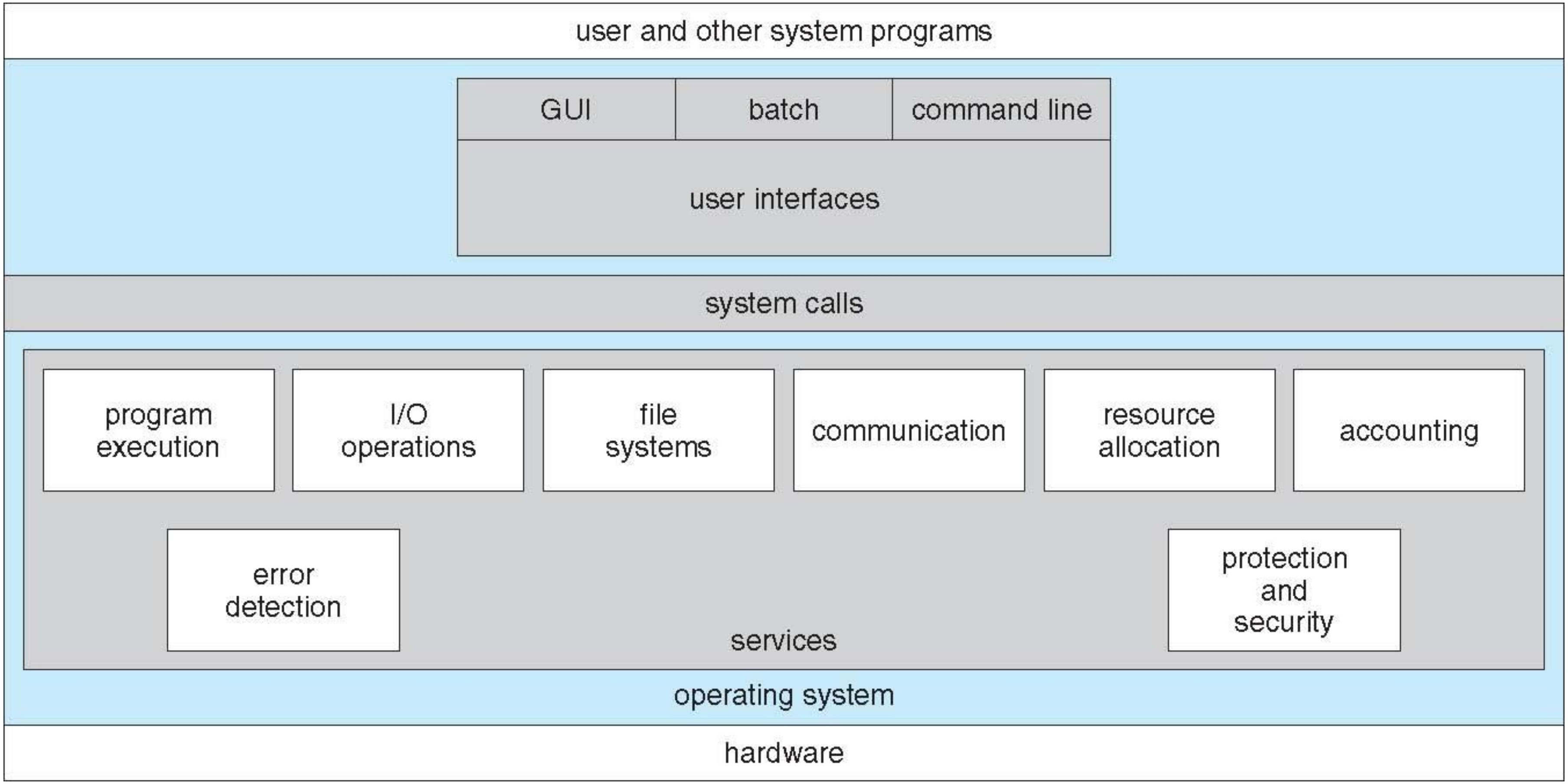
Operating Systems

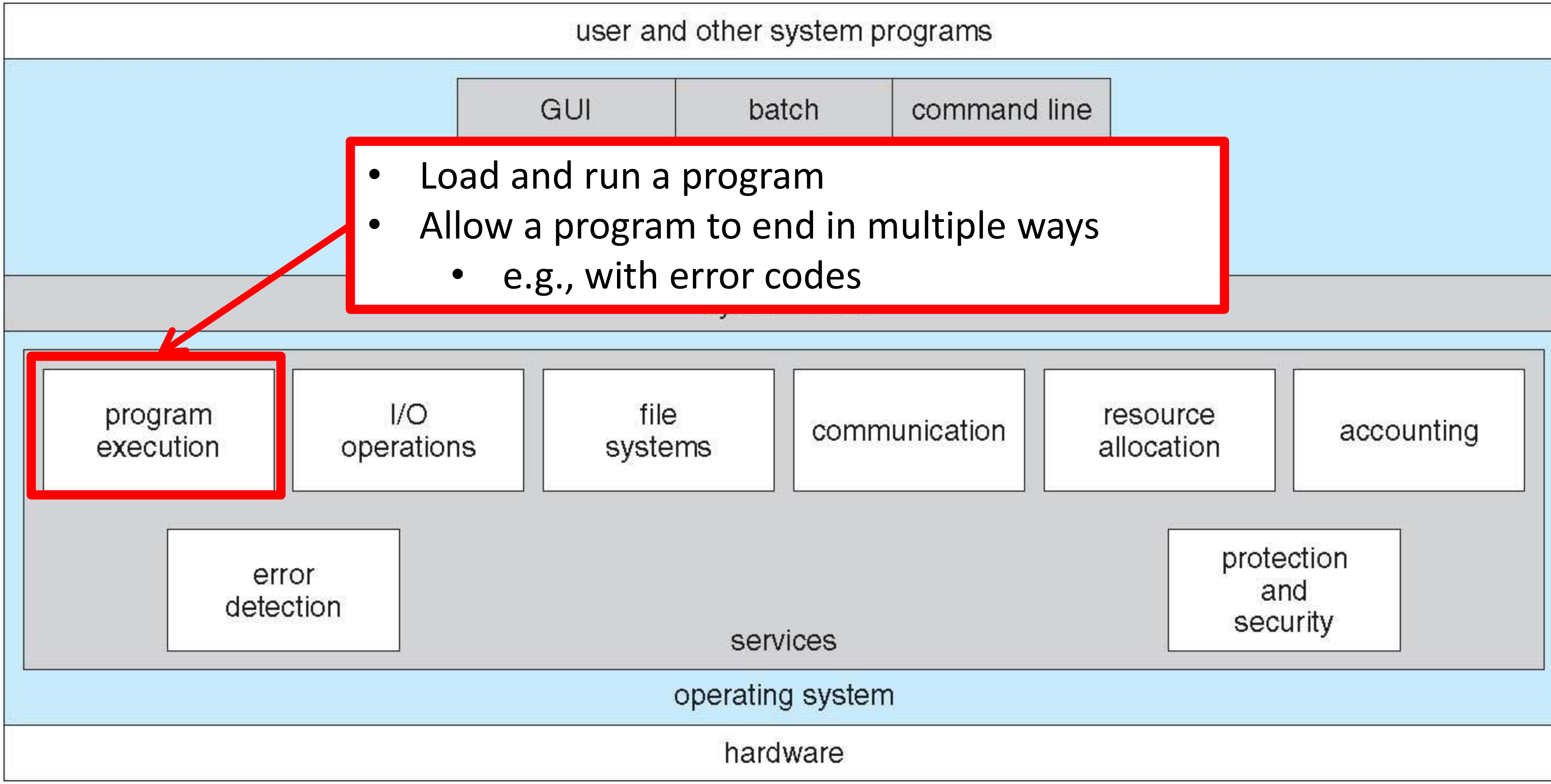
CSC-4320/6320 –Summer 2014

Dustin Kempton (dkempton1@cs.gsu.edu)

OS Services and Features

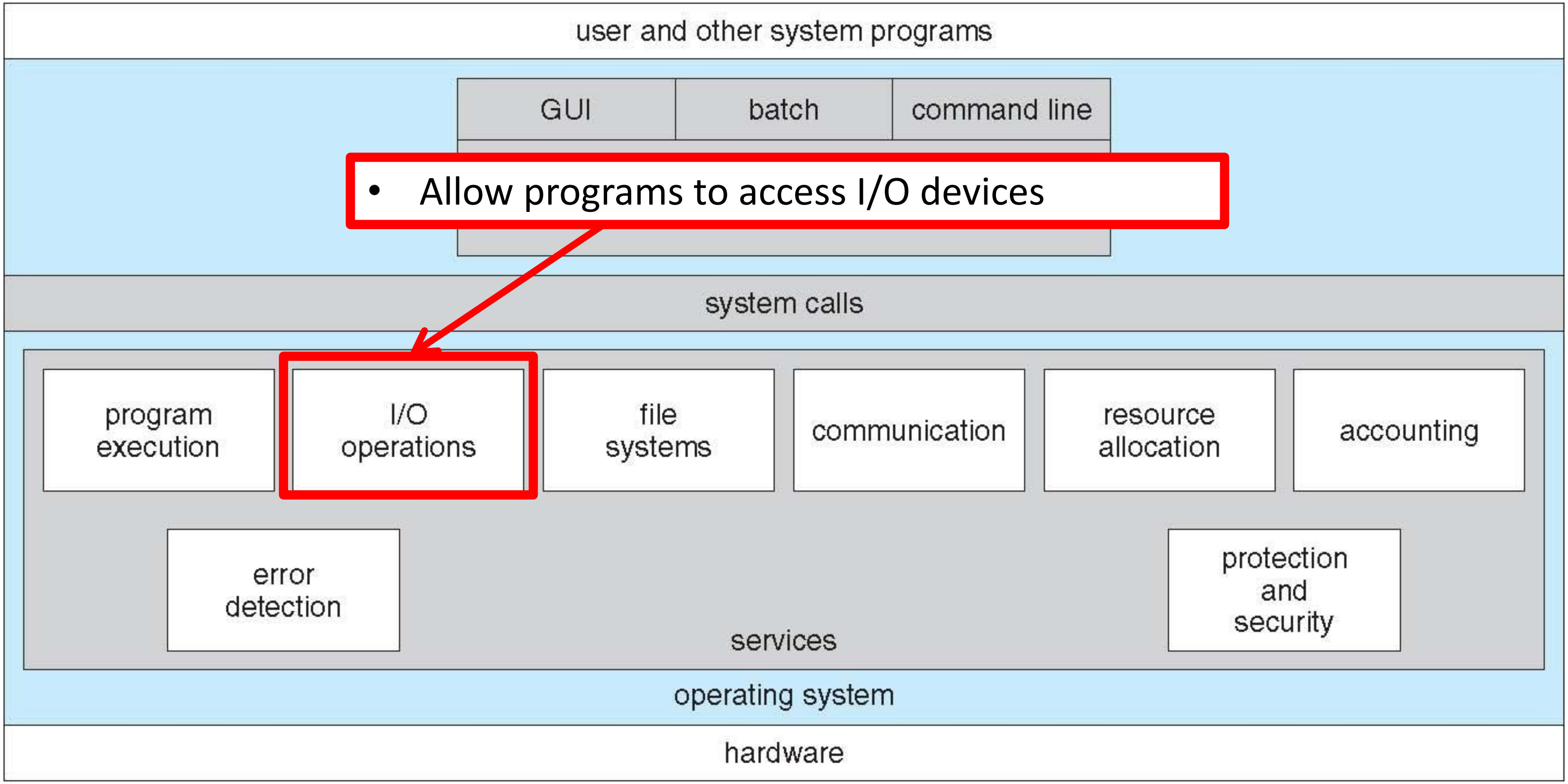


OS Services



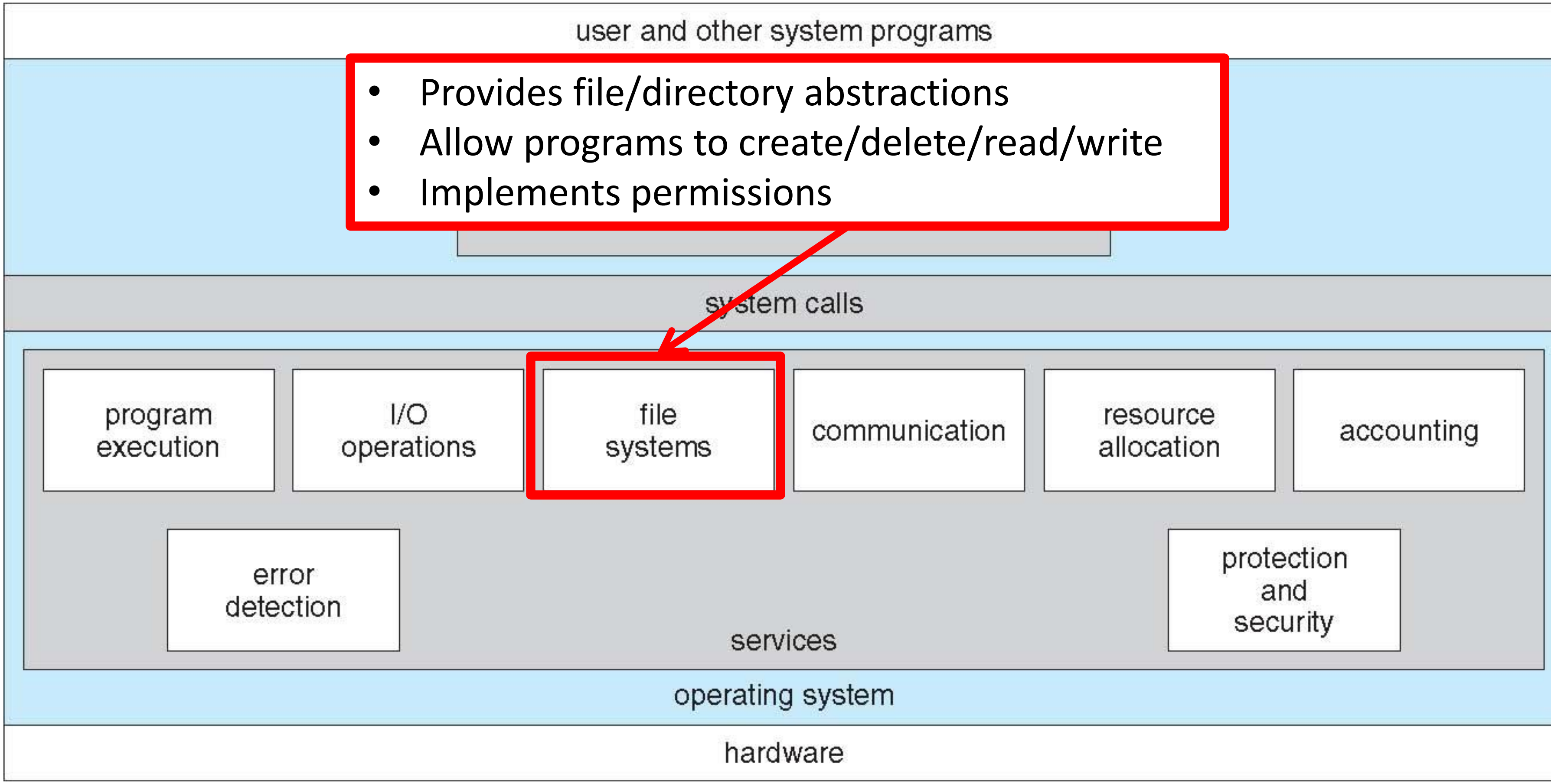
- Load and run a program
- Allow a program to end in multiple ways
  - e.g., with error codes

OS Services

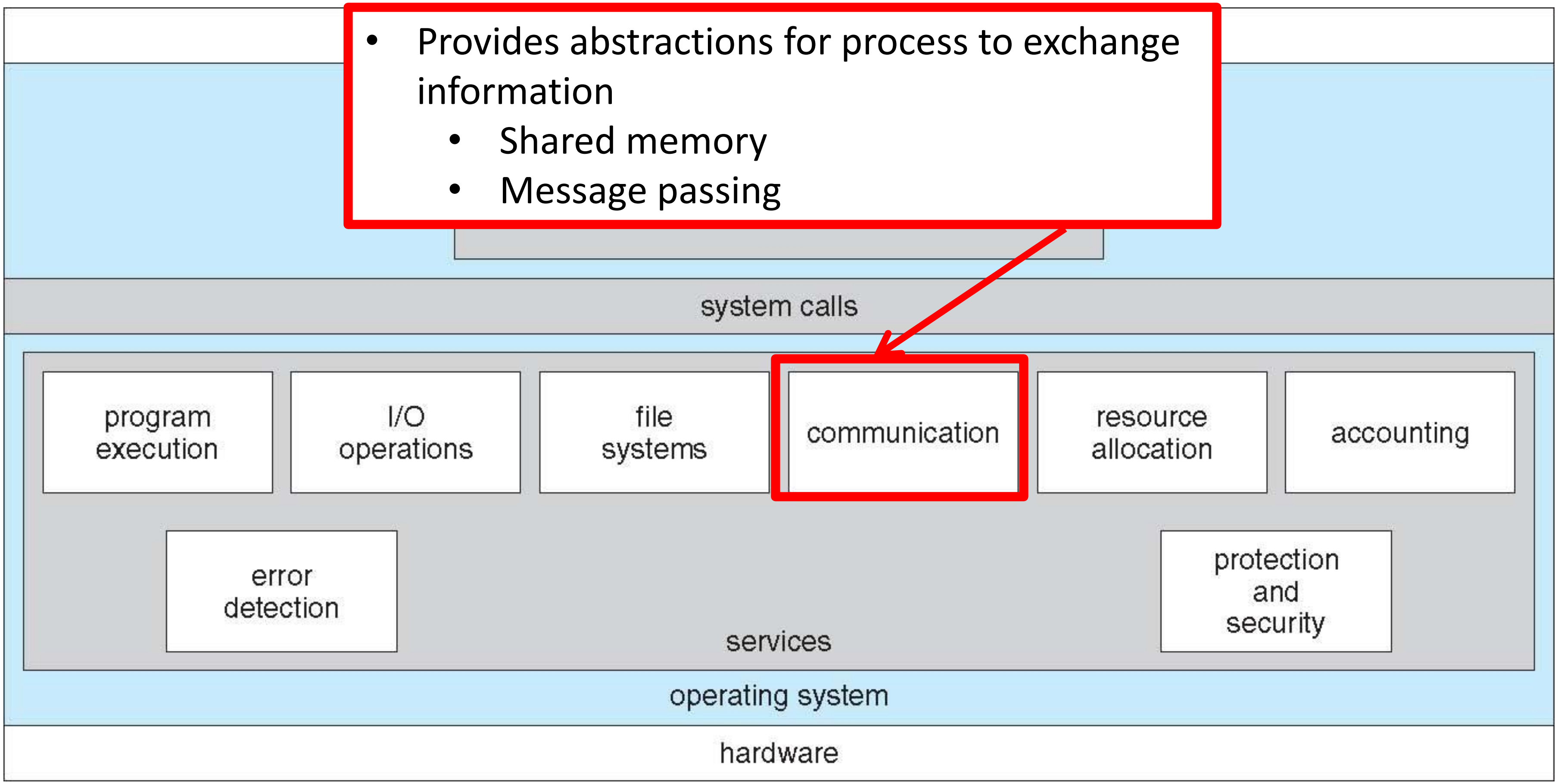

user and other system programs
GUI
batch
command line
• Allow programs to access I/O devices
system calls
program execution
I/O operations
file systems
communication
resource allocation
accounting
error detection
protection and security
services
operating system
hardware

OS Services

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



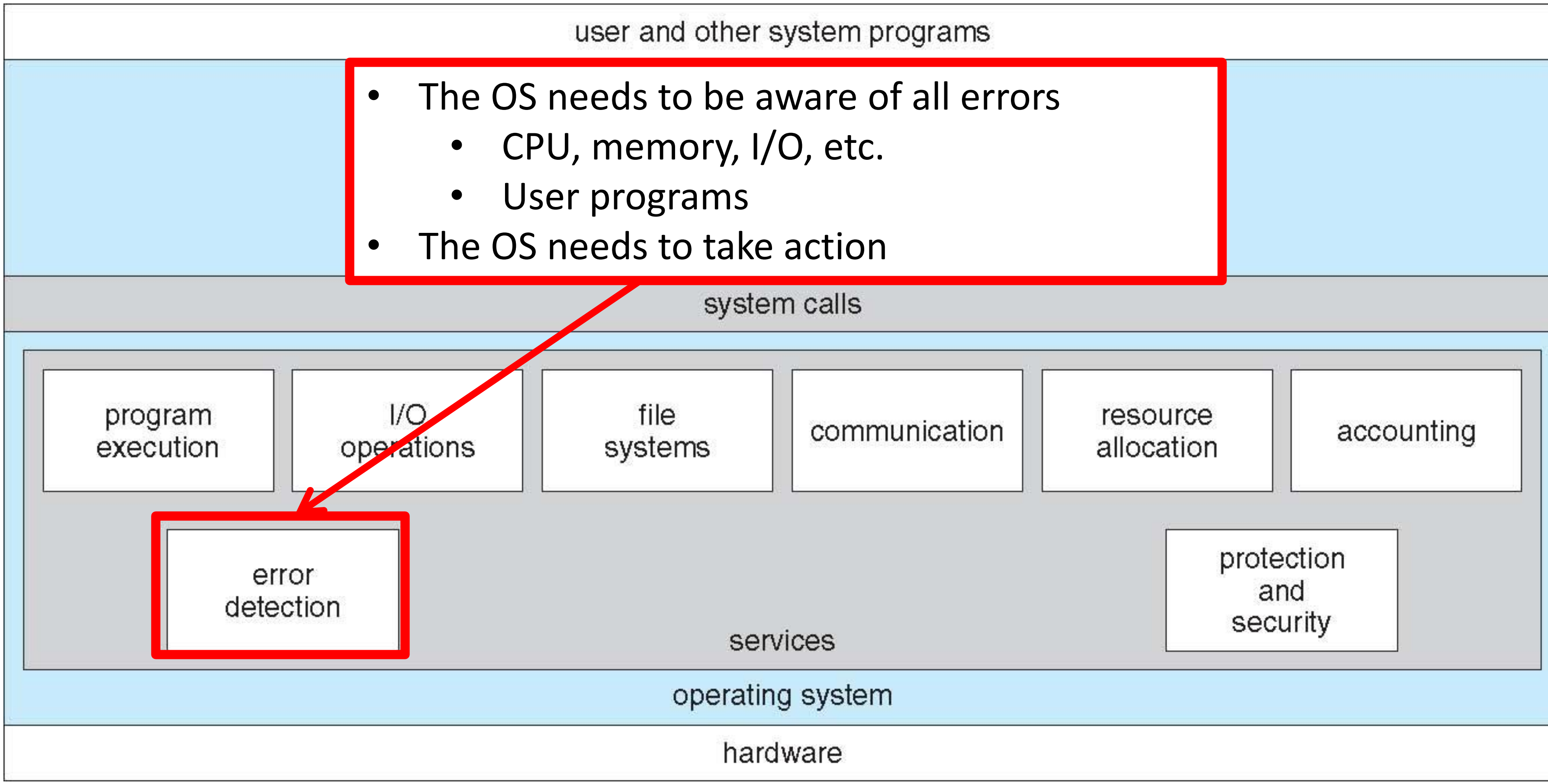
OS Services



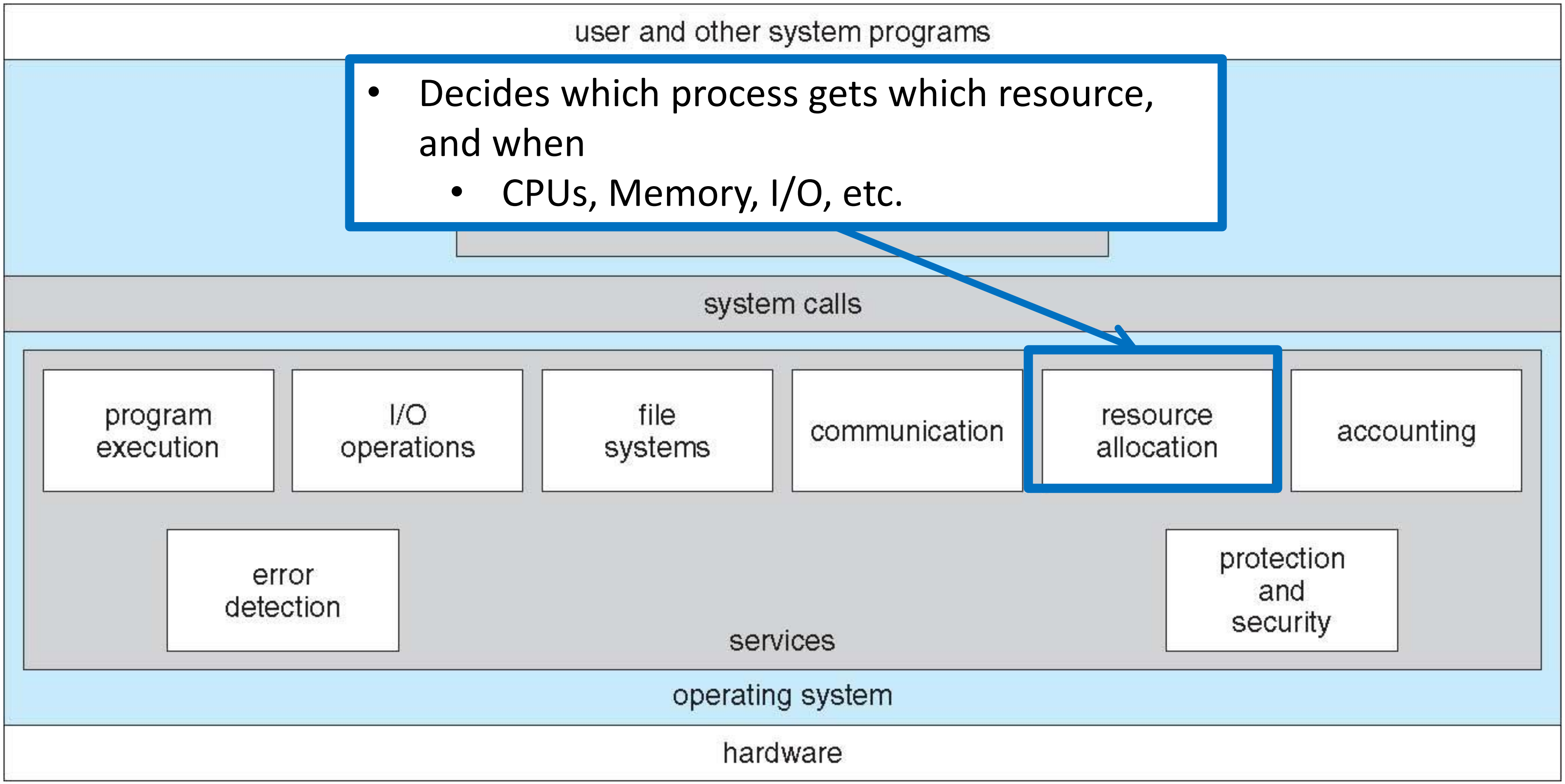
- Provides abstractions for process to exchange information
  - Shared memory
  - Message passing
system calls
program execution
I/O operations
file systems
communication
resource allocation
accounting
error detection
protection and security
services
operating system
hardware

OS Services

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



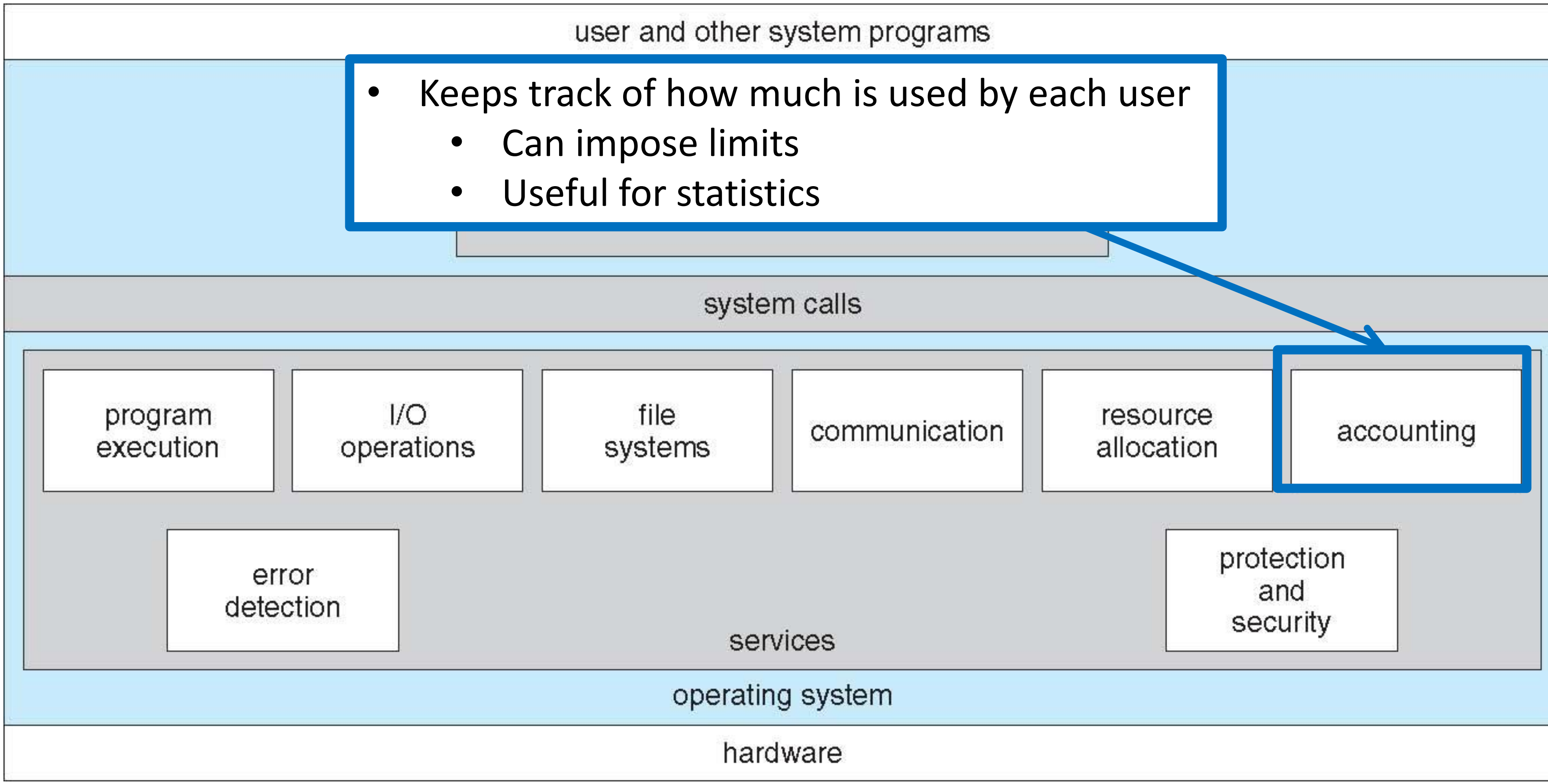
OS Features



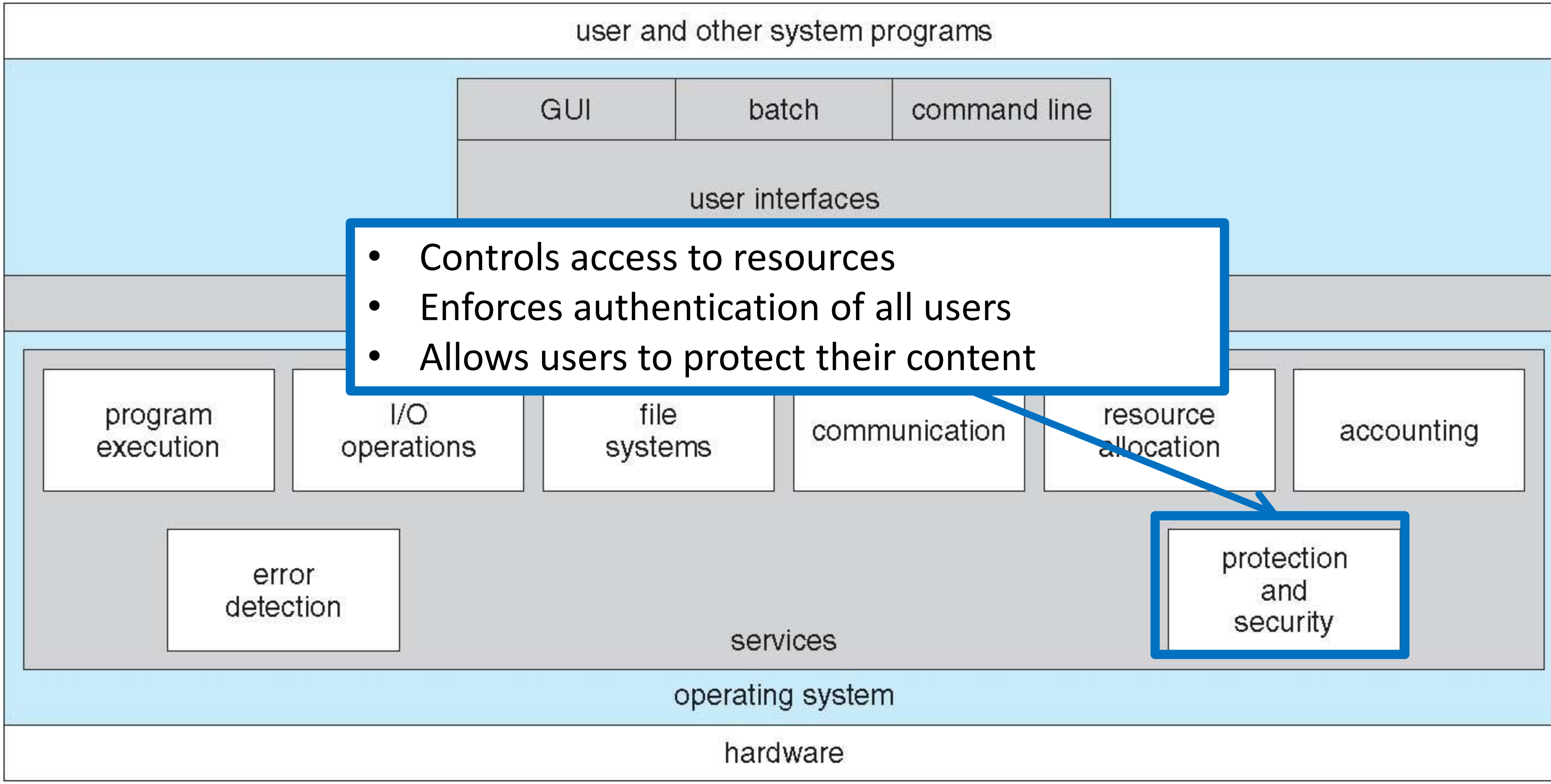
- Decides which process gets which resource, and when
  - CPUs, Memory, I/O, etc.

OS Features

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

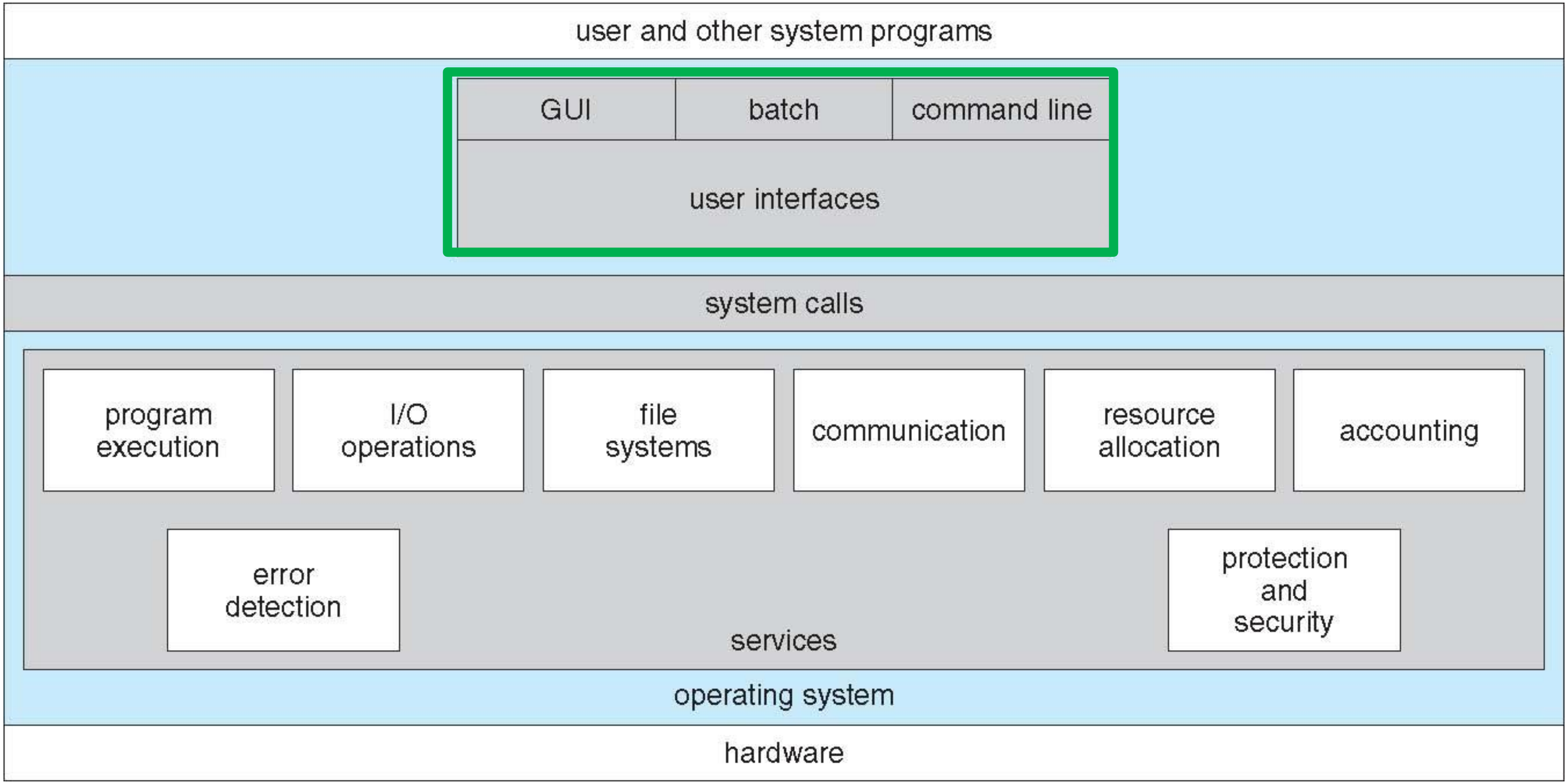


OS Features



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

OS Services and Features



User Interface

- **Embedded systems**: special purpose buttons and displays
- **Unix and batch systems**: command line interface (CLI)
  - Direct command entry
  - Fetch command and execute
  - Fast commands execute directly, others launch system programs
- **Windows and IOS**: Graphical User Interface (GUI)
  - Point and click: mouse, keyboard, and monitor
  - Touch: finger movement triggers actions
  - Icon based: files, programs, actions , etc show as icons
  - Various types of mouse clicks respond accordingly
- **Hybrids**: Both CLI and GUI components
  - Microsoft Windows is GUI with CLI “command” shell
  - Apple Mac OS X is GUI interface with various UNIX shells
  - Solaris is CI with optional GUIs (Java Desktop, KDE)

OS Interfaces: The Shell (CLI)

- 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
  - You can see them all by doing a “man bash” (search for the last occurrence of “BUILTIN”)
- The shell cannot implement all commands (i.e., contain their code)
  - This would make the shell a huge program
  - 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 the 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 it’s 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

System Programs

- Some system programs are simple wrappers around system calls (we will talk about them 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 are applications
  - Do you think of the javac compiler as a command, an application or 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 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 to not provide a command-line interface, but Mac OS X does: Terminal
- Question: is the GUI part of the OS or not?

System Calls

- System calls are the (lowest-level) interface to the 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)
- On Linux there is a “command” called strace that gives details about which system calls were made by a program during execution
  - dtrus on Mac OSX is a rough equivalent
  - strace can be “attached” to a running program to find out why it is stuck

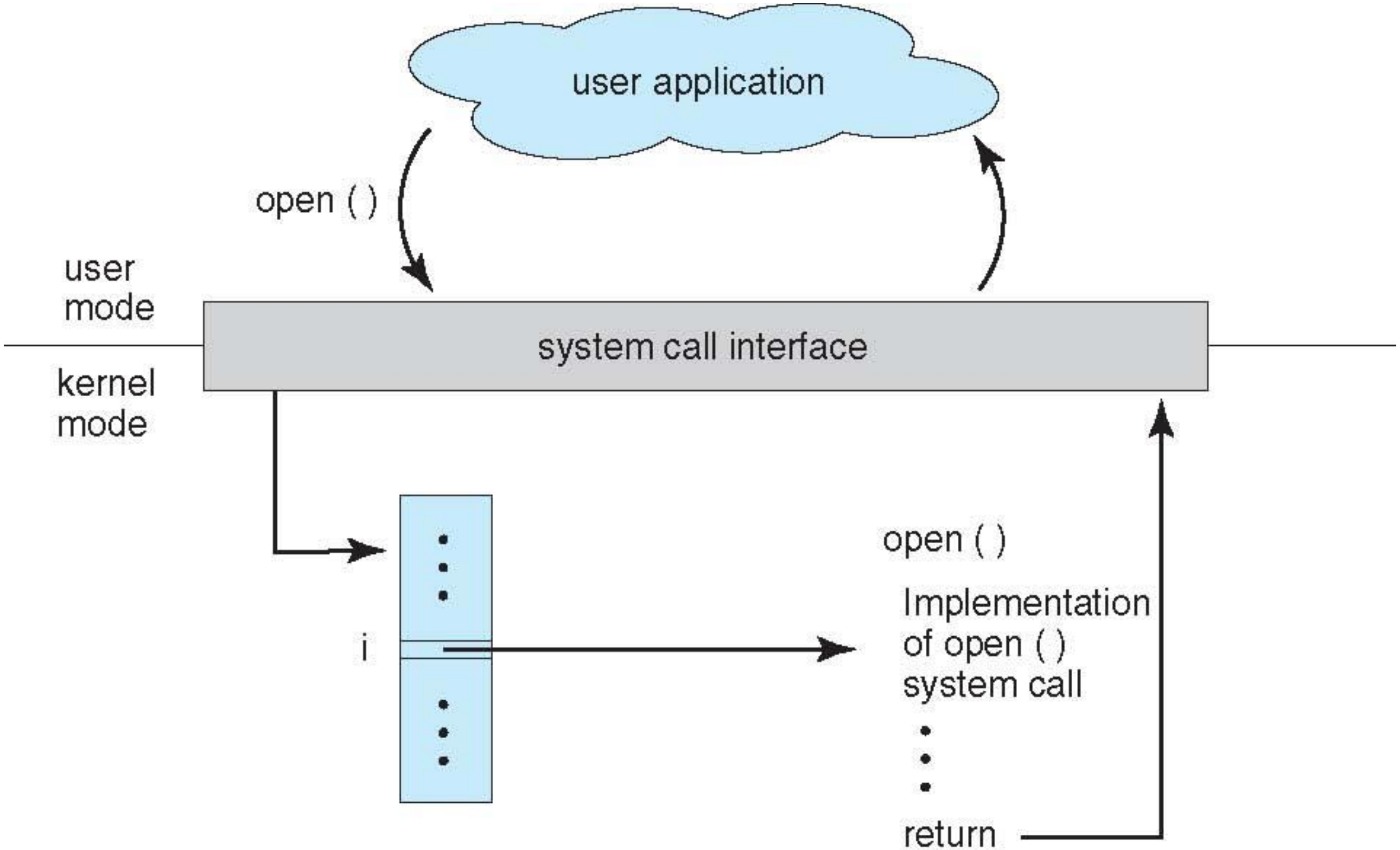
Time Spent in System Calls?

- 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)

APIs

- System calls are mostly accessed by programs via a high-level Application Program Interface (API)
  - API functions can call (multiple) system calls
  - API calls are often simpler than full-fledged 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
  - Win32 API for Windows
  - POSIX for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X)
  - The Java API, which provides API to the Java Virtual Machine (JVM), which has OS-like functionality on top of the OS

System Call-OS Application API

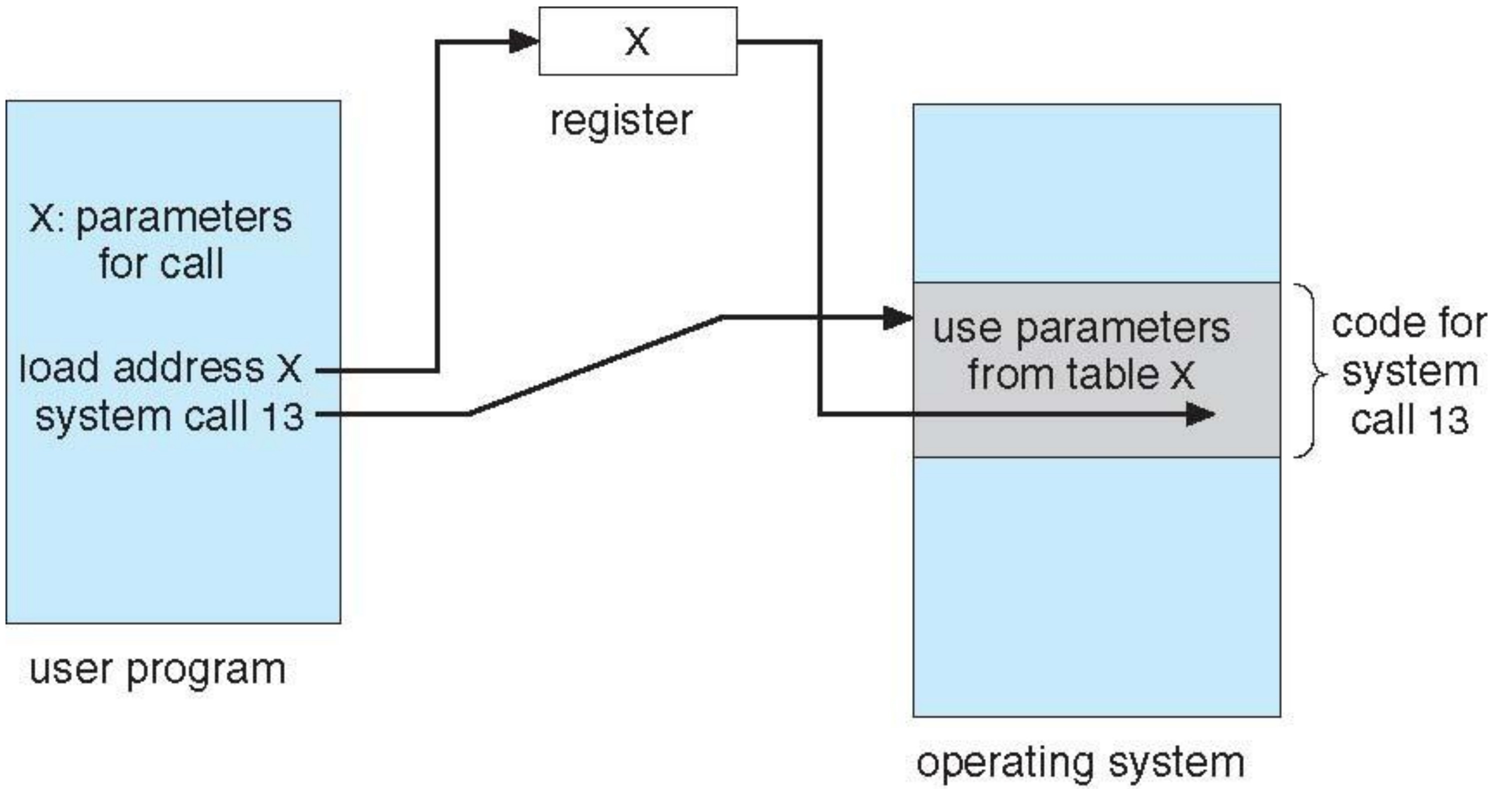


- The OS implements a table of available system calls, each with a well-defined interface and supported by compiler run-time libraries
- User programs invoke calls to run-time libraries, which execute privileged instructions and traps to the OS kernel.

System Call arguments

1. First approach: Pass in hardware registers
   Disadvantage: There may not be enough registers
2. Second approach: Pass in a register addressed array
   Advantage: There is no argument limit
   Note: Linux and Solaris use this approach
3. Push parameters onto a special stack
   Advantage: most flexible
   Disadvantage: loss of efficiency

Parameter Passing via Table



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

Mechanisms and Policies

- One ubiquitous principle: separating mechanisms and policies
  - Policy: what should be done
  - Mechanism: how should it 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

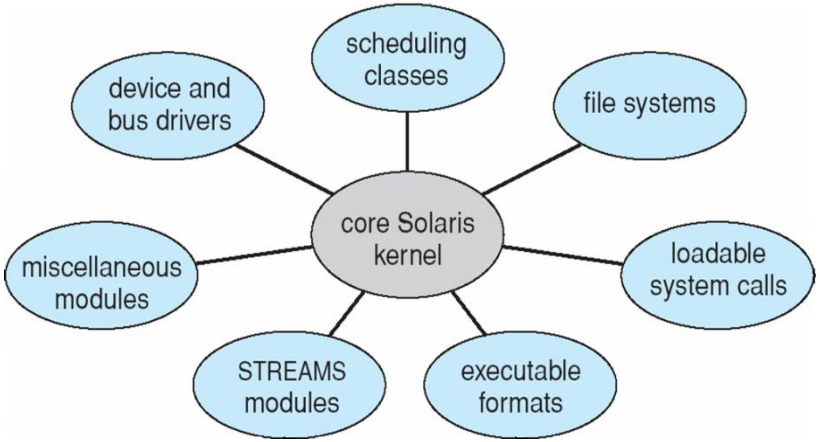
OS Implementation

- OSes used to be written in assembly
  - MS-DOS was written all in assembly (would not like to have been those developers)
- Modern OSes are written in languages like C or C++, with a dash of assembly here and there
  - Linux and 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)

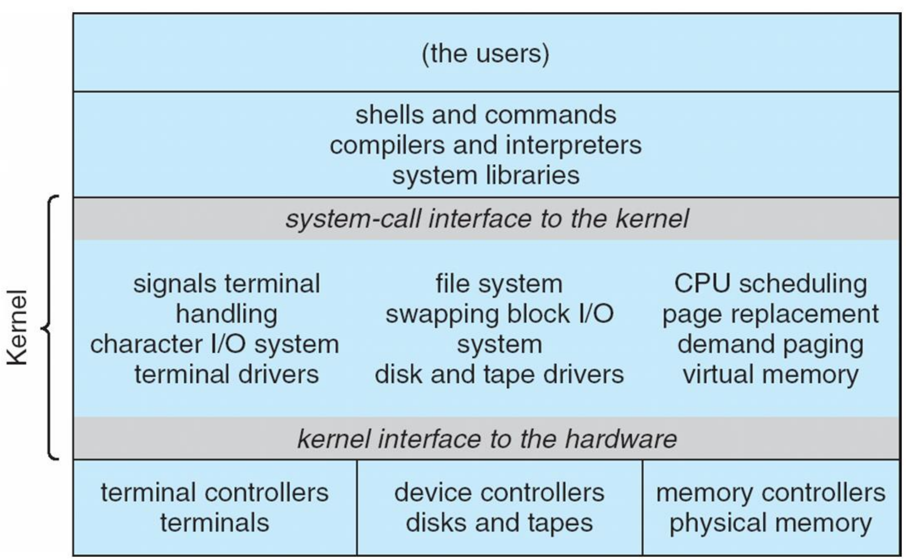
Operating System Structure

- The General-purpose OS is a very large program
- Various ways to structure the OS
  - Simple structure-one piece of code, which is monolithic, non-modular, and unprotected (MSDOS)
  - More Complex but also more maintainable
    - Layered design: Lower levels represent services from higher levels
      - Advantage: Easy testing and replacing layer implementations
      - Less efficient: multiple levels of system calls from layer to layer
      - Difficulty: no clear cut way to assign system functions to layers
    - Modular: separate object oriented modules with well-defined interfaces
    - Micro kernel
      - Perform as many functions as possible in user mode
      - Communication using a message passing paradigm
      - Easier to extend and port: is more reliable and secure
      - Degraded performance: inefficient user to kernel communication

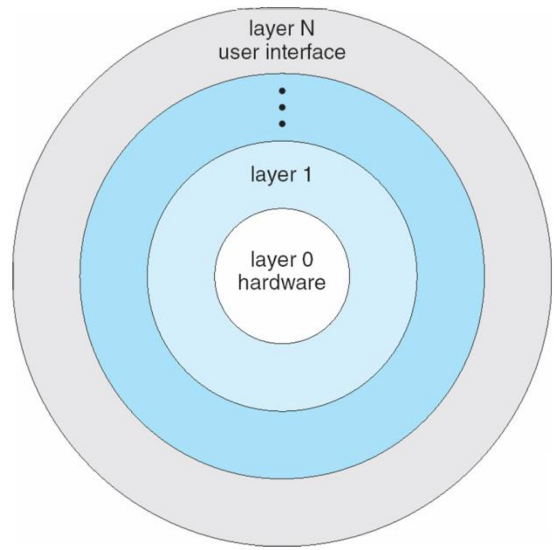
OS Organization Examples



Solaris Kernel Modules



Unix Kernel

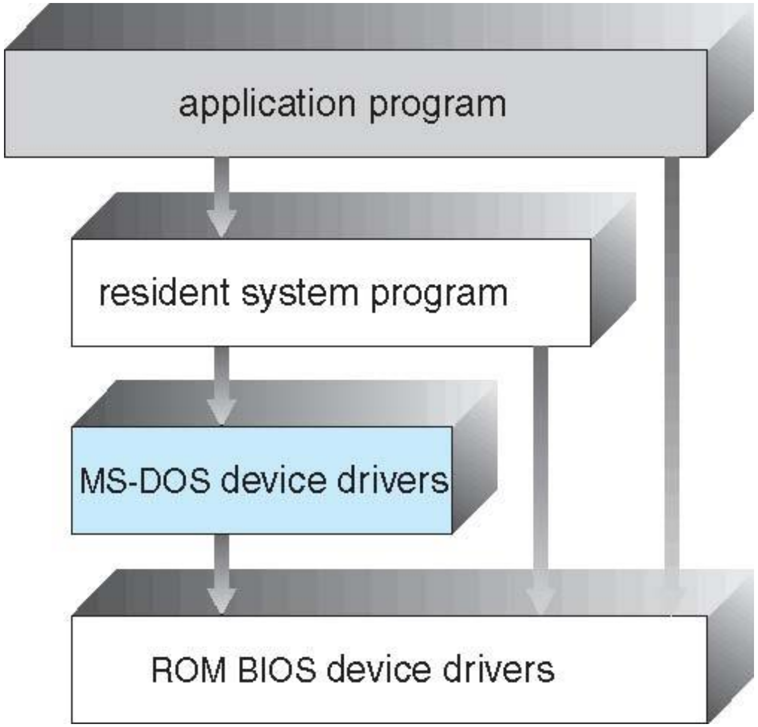


Layered Approach

- Each layer built on top of lower layer
- The bottom layer (layer 0), is hardware
- The highest (layer N) is user interface
- Layers use services of lower-level layers
- Easy to debug, and test replacement layers

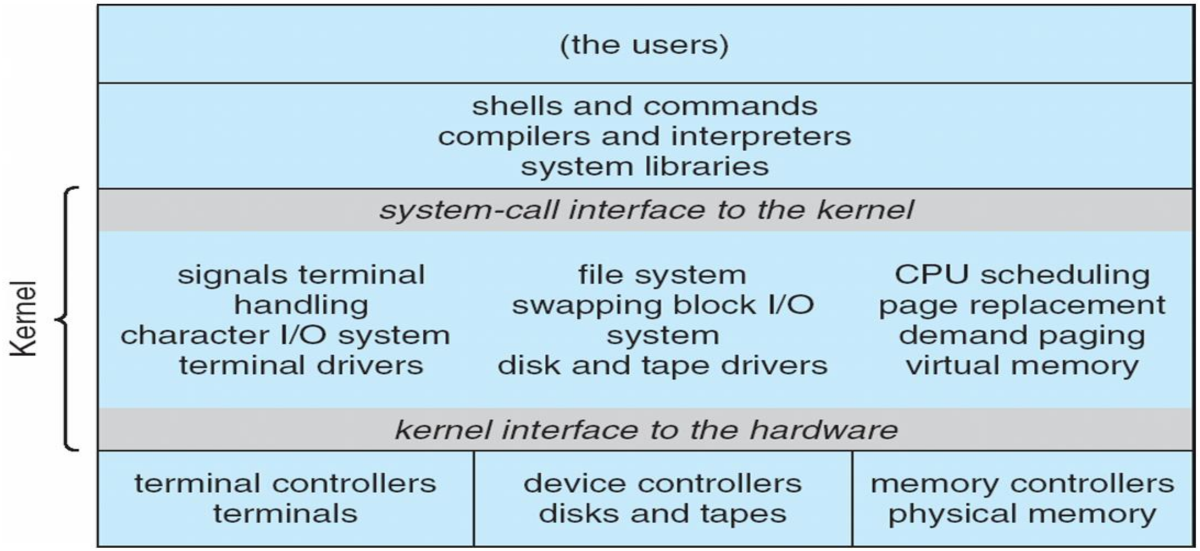
OS Structure: Simple

- Early operating systems 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, and security
  - User programs could 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



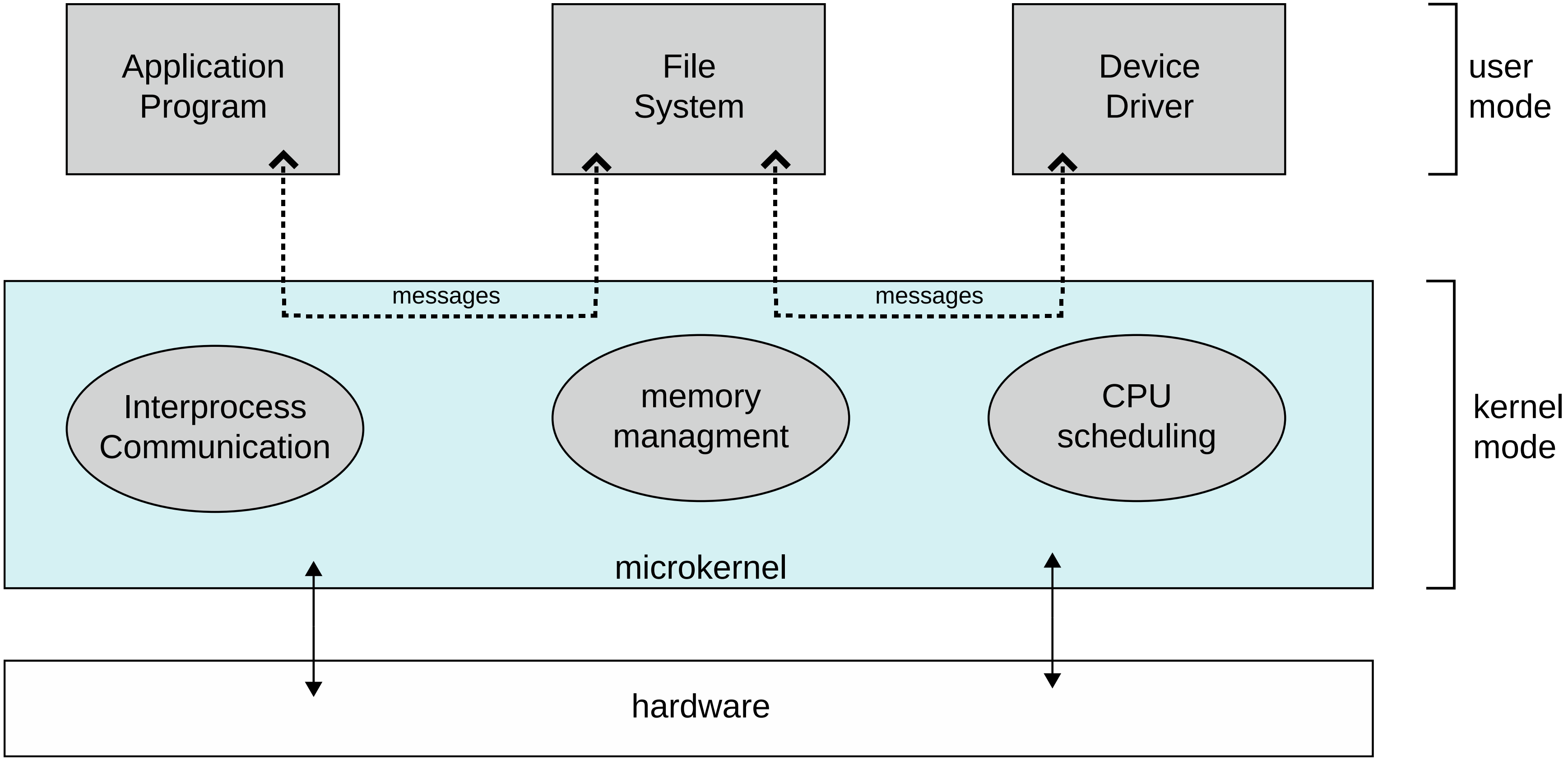
OS Structure: Layered

- 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 (however more stable which is why it was the choice in office/industrial settings, among other reasons)
  - This was later fixed by putting things back into the no-longer-micro kernel
  - WinXP is closer to monolithic than micro
  - This shows that we constantly experiment, and you'll find OS people storngly disagreeing on OS structure

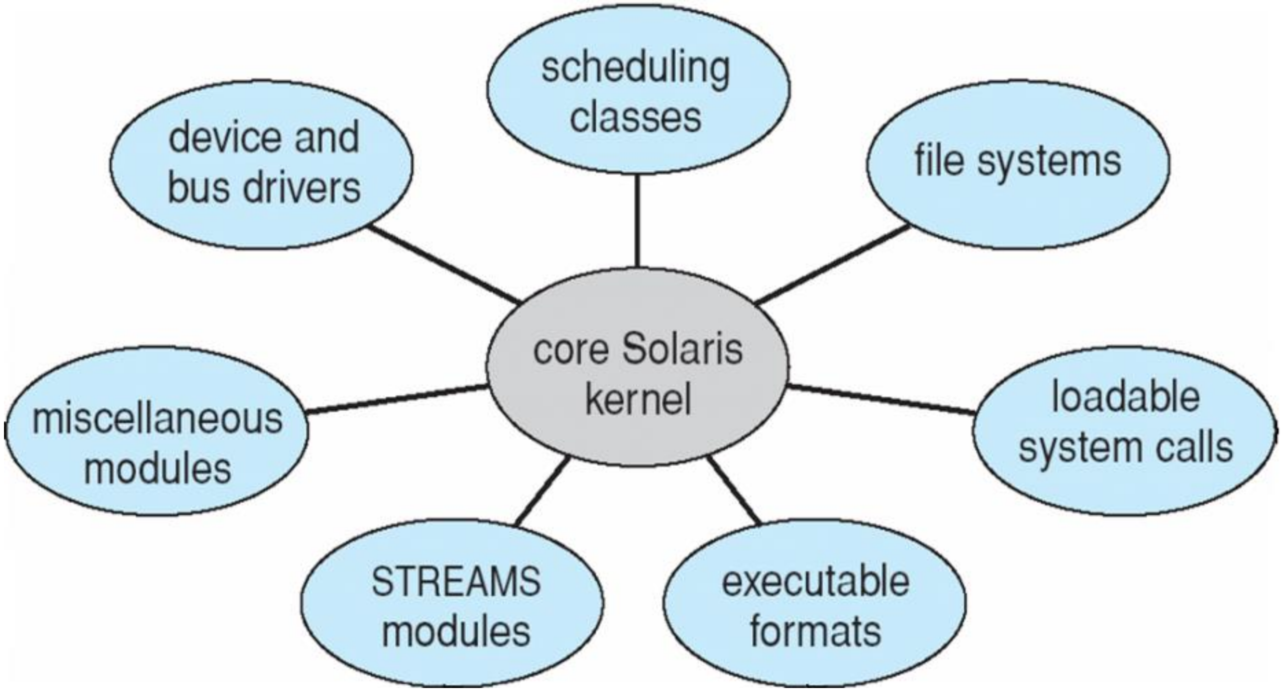
OS Structure: Microkernels



OS Structure: Modules

- Most modern operating systems implement modules
  - Uses object-oriented approaches
  - Each core component is separate
  - Each talks to the other 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 advantages of microkernels
  - Without the overhead problem

Solaris



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

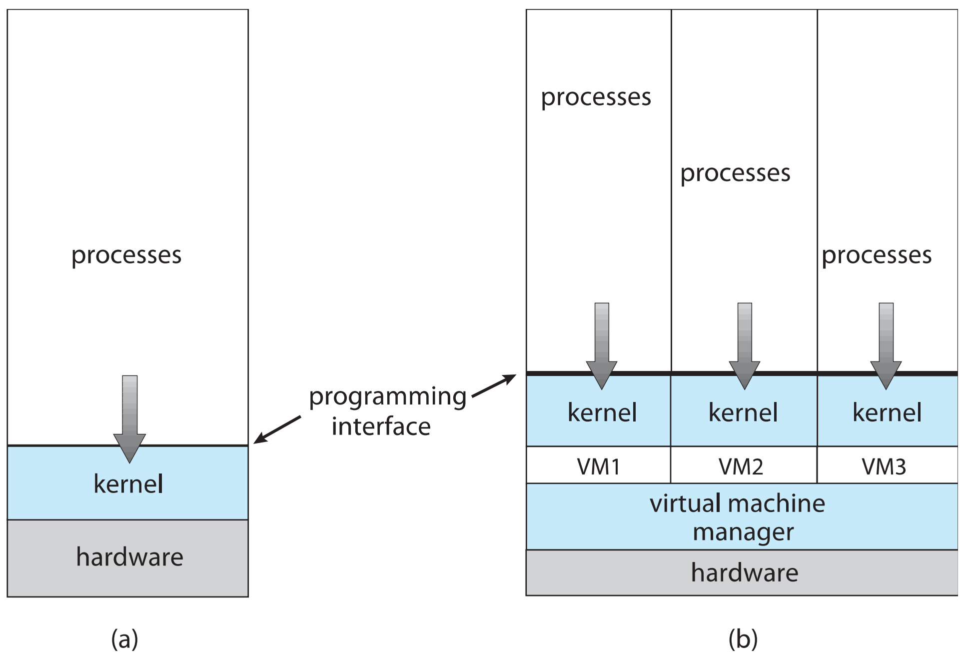
Hybrid systems

- Very few modern OSes adhere strictly to one of these designs
- Instead, they try to take the best features of multiple design ideas
- Typical approaches:
  - 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

Virtual Machines

- Virtual machines provide
  - API identical to bare hardware
  - Allocate portions of disk to each virtual machine (a file as a virtual hdd)
  - Spooling to virtual print devices
  - Workstation virtual console
  - Combine software emulation or hardware virtualization
- Advantages:
  - Operating systems research
  - Cross-platform testing
- Disadvantages:
  - No direct sharing of resources between virtual machines
  - Significant loss of performance (though less when hardware provides mode bits for virtualization)

The operating system has the illusion of multiple processes executing on their own bare-hardware processor



Conclusion

- Reading Assignment:
  - Chapter 2 (should have already)
  - Read Programming Project (page 96)
    - Adding a system call to Linux
    - Play around with it if you're into it (using VirtualBox to install Linux on your system)
- "Programing" Assignment #1…