Operating Systems 2017/2018 | Part I – Basic Concepts

Ricardo Rocha

Department of Computer Science

Faculty of Sciences

University of Porto

Slides based on the book

'Operating System Concepts, 9th Edition,

Abraham Silberschatz, Peter B. Galvin and Greg Gagne, Wiley'

Chapters 1 and 2

What is an Operating System?

- Motivation
  - Hardware alone is not particularly easy to use, thus some kid of software is required
  - Different software programs usually require common operations to operate the hardware resources
  - A good idea would be to brought together into **one piece of software** the **common functions of controlling and allocating hardware resources**
  - This common piece of software can be seen as the **operating system**

What is an Operating System?

- How can we then define an operating system or what is part of an operating system?
  - Sorry, but **no universally accepted definition**
  - A good approximation is ***"Everything a vendor ships when you order an operating system"***, but varies wildly
  - A more restricted definition is ***"The one program running at all times on the computer"***, everything else is either a system program (ships with the operating system) or an application program

- An operating system is what **manages a computer's hardware**
  - Acts as an intermediary between the user and the hardware
  - Provides a basis for application programs
  - Some operating systems are designed to make the computer system **convenient to use**, others are designed to use the computer hardware in an **efficient manner**, and others to be some **combination of the two**

What Operating Systems Should Do?

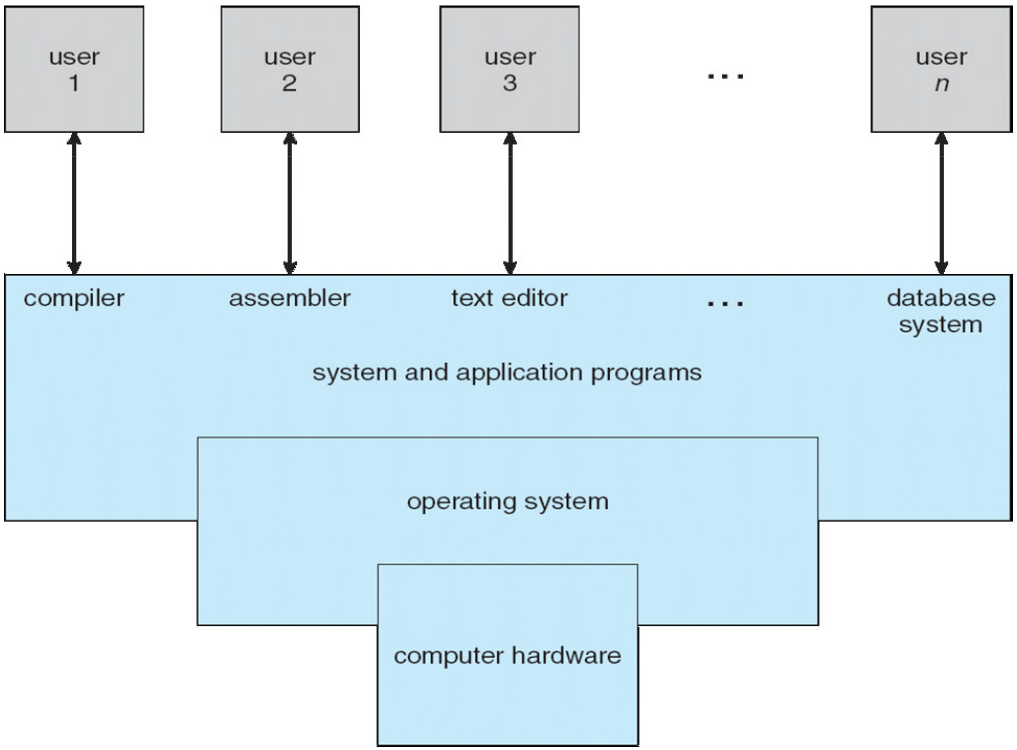
- Shared computers, such as **mainframes, workstations and/or servers**, must **keep all users happy**
  - The operating system is designed to maximize resource utilization and to assure that all available CPU time, memory, and I/O are used efficiently and fairly between all users
- **PCs** are **optimized for the single-user experience** rather than the requirements of multiple users
  - The operating system is designed mostly for ease of use, with some attention paid to performance and almost none paid to resource utilization

What Operating Systems Should Do?

- Mobile computers, such as **smartphones and tablets**, are **optimized for usability and battery life**
  - The operating system is designed mostly for ease of use, with particular attention paid to resource utilization
- Many computers, such as **embedded computers in home devices and automobiles**, have **little or no user interface**
  - The operating system is designed to run with no/minimal user intervention

Computer System Structure

- A computer system can be divided into four main components:
  - **Hardware** – the basic system resources (CPU, memory, I/O devices, …)
  - **Operating system** – controls and coordinates use of hardware among various applications and users
  - **Programs** – define the way in which the system resources are used to solve the users' needs (word processors, web browsers, database systems, video games, compilers, …)
  - **Users** (people, other programs/computers)



Operating System Principles

- Operating system as a **resource allocator/controller**
  - Acts as a manager of all resources: CPU time, memory space, I/O devices, …
  - Decides between conflicting requests for efficient and fair resource use
  - Prevents errors and improper use of the computer
  - Especially important where many users access the same resources
- Operating system as a **facilitator**
  - Provides facilities/services that everyone needs
  - Make application programming easier, faster, less error-prone
  - Especially concerned with the operation of various programs
- Most features **reflect both principles**
  - For example, file system is needed by everyone (facilitator) but must be efficiently used and protected (controller)

Operating System Major Components

- Modern operating system usually include the following major components:
  - Process management
  - Memory management
  - Storage management
  - I/O management

Process Management

- Main activities
  - Creating, suspending, resuming and terminating (user/system) processes
  - Providing mechanisms for process communication
  - Providing mechanisms for process synchronization
  - Providing mechanisms for deadlock handling

Memory Management

- Main activities
  - Allocating and deallocating memory space as needed
  - Tracking which parts of memory are currently being used and by whom
  - Deciding which processes/data to move into and out of memory and when

Storage Management

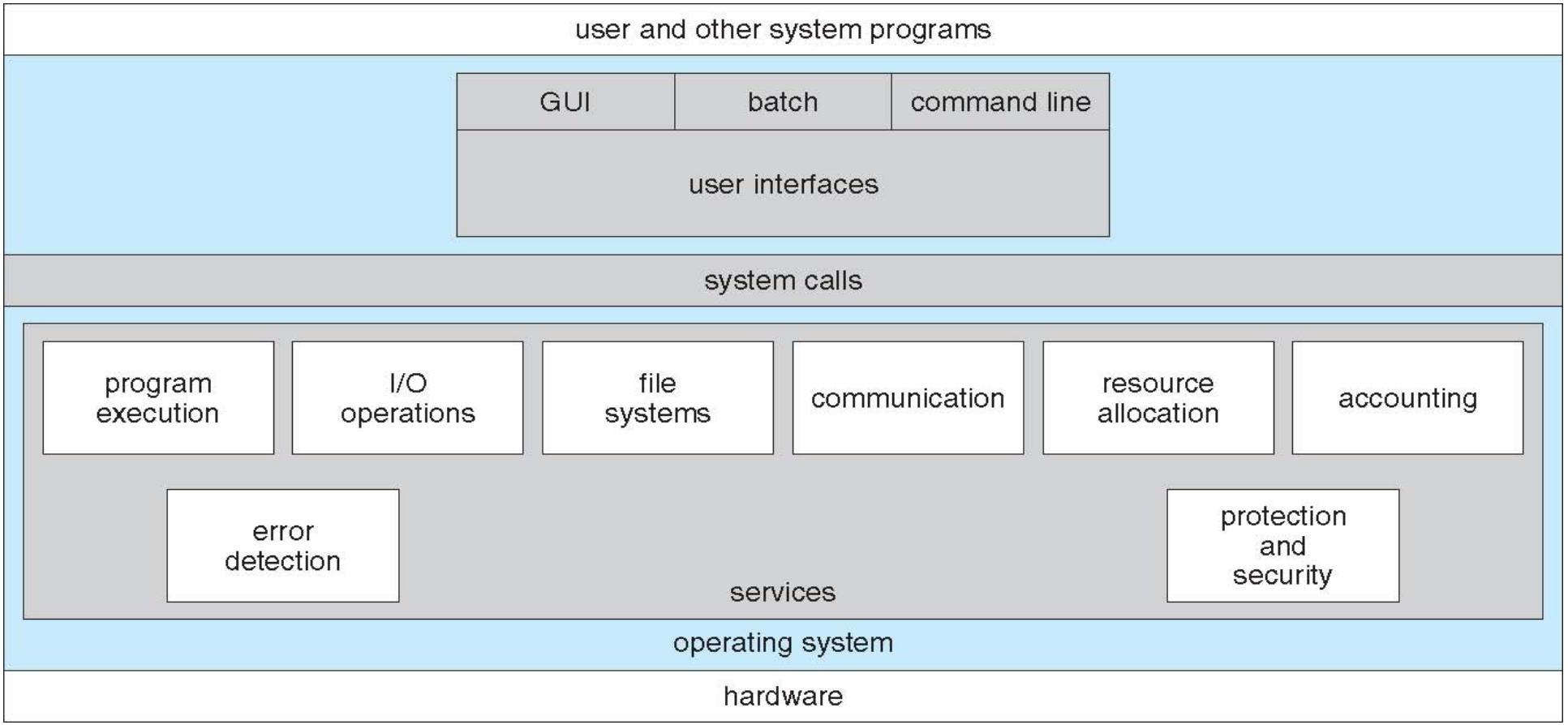
- Main activities
  - Providing uniform, logical view of information storage (abstracts physical properties into files and directories logical view)
  - Supporting primitives to create, delete and manipulate files and directories
  - Supporting access control policies to determine who can access what
  - Mapping/backing up files onto nonvolatile secondary storage media

I/O Management

- Main activities
  - Hide peculiarities of hardware devices from the user
    - General device-driver interface
    - Drivers for specific hardware devices
  - Responsible for memory management of I/O including:
    - Buffering – storing data temporarily while it is being transferred
    - Caching – storing parts of data in faster storage for performance
    - Spooling – the overlapping of output of one job with input of other jobs

Operating System Common Services

- Operating systems provide an environment for the execution of programs and, for that, they provide specific services to programs and users



Operating System Common Services

- The services provided differ from one operating system to another, but we can identify common classes:
  - **User interfaces** – to allow effective operation and control of the system
  - **Program execution** – to load a program into memory and to run that program
  - **I/O operations** – to provide a means to do I/O operations
  - **File systems** – to allow effective manipulation of files and directories
  - **Communications** – to allow exchange of information between processes on the same computer or between computers over a network
  - **Error detection** – to be constantly aware of possible errors that may occur in the CPU/ memory hardware, in I/O devices or in user programs in order to take the appropriate action to ensure correct and consistent computing

Operating System Common Services

- Another set of services exists not for helping the user but rather for ensuring the efficient operation of the system itself:
  - **Resource allocation** – when multiple processes are running concurrently, the available resources (such as CPU cycles, main memory, file storage, I/O devices) must be efficiently allocated to each of them
  - **Accounting** – to keep track of which users use how much and what kinds of computer resources
  - **Protection and security** – to avoid concurrent processes to interfere with each other or with the OS itself and to secure the system from outsiders

Multiprogramming

- One of the most important aspects of operating systems is the ability to have **multiple programs running**
- Multiprogramming **increases CPU utilization** by organizing jobs so that the **CPU can always execute one job**. The idea is as follows:
  - The operating system begins to execute one job via job scheduling
  - Eventually, the job may have to wait for some task, such as an I/O operation
  - In a non-multiprogrammed system, the CPU would sit idle
  - In a multiprogrammed system, the operating system switches to another job. When that job needs to wait, the CPU switches again to another job, and so on. Eventually, the first job finishes waiting and gets the CPU back.
  - As long as at least one job needs to execute, the CPU is never idle

Multitasking (or Time Sharing)

- Multiprogramming increases CPU utilization, but does not necessarily provide for user interaction with the computer system
- Multitasking (or time sharing) is a logical extension of multiprogramming that **increases response time** in which CPU switches jobs so frequently that **users can interact with each job while it is running**
  - Gives the impression that the entire computer system is dedicated to a user's use, even though it is being shared among many users
  - Time-sharing is based on the idea that while any single user would make inefficient use of a computer, a large group of users together would not
- Developing a system supporting multitasking was, at the time, a completely different concept: **the state of each user and their programs would have to be kept in the system and then switched between quickly** (response time should be < 1 second)

Computer Startup

- Bootstrap program, generally known as **firmware**, is loaded at power-up or reboot
  - Typically stored in read-only memory (ROM or EPROM)
  - Initializes all aspects of the system
  - Loads operating system kernel and starts its execution
- Once the kernel is loaded, it can start providing services to the users
  - Some services are provided outside of the kernel, by **system processes** that are loaded at boot time (on UNIX, the first system process is the *init* process that starts many other system processes)
- Once this phase is complete, the system is fully booted, and **starts waiting for some event to occur**

Interrupts

- The occurrence of an event is usually signaled by an **interrupt** from either the hardware or the software
  - Hardware devices may trigger an interrupt at any time by sending a **signal to the CPU** to communicate that they require attention from the OS
  - Software may trigger an interrupt by executing a special operation called a **system call**
- When the CPU is interrupted, it suspends the current activity, saving its state, and immediately transfers execution to a fixed function called an **interrupt handler** to deal with the event
  - An interrupt vector contains the addresses of all the interrupt service routines
- On completion, the CPU resumes the interrupted computation
  - Interrupt architecture must save the address of the interrupted instruction

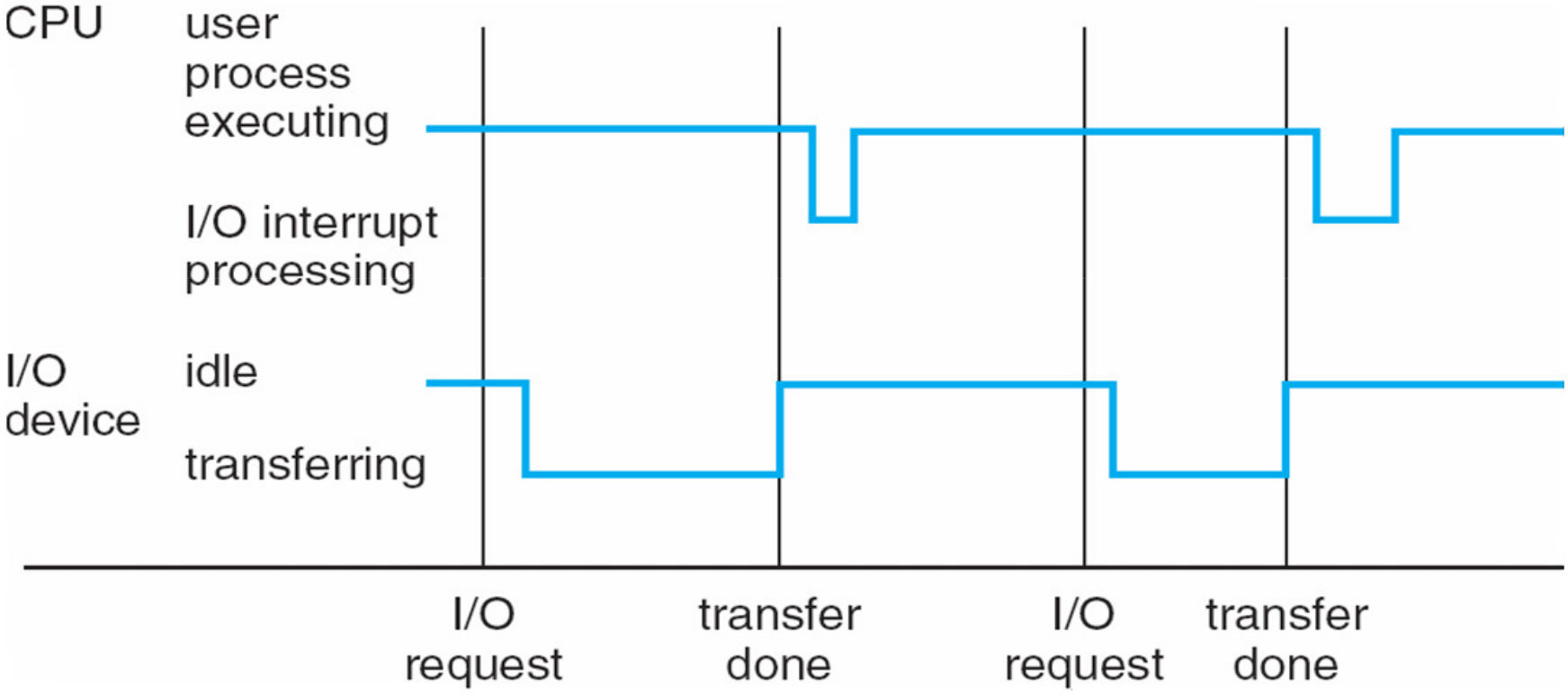
I/O Devices

- General-purpose computer systems consist of several **device controllers (hardware components)** that are connected through a common bus
  - Some examples are the USB controller, the PCI controller, the IDE controller, the SCSI controller, …
  - Each device controller is in charge of a specific type of device (for example, a SCSI controller can be in charge of 4 SCSI devices)
  - A device controller maintains some **local buffer storage** and a set of special-purpose registers
  - The device controller is responsible for moving data between the devices it controls and its local buffer storage
- To communicate with each device controller, operating systems require a specific **device driver (software component)**
  - The device driver understands the device controller and provides the rest of the operating system with a uniform interface to the device

I/O Devices

- To start an I/O operation, the device driver loads the appropriate registers within the device controller
  - The controller, in turn, examines the contents of these registers to determine what action to take (e.g., read a character from the keyboard)
  - The controller then starts the transfer of data from the device to its local buffer
- Once the I/O operation has completed, the device controller informs the device driver **via an interrupt**
  - The device driver then returns control to the operating system

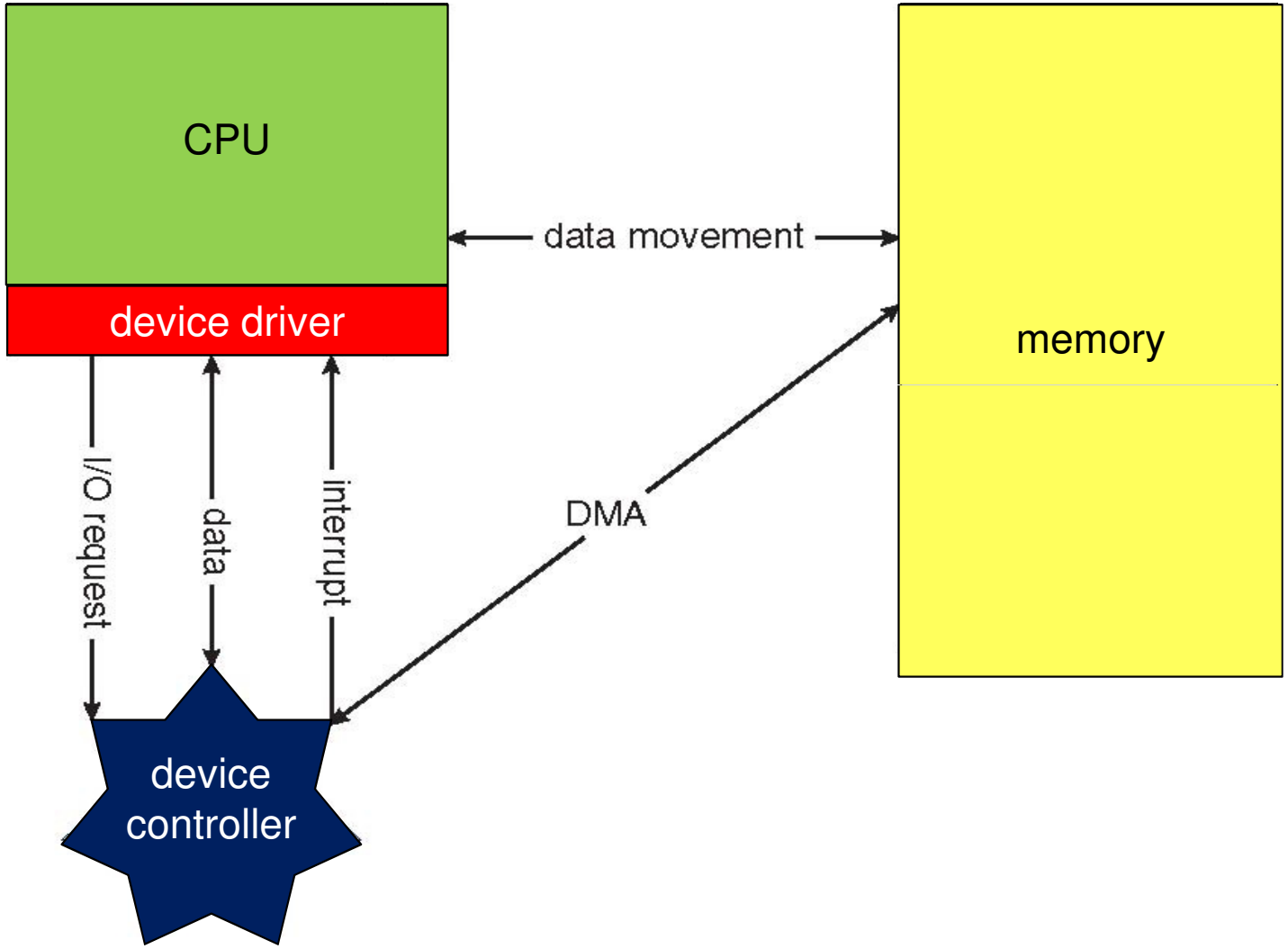
Interrupt Timeline



Direct Memory Access (DMA)

- This form of interrupt-driven I/O is fine for moving small amounts of data but can produce high overhead when used for bulk data movement such as disk I/O
- For high-speed I/O devices, able to transmit information at close to memory speeds, this problem is solved by using **direct memory access**
  - Examples of hardware devices that use DMA include disk drive controllers, graphics cards, network cards, sound cards, …
- Device controller transfers blocks of data from its own buffer storage directly to main memory, **without device driver (CPU) intervention**
  - Only one interrupt is generated per block, to tell the device driver that the operation has completed, rather than the one interrupt per byte generated for low-speed devices

Putting All Together



CPU Protection

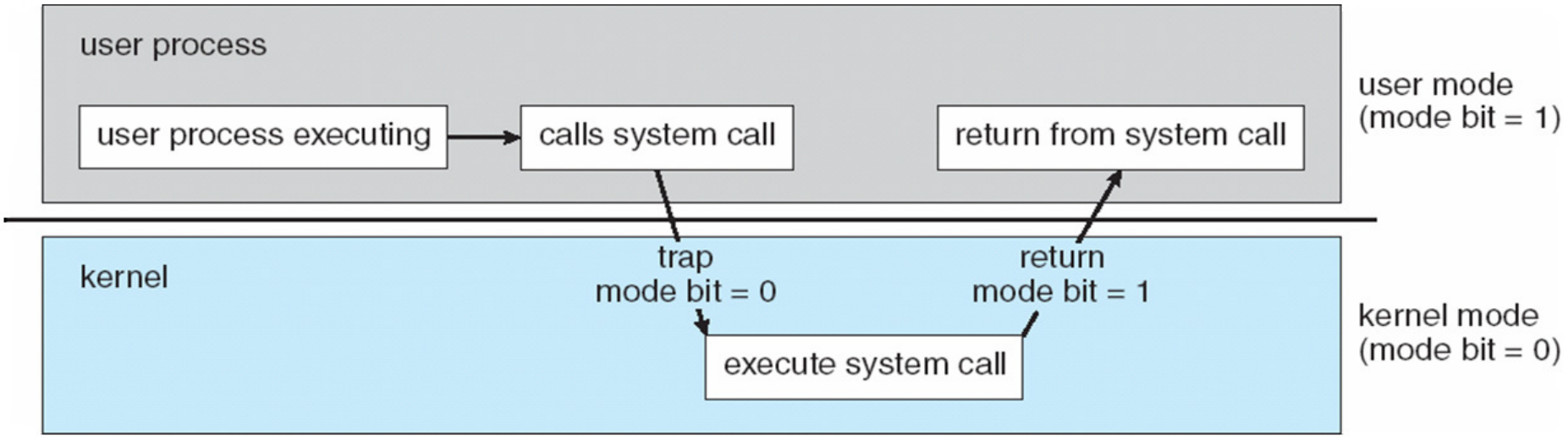
- We cannot allow a user program to get stuck or fail and never return control back to the operating system
  - To accomplish this goal, we can use a **timer**
- A timer can be set to interrupt the computer after a specified period
  - The operating system sets a counter
  - Every time the clock ticks, the counter is decremented
  - When the counter reaches 0, an interrupt occurs
- If the timer interrupts, control transfers automatically to the operating system, which may treat the interrupt as a fatal error or may give the program more time to continue executing
  - Before turning over control back to the user, the operating system ensures that the timer is set to interrupt

Dual-Mode Operation

- A properly designed operating system must ensure that an incorrect or malicious program cannot cause other programs to execute incorrectly
  - Examples include software errors (division by zero, infinite loops, …), processes modifying parts of others or even parts of the operating system
- The approach taken by most computer systems is to provide hardware support that allows us to differentiate among, at least, two separate modes of operation:
  - **User mode**
  - **Kernel mode** (also called **supervisor, system or privileged mode**)
- Dual-mode operation allows the operating system to protect itself and other system components

Dual-Mode Operation

- A **mode bit**, provided by the hardware, indicates the current mode:
  - Allows to distinguish when system is running user code or kernel code
  - Some instructions, designated as **privileged instructions**, are **only executable in kernel mode** (instructions for I/O control, timer management, interrupt management, …)
  - Interrupts or system calls change mode to kernel, return from interrupts or system calls reset it to user mode

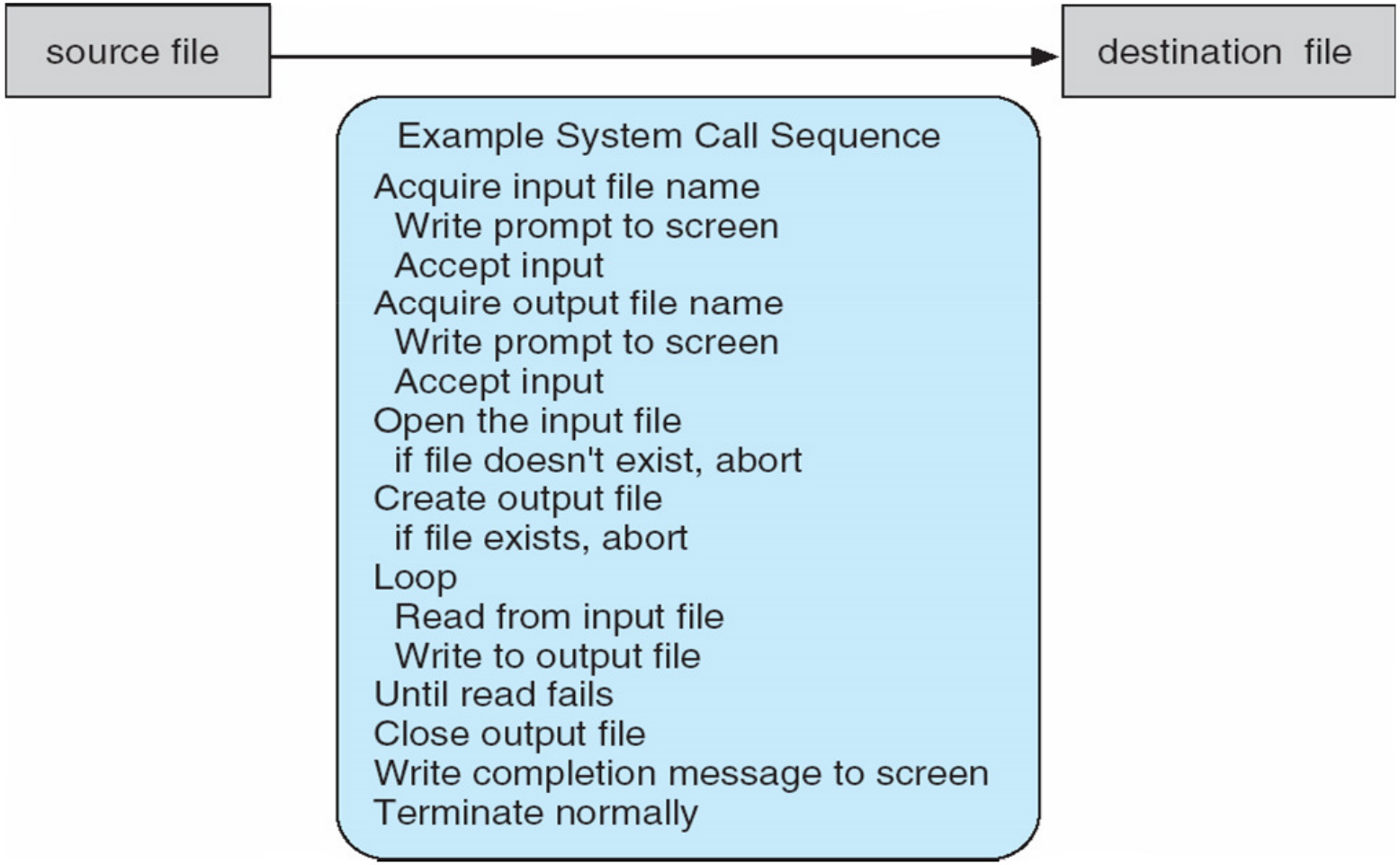


System Calls

- System calls provide an **interface to the operating system services**
  - Typically written in a high-level language (C or C++)
- System calls are mostly **accessed via a high-level Application Program Interface (API)** rather than direct system call use
  - Most common APIs are **Win32 API** for Windows, **POSIX API** for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X), and **Java API** for the Java virtual machine (JVM)
- Why use APIs rather than invoking directly the system calls?
  - Program portability, the same program must compile and run on any system that supports the same API
  - System calls can be more detailed and difficult to work with than the API available to an application programmer (nevertheless, many of the POSIX and Windows APIs are similar to the native system calls provided by the OS)

Example of System Calls

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

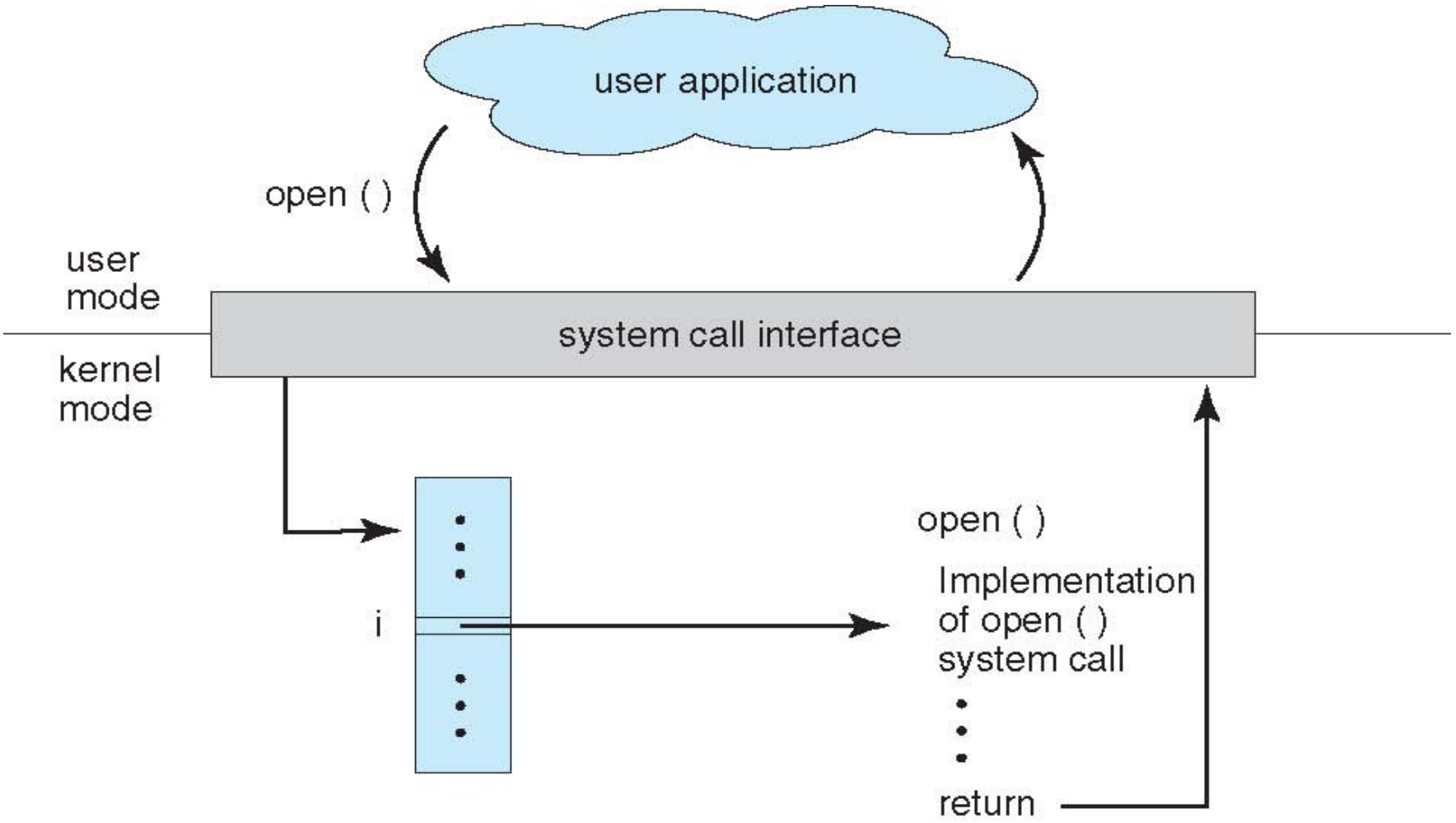


System Call Implementation

- Typically, a **number is associated with each system call** and the system call interface maintains a table indexed according to these numbers

- The system call interface invokes the intended system call in the OS kernel and returns status of the system call and any return values
  - The caller need to know nothing about how the system call is implemented
  - Just needs to obey API and understand what OS will do as a result
  - Most details of OS interface are hidden from the programmer by the API

System Call Implementation



Examples of Types of System Calls

- System calls can be grouped roughly into six major categories:
  - Process control
  - File manipulation
  - Device manipulation
  - Information maintenance
  - Communication
  - Protection

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