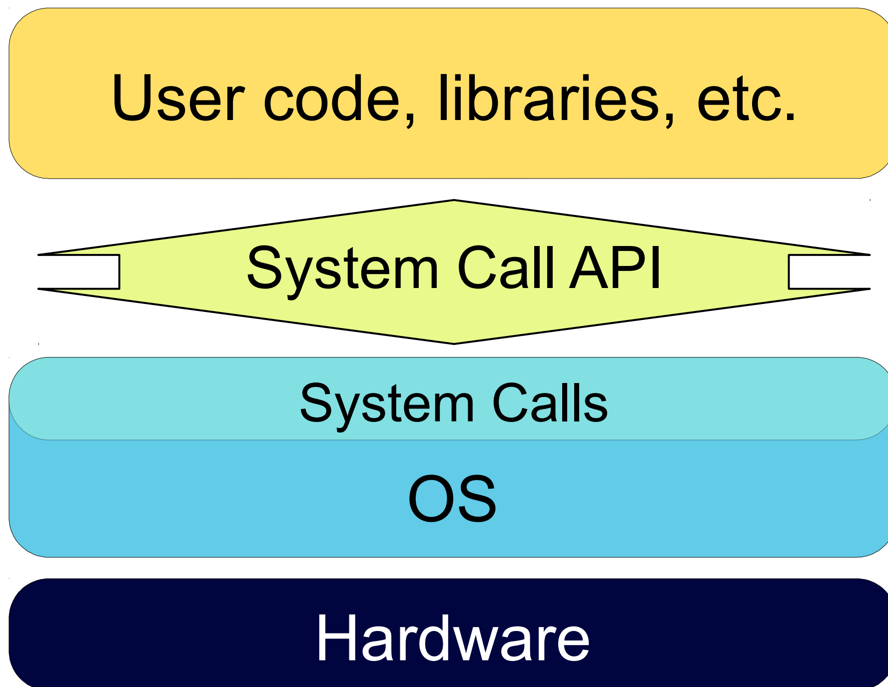
- System calls are mostly accessed by programs via a high-level Application Program Interface (API)
  - API functions can call (multiple) system calls under the cover
  - API calls are often simpler than full-fledge system calls
    - Some system calls are really complicated (Programmers would likely write their own “wrappers” anyway)
    - In many cases, however, the API call is very similar to the corresponding system call (just a “wrapper”)
- If the API is standard, then the code can be portable
- Standard APIs:
  - Win16, Win32, Win64 API for Windows OS
  - POSIX [Portable Operating Systems Interface IEEE-IX] for UNIX systems (POSIX: HP-UX, AIX, Solaris, OS X; POSIX-Compliant: Linux, Android, Cygwin)
  - The Java API (provides API to the Java Virtual Machine (JVM) which has OS-like functionality on top of the OS)

Java API Example



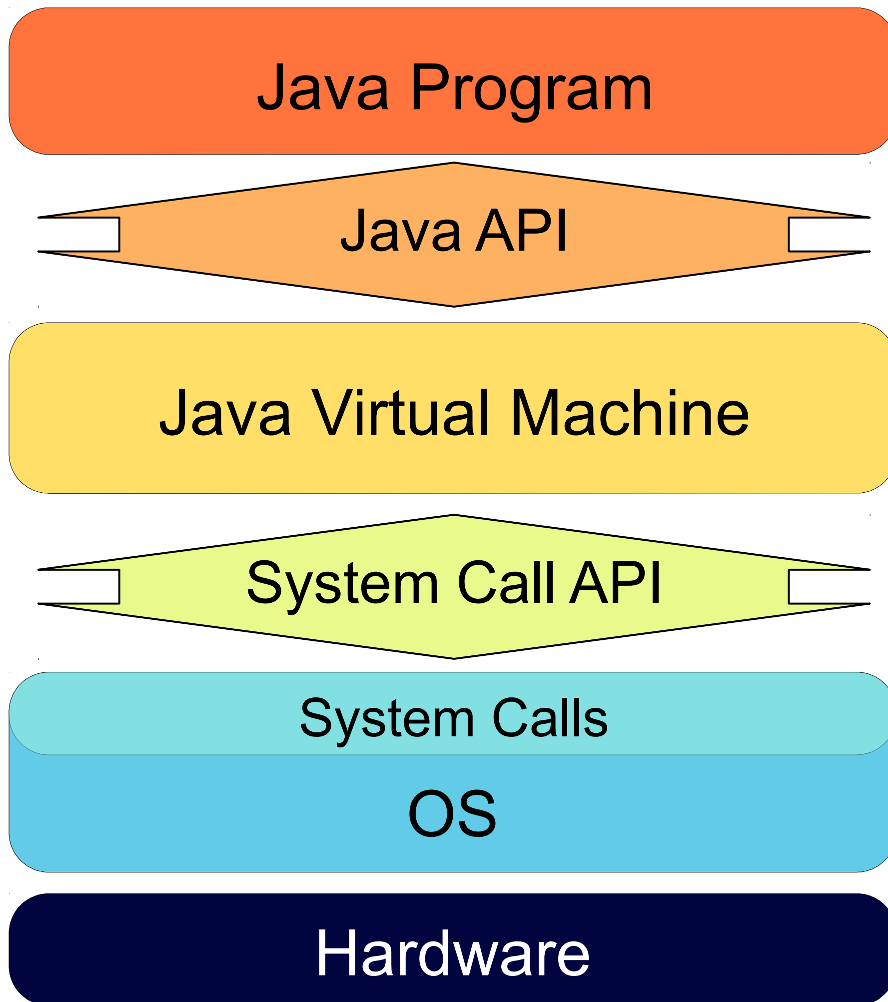
- The write method in java.io.OutputStream, to write to a file or network
  - b: the byte array that contains the data to be written
  - off: the starting offset in array b
  - len: the number of bytes to be written
- Similar in spirit (if not in details) to the write system calls in other standard APIS
- Let’s do a “man 2 write” on a Linux system and compare
  - The book show the read system call in C (page 64)

The JVM and the OS



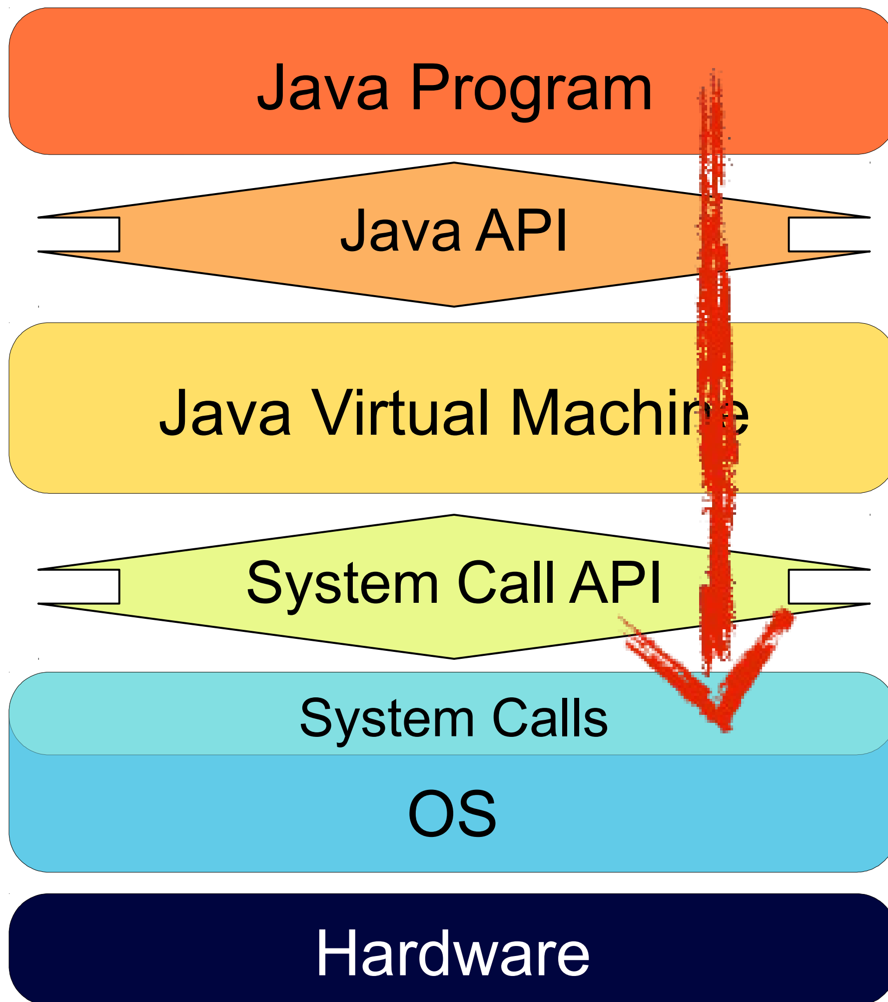
- This is the traditional view of an application running on top of the OS
- The application uses the System Call API to place system calls
- The OS performs work on behalf of the application for each system call
- A lot of that work entails interacting with the hardware

The JVM and the OS



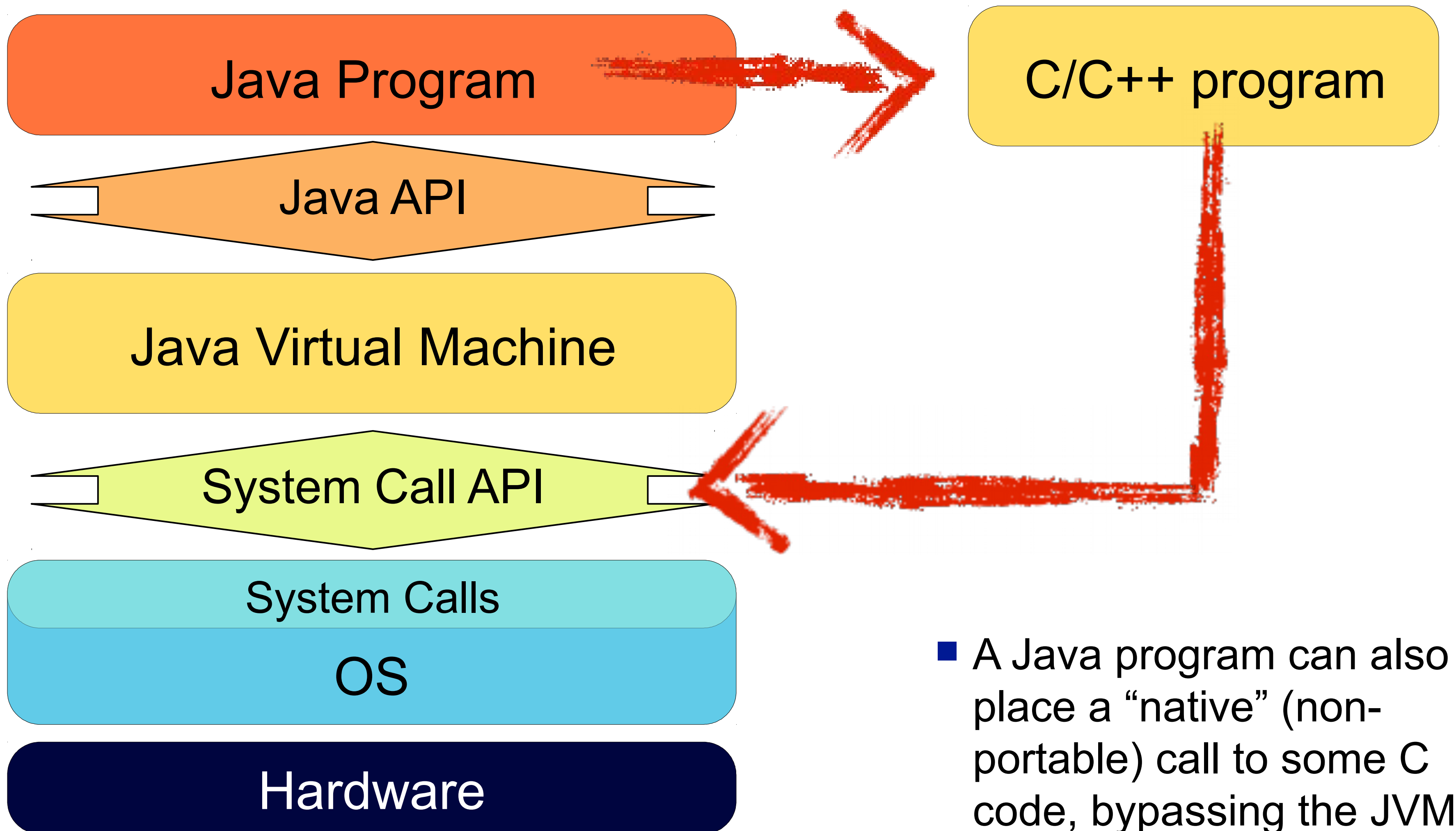
- The JVM is just an application
- It interacts with the OS using the System Call API
- But it also interprets byte code that places calls to the Java API
- The JVM performs work on behalf of the byte code for each API call
- Some of this work is then passed on to the OS from the JVM via the System Call API

The JVM and the OS



- The JVM is just an application
- It interacts with the OS using the System Call API
- But it also interprets byte code that places calls to the Java API
- The JVM performs work on behalf of the byte code for each API call
- Some of this work is then passed on to the OS from the JVM via the System Call API
- The Java API is NOT an interface to system calls
- BUT some of the API calls place (more or less direct)

The JVM and the OS

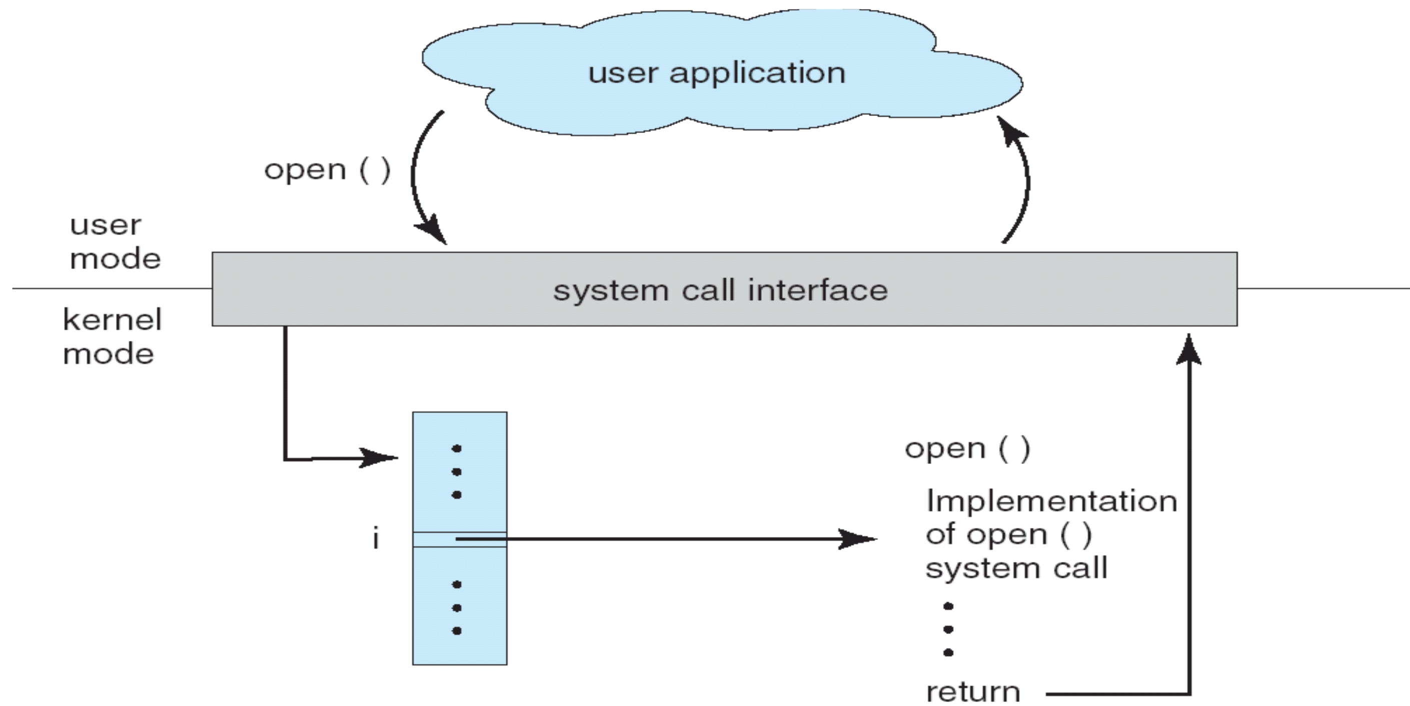


- A Java program can also place a “native” (non-portable) call to some C code, bypassing the JVM

(See JNI for further info)

The System Call Interface

- Remember that each system call is identified by a number
- The run-time support system provides a system call interface
  - The run-time support system is a set of useful functions built into libraries included with a compiler
- System calls numbers are stored in an internal table



The Syscall Table

- Let's look a bit inside the Linux Kernel
- include/[uapi/]asm/unistd_64.h defines syscall numbers for a 64-bit system

  (*locate <filename>; updatedb* as super-user to create the filenames database)

  - There are ~400 system calls
- Can we identify a few of them?

- Process control
- File management
- Device management
- Information maintenance
- Communications
- Protection

Read Section 2.4 as “warm up”

We’ll talk about the above in detail in future lectures

OS Design

- We don't know the best way to design and implement an OS
- As a result, the internal structure of different OSes can vary widely
  - Luckily, some approaches have worked well
- Goals lead to 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
      - Basically all the good software engineering that one should almost always strive for

Mechanisms and Policies

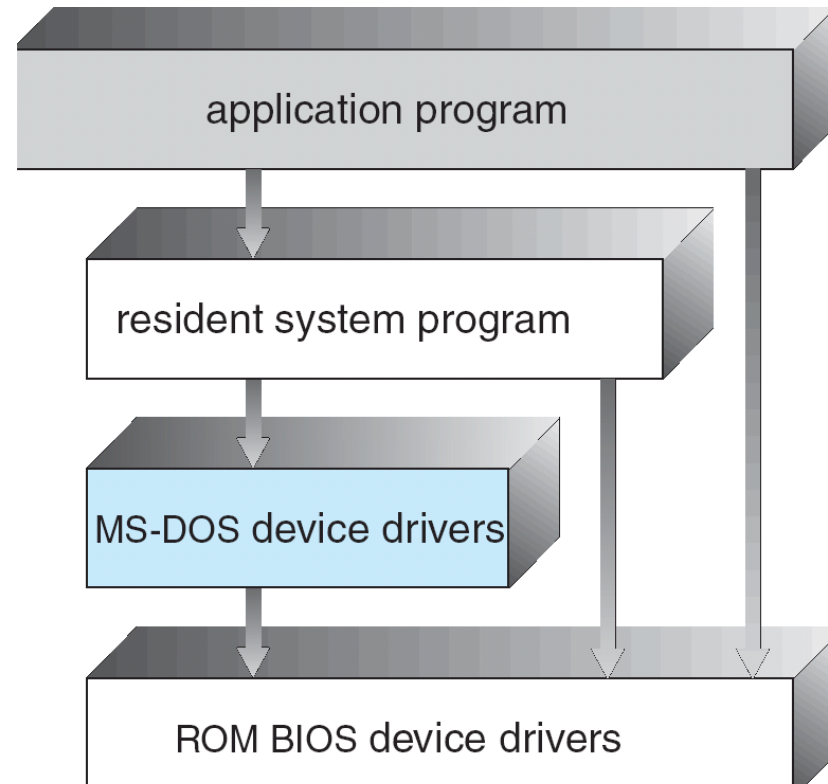
- One ubiquitous principle: separating mechanisms and policies
  - Policy: *what* should be done
  - Mechanism: *how* it should be done
- Separation is important so that, most of the time, one can change policy without changing mechanisms
  - Mechanisms should be low-level enough that many useful policies can be built on top of them
  - Mechanisms should be high-level enough that implementing useful policies on top of them is not too labor intensive
- Some OS designs take this separation principle to the extreme (e.g., Microkernels)
  - e.g., Solaris implements completely policy-free mechanisms
- Some OS designs not so much
  - e.g., Windows

OS Implementation

- OSes used to be written in assembly
  - e.g. MS-DOS (yikes!)
- Modern OSes are written in languages like C, or “improved” Cs, with a splash of assembly
  - Linux, Windows XP
  - The OS should be fast, and compilers are good enough, and machines are fast enough that it makes sense nowadays to use high-level languages
    - Besides, some small crucial sections can be rewritten in assembly if needed (not so much for speed as for calling specific instructions)

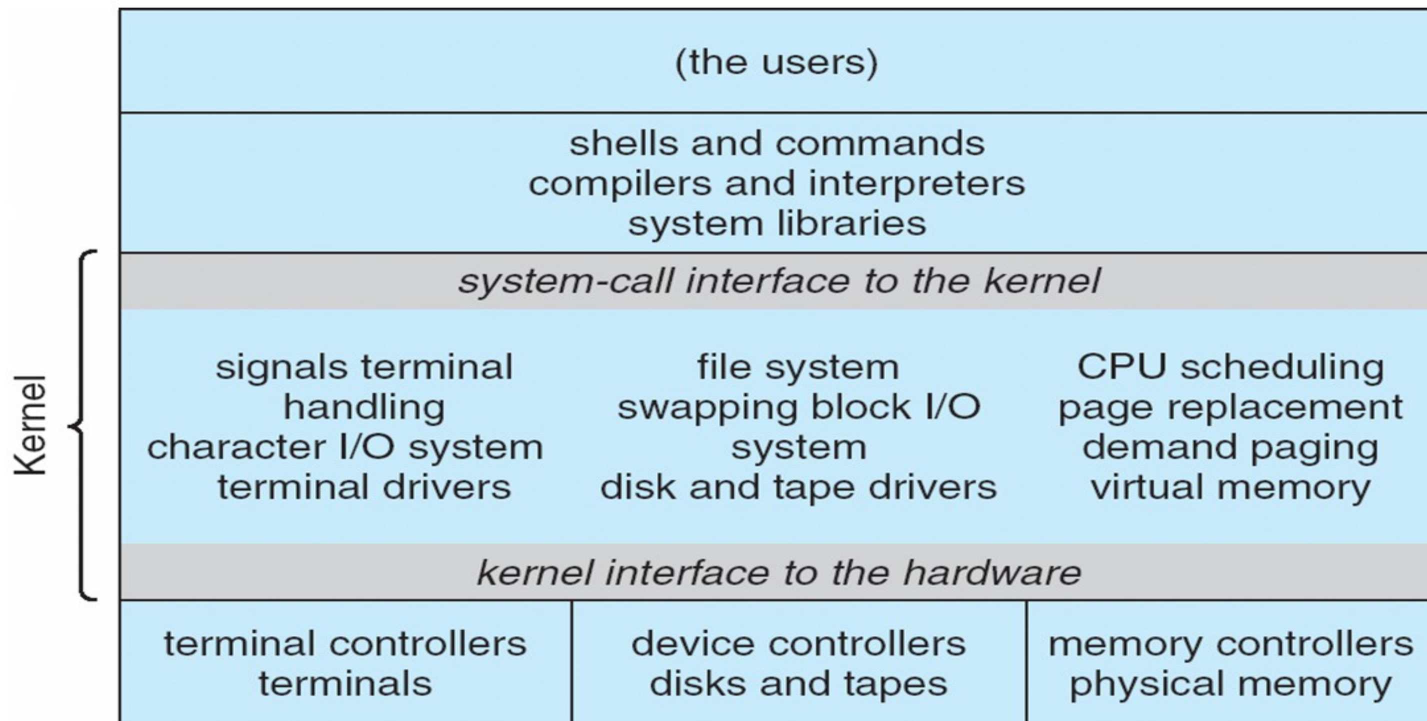
OS Structure: Simple

- Early OSes didn't really have a precisely defined structure (which became a problem when they grew beyond their original scope)
- MS-DOS was written to run in the smallest amount of space possible, leading to poor modularity, separation of functionality, and security
  - e.g., user programs can directly access some devices!
  - the hardware at the time had no mode bit for user/kernel differentiation, so security wasn't happening anyway

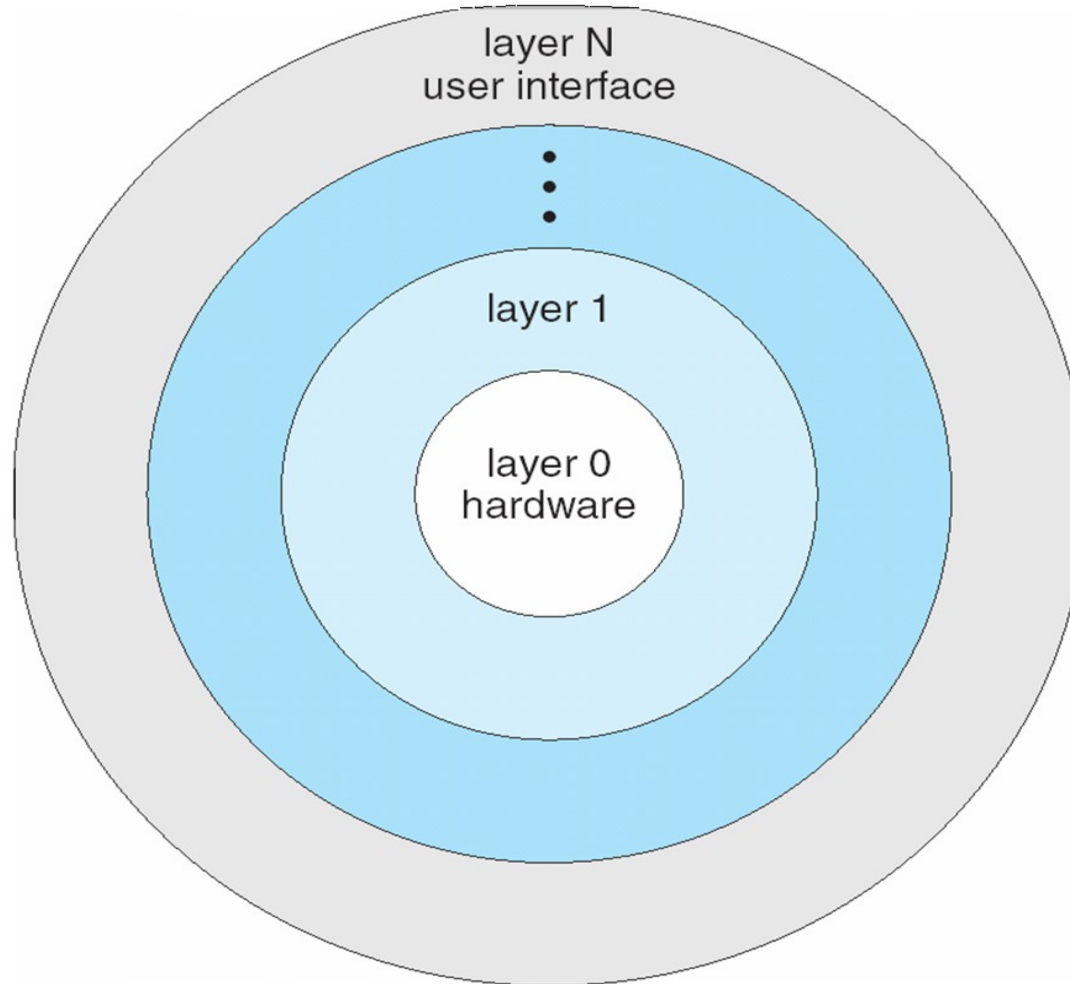


OS Structure: Simple

- Early UNIX also didn’t have a great structure, but at least had some simple layering
  - The huge, monolithic Kernel did everything and was incredibly difficult to maintain/evolve



OS Structure: Layered



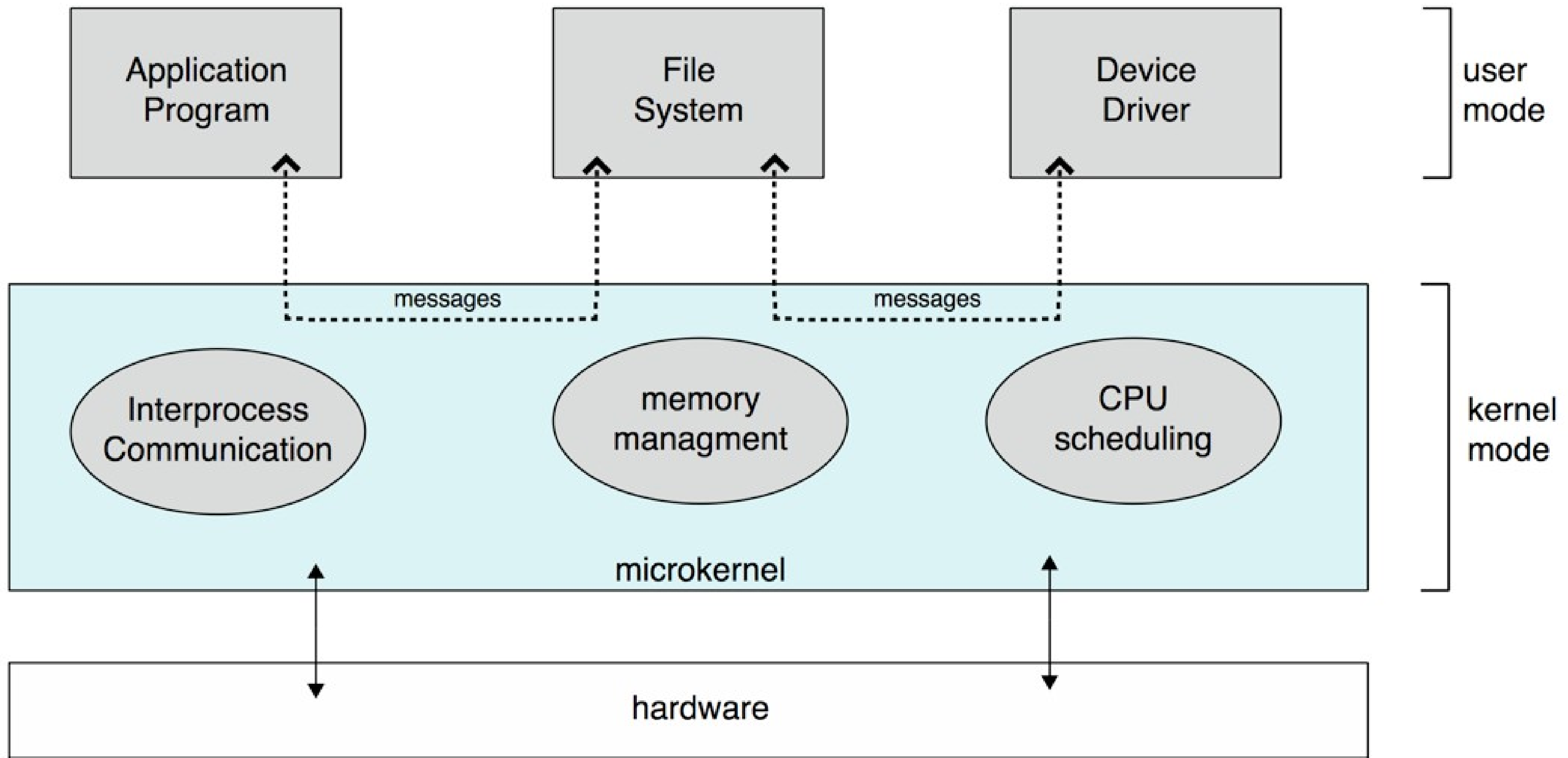
OS Structure: Layered

- Natural way to add more modularity: pack layers on top of each other
  - Layer n+1 uses *only* layer n
    - Everything in layers below is nicely hidden and can be changed
  - Simple to build and debug
    - debug layer n before looking at layer n+1
- Sounds nice, but what goes in what layers?
  - For two functionalities X and Y, one must decide if X is above, at the same level, or below Y
  - This is not always so easy
- And it can be much less efficient
  - Going through layers for each system call takes time
    - Parameters put on the runtime stack, jump, etc.
- There should be few layers

OS Structure: Microkernels

- By contrast with the growing monolithic UNIX kernel, the microkernel approach tries to remove as much as possible from the kernel and putting it all in system programs
  - Kernel: process management, memory management, and some communication
- Everything is then implemented with client-server
  - A client is a user program
  - A server is a running system program, in user space, that provides some service
  - Communication is through the microkernel's communication functionality
- This is very easy to extend since the microkernel doesn't change
  - And no decision problems about layers
- Problem: increased overhead
  - WinNT 4.0 had a microkernel... and was slower than Win95
  - This was later fixed by putting things back into the no-longer-micro kernel
  - WinXP is closer to monolithic than micro
  - This shows that OS developers constantly experiment, and you'll find OS people strongly disagreeing on OS structure

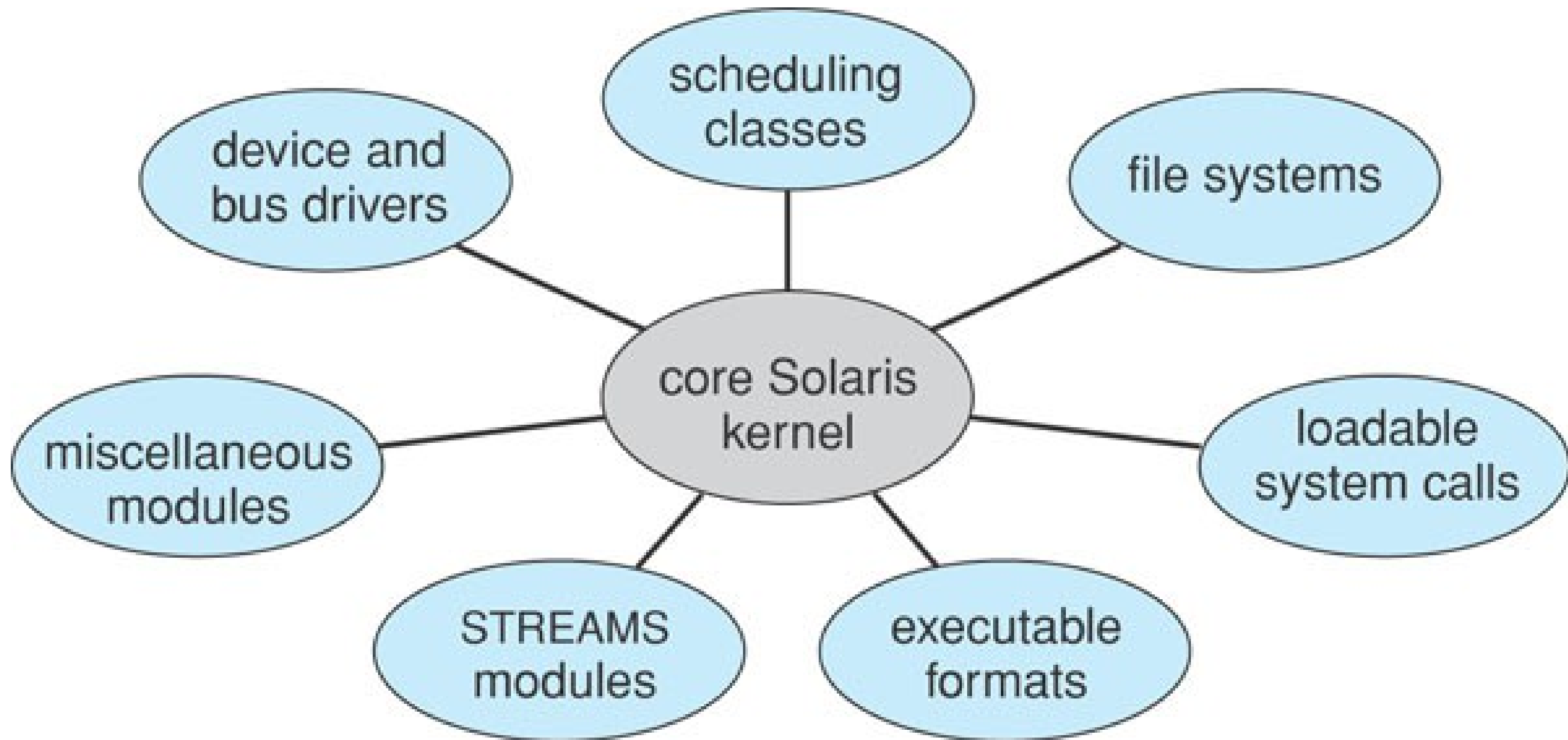
OS Structure: Microkernels



OS Structure: Modules

- Most modern OSes implement modules
  - Uses object-oriented approach
  - Each core component is separate
  - Each talks to the others over known interfaces
  - Each is loadable as needed within the kernel
- Loadable modules can be loaded at boot time or at runtime
- Like a layered interface, since each module has its own interface
- But a module can talk to any other module, so it's like a microkernel
- But communication is not done via message passing since modules are actually loaded into the kernel
- Bottom line:
  - Design has advantage of microkernels
  - Without the overhead problem

Solaris

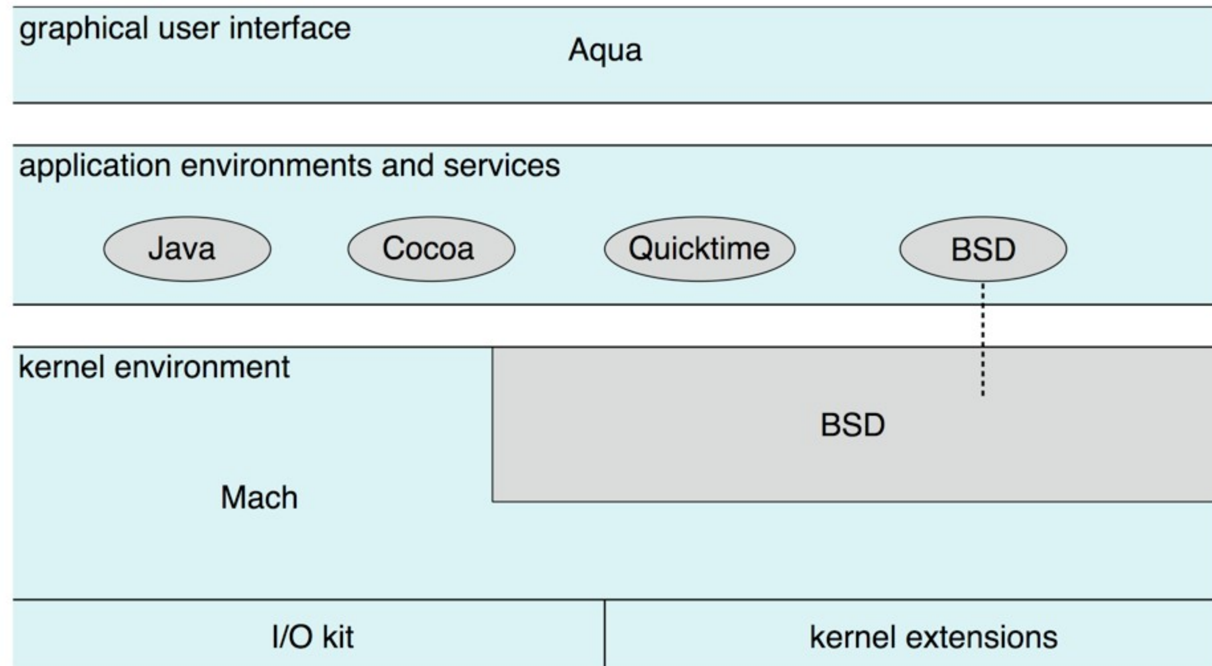


- 7 default modules
- Others can be added on the fly

Hybrid systems

- Very few modern OSes adhere strictly to one of these designs
- Instead, they try to take grab the best features of multiple design ideas
- Typical approach:
  - Don’t stray too far away from monolithic, so as to have good performance
  - Most OSes provide the notion of modules
- The book gives three examples
  - Mac OSX, iOS, Android

Mac OS X



- Hybrid structure: two kernel layers
  - Mach: Memory management, Remote Procedure Calls, Inter-Process Communication, Thread Scheduling
    - One of the oldest micro-kernels
  - BSD: Implements all POSIX services (file system, networking, I/O, dynamically loadable modules, etc.)
- I/O kit: used to develop device drivers (see later)
- Kernel extensions: loadable modules

OS Debugging

- OS debugging is hard
  - The kernel is complex and does many hardware things

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"

  - A crash dump (like a core dump, but for the kernel, since kernel failure leads to a crash) can be generated
  - But a kernel bug in the file system makes generating a crash dump difficult
    - One possibility: use a special disk area to write crash dump data
    - Upon reboot, crash dump data is written to a file in the file system

OS Debugging

- Kernel debugging isn’t as much a dark art as it used to be: we have some tools
- DTrace tool in Solaris, FreeBSD, Mac OS X (SystemTap for Linux) allows live instrumentation on production systems
  - Probes fire when code is executed, capturing state data
  - Section 2.8.3 has many details
- In fact, there are simple command-line tools to do basic tracing of system calls:
  - On a Linux system, we’ve seen strace
  - On Max OS X, Solaris there’s truss/dtruss
  - On a Windows system: ProcessMonitor (not built-in) / Core OS Tools for Win 7+ (not sure if available in all Win)

OS Boot

- So how does this thing start anyway?
- The first thing to do is to load the kernel into memory, which is called booting
- When the computer is powered on, the instruction pointer is loaded with a particular address, and execution starts there
- At that address is a program called the bootstrap loader
- Like all “firmware”, it is stored in ROM
  - Initially, RAM state is completely undefined
  - ROM is expensive, so the firmware had better be small
- The bootstrap loader runs diagnostic, initializes registers, memory, device controllers, etc.
  - e.g., all the stuff you see go by when you boot your a Linux machine

OS Boot

- At this point the OS is still stored on disk
- The bootstrap loader loads a disk block at a standard location (say block #0 on the disk) into RAM
- This boot block contains a program that knows how to load the OS
  - Or it knows how to load a more complex bootstrap program into RAM, which then knows how to load the OS
- A disk that has such a boot block is called a boot disk, a bootable disk, or a system disk (or a disk partition)
- The Kernel is then loaded into memory
- The bootstrap loader starts the Kernel
  - Which starts the ***init*** process and simply **waits** for events

Conclusion

- Reading Assignment:
  - Textbook, Chapter 2
  - Read Programming Project (page 96)
    - Adding a system call to Linux
    - And play around with it if you're into it (using VirtualBox to install Linux on your system)
    - lsmod, insmod, rmmod, dmesg