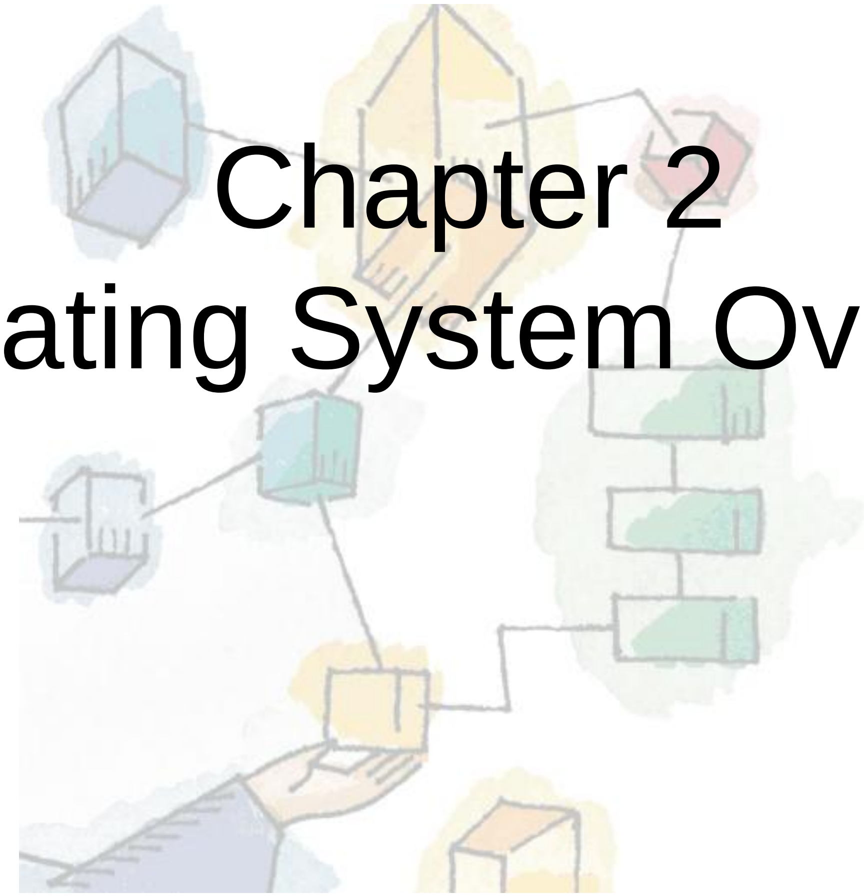
Chapter 2
Operating System Overview



Operating System

- A program that controls the execution of application programs
- An interface between applications and hardware
- Main objectives of an OS:
  - Convenience - Makes computer easy to use
  - Efficiency - Manages resources efficiently
  - Ability to evolve – Easy to adapt changes

Layers and Views

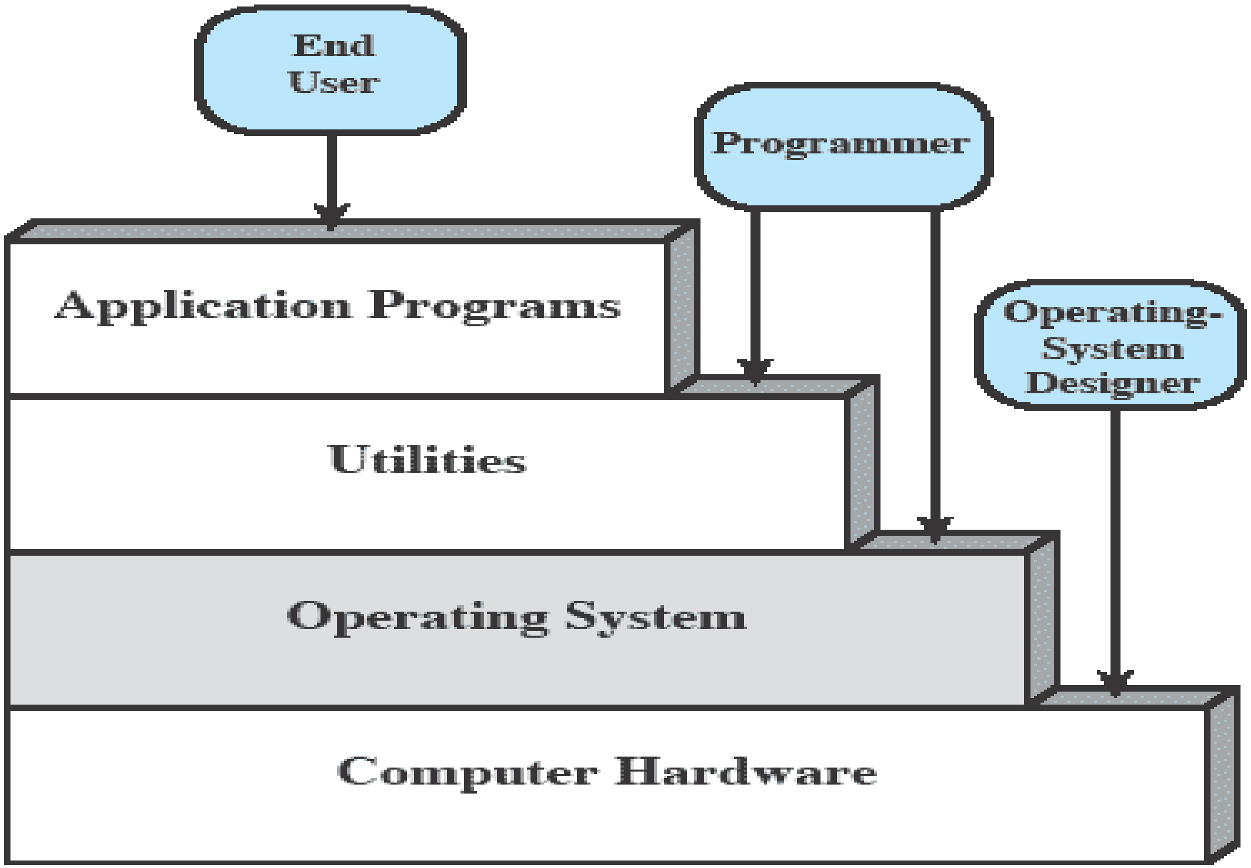


Figure 2.1 Layers and Views of a Computer System

Services Provided by the Operating System

- Program development
  - Editors and debuggers.
- Program execution
  - OS handles scheduling of numerous tasks required to execute a program
- Access I/O devices
  - Each device will have unique interface
  - OS presents standard interface to users

Services cont…

- Controlled access to files
  - Accessing different media but presenting a common interface to users
  - Provides protection in multi-access systems

- System access
  - Controls access to the system and its resources

Services cont…

- Error detection and response
  - Internal and external hardware errors (Ex: memory / Device failure)
  - Software errors (Ex: Divide by zero)

- Accounting
  - Collect usage statistics (Ex: Response Time)
  - Monitor performance

The Role of an OS

- A computer is a set of resources for the
  – movement,
  – storage and
  – processing of data

- The OS is responsible for managing these resources. – Hence OS is called resource manager

OS as Resource Manager

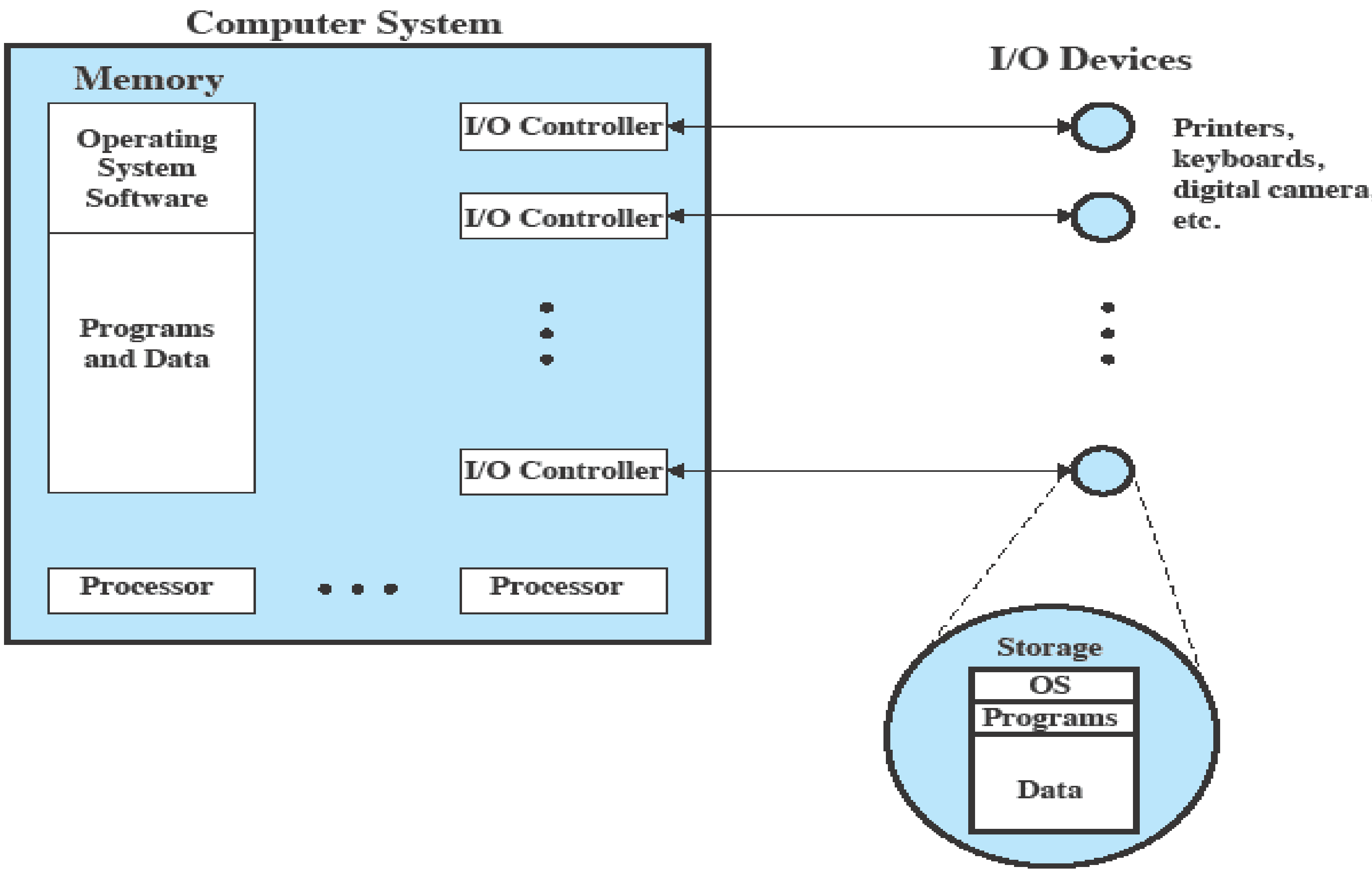


Figure 2.2 The Operating System as Resource Manager

Operating System as Software

- The OS functions in the same way as an *ordinary computer software*
  - It is a program that is executed by the CPU

- Operating system relinquishes control of the processor and regains when needed

Evolution of Operating Systems

- Operating systems have evolved over time
  - Hardware upgrades plus new types of hardware
  - New services
  - Fixes

Evolution of Operating Systems

- It may be easier to understand the key requirements of an OS by considering the evolution of Operating Systems

- **Evolution stages include**
  - Serial Processing
  - Simple Batch Systems
  - Multiprogrammed batch systems
  - Time Sharing Systems

Serial Processing

- *No operating system*
- Machines run from a console with
  - display lights,
  - toggle switches,
  - input device and
  - printer

Serial Processing Issues

- Scheduling
  - User assigned fixed amount of time
  - Problem when more or less time is used

- Setup time
  - Loading of compiler and program can be time consuming

Simple batch system

- Early computers were extremely expensive
  - Important to maximize processor utilization

- Hence Concept of Monitor was introduced
  - Software that controls the sequence of events
  - Batch jobs together
  - Program returns control to monitor when finished

Monitor’s perspective

- Monitor controls the sequence of events
- *Resident Monitor* is software always in memory
- Monitor reads in job and gives control
- Job returns control to monitor

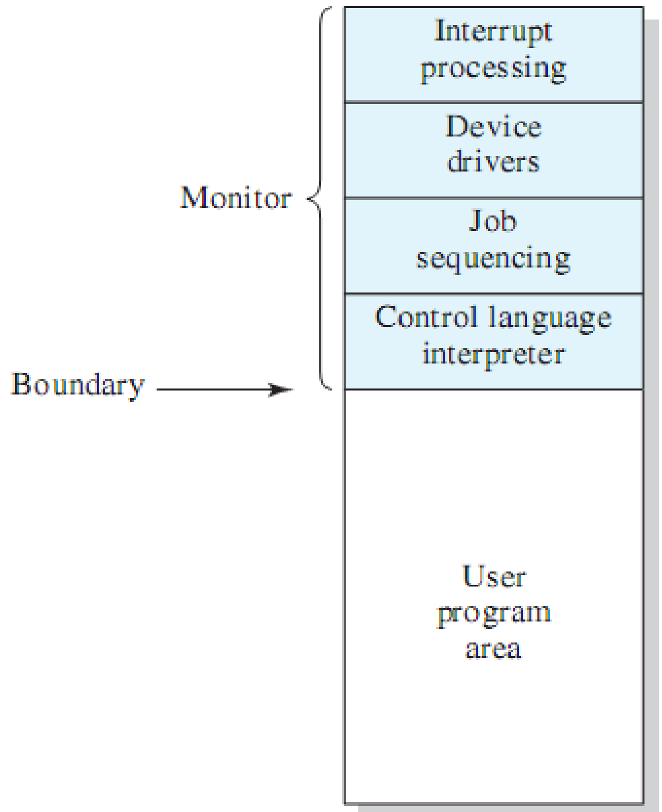


Figure 2.3 **Memory Layout for a Resident Monitor**

Modes of Operation

- User Mode
  - User program executes in user mode

- Kernel Mode
  - Monitor executes in kernel mode
  - Privileged instructions may be executed, all memory accessible.

Simple Batch System Issues

- CPU is often idle
  - Even with automatic job sequencing.
    - I/O devices are slow compared to processor

Read one record from file	15 μs
Execute 100 instructions	1 μs
Write one record to file	15 μs
TOTAL	31 μs

$$\text{Percent CPU Utilization} = \frac{1}{31} = 0.032 = 3.2\%$$

Figure 2.4 System Utilization Example

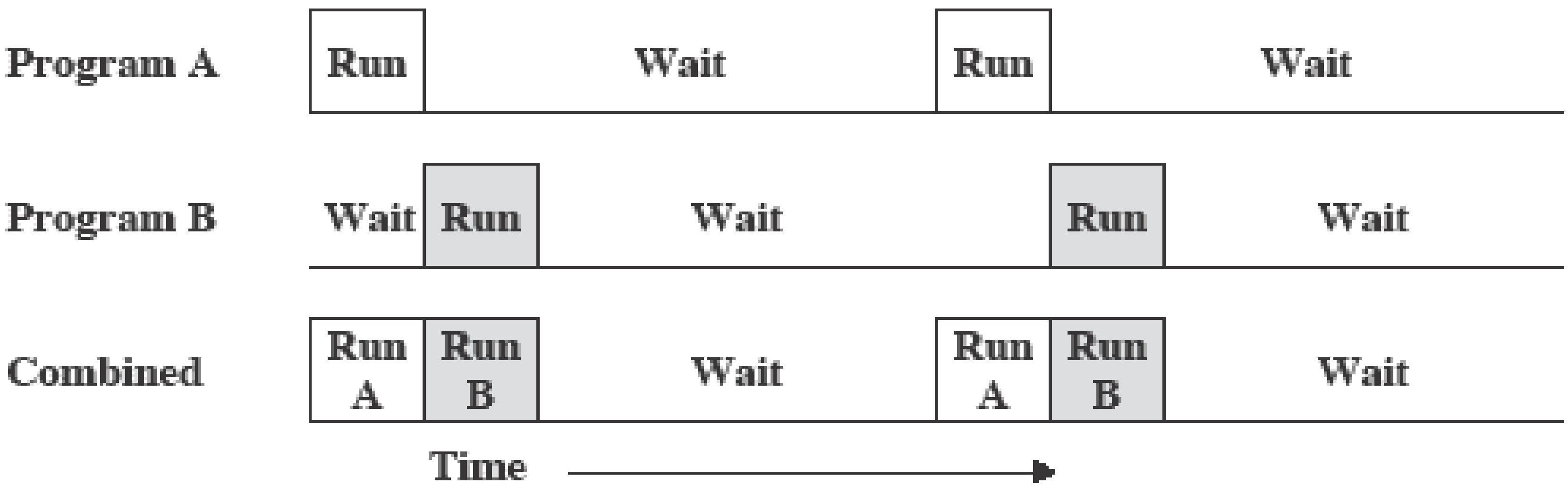
Uniprogramming

- Processor must wait for I/O instruction to complete before preceding

(a) Uniprogramming

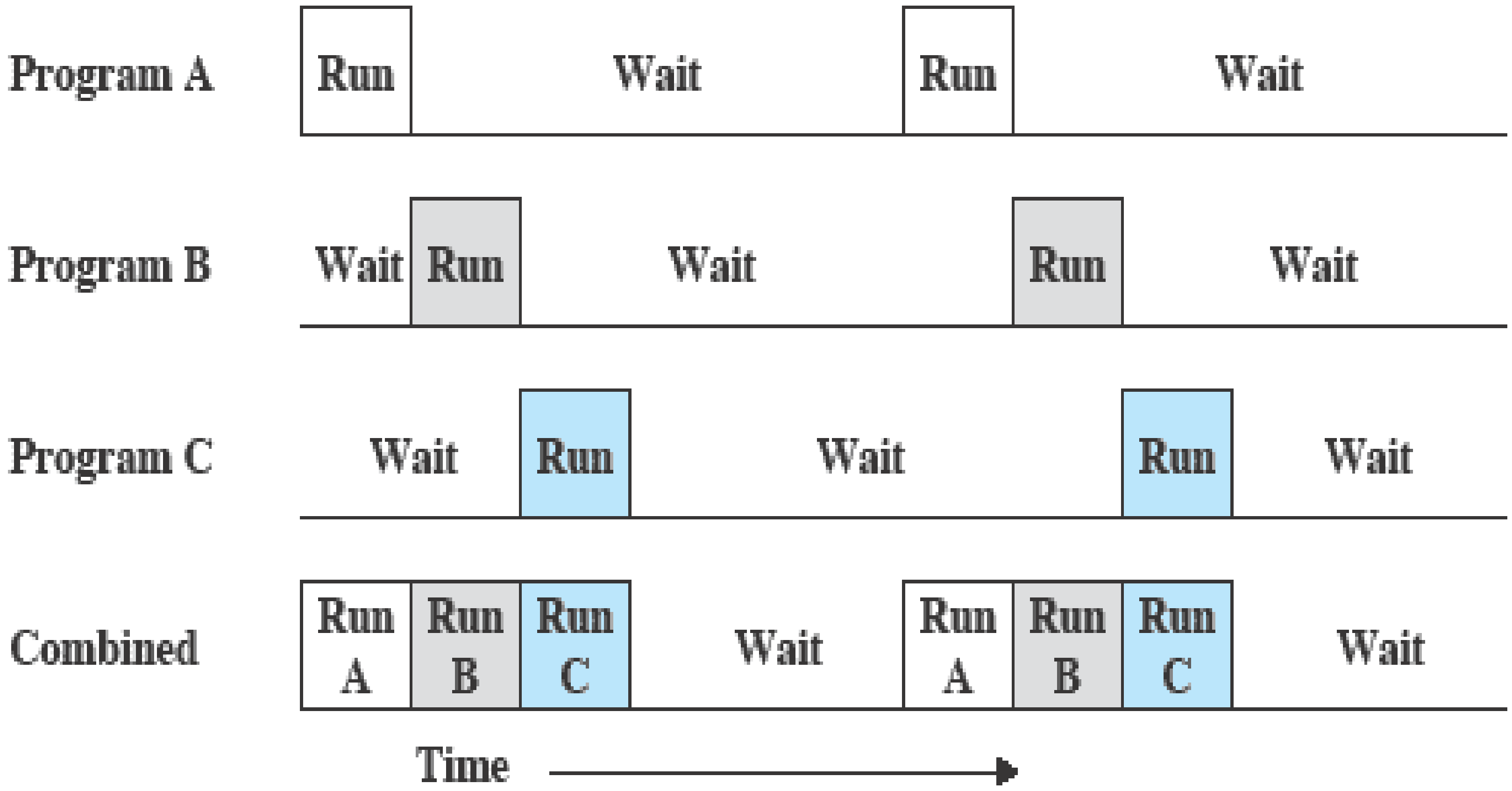
Multiprogramming

- When one job needs to wait for I/O, the processor can switch to the other job
- No user interaction



(b) Multiprogramming with two programs

Multi-programmed Batch Systems



(c) Multiprogramming with three programs

Time Sharing Systems

- Using multiprogramming to handle multiple interactive jobs
- Multiple users simultaneously access the system through terminals

Batch Multiprogramming vs. Time Sharing

Table 2.3 Batch Multiprogramming versus Time Sharing

	Batch Multiprogramming	Time Sharing
Principal objective	Maximize processor use	Minimize response time
Source of directives to operating system	Job control language commands provided with the job	Commands entered at the terminal

Issues

- *Multiple jobs* in memory must be protected from each other's data
- *File system* must be protected so that only authorised users can access
- *Contention* for resources must be handled
  - Printers, storage etc

Symmetric multiprocessing (SMP)

- An SMP system has
  - *multiple processors*
  - These processors share same main memory and I/O facilities
  - All processors can perform the same functions

- The OS of an SMP schedules processes or threads across all of the processors.

SMP Advantages

- **Performance**
  - Allowing parallel processing
- **Availability**
  - Failure of a single process does not halt the system
- **Incremental Growth**
  - Additional processors can be added.

Multiprogramming and Multiprocessing

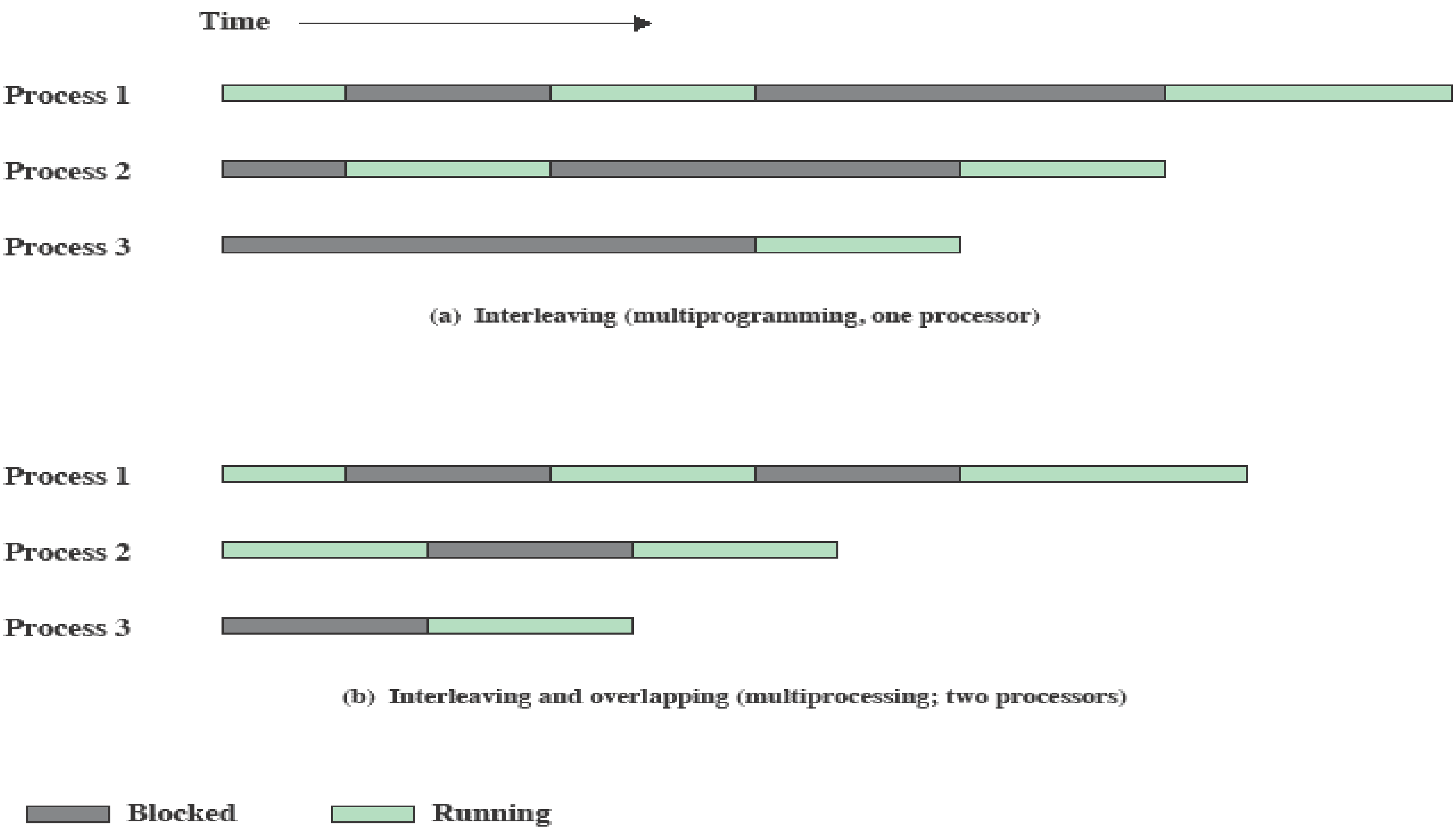


Figure 2.12 Multiprogramming and Multiprocessing

Distributed Operating Systems

- Provides the illusion of
  - a single main memory space and
  - single secondary memory space
- Early stage of development

Real Time Operating Systems

- A Real Time Operating System, commonly known as an **RTOS**,
  - Is a software component that *rapidly switches* between tasks,
  - Giving the *impression* that multiple programs are being executed at the same time on a single processing core.

Real Time Operating Systems

- In actual fact the processing core can only execute one program at any one time,
- And what the RTOS is actually doing is *rapidly switching* between individual programming threads (or Tasks)
- To give the impression that multiple programs are executing simultaneously.

System Calls

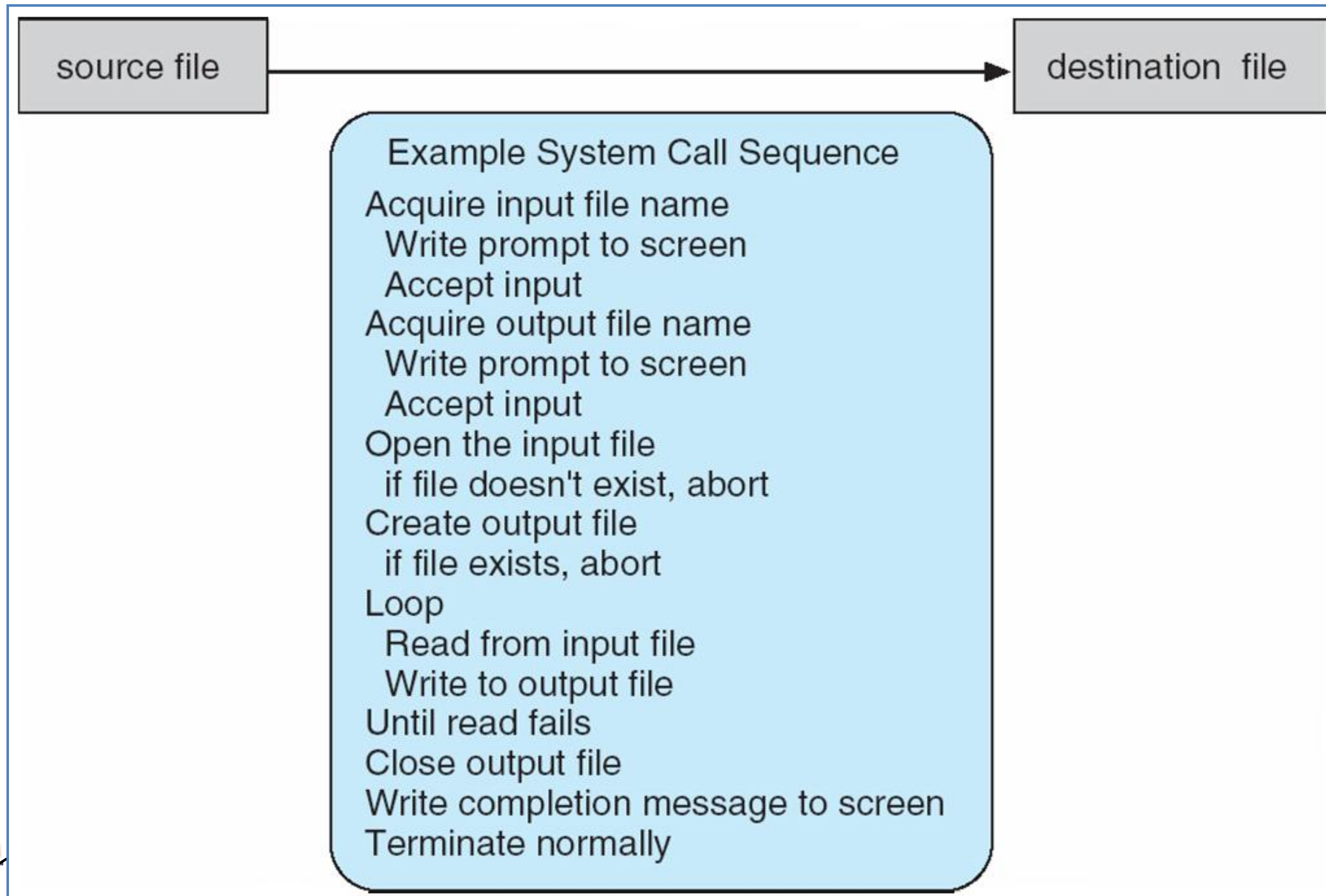
- **Definition:**
  - *Programming interface* to the services provided by the OS
- Typically written in a *high-level language* (C or C++)
- **Use:**
- Mostly accessed by programs via a high-level **Application Programming Interface** **(API)** rather than direct system call use

System Calls

- **Three 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),
  - 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



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)
```

$$\underbrace{\texttt{ssize\_t}}_{\text{return value}}\quad \underbrace{\texttt{read}}_{\text{function name}}\underbrace{\texttt{(int fd, void *buf, size\_t count)}}_{\text{parameters}}$$

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 where the data will be read into
- `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.

System Call Implementation

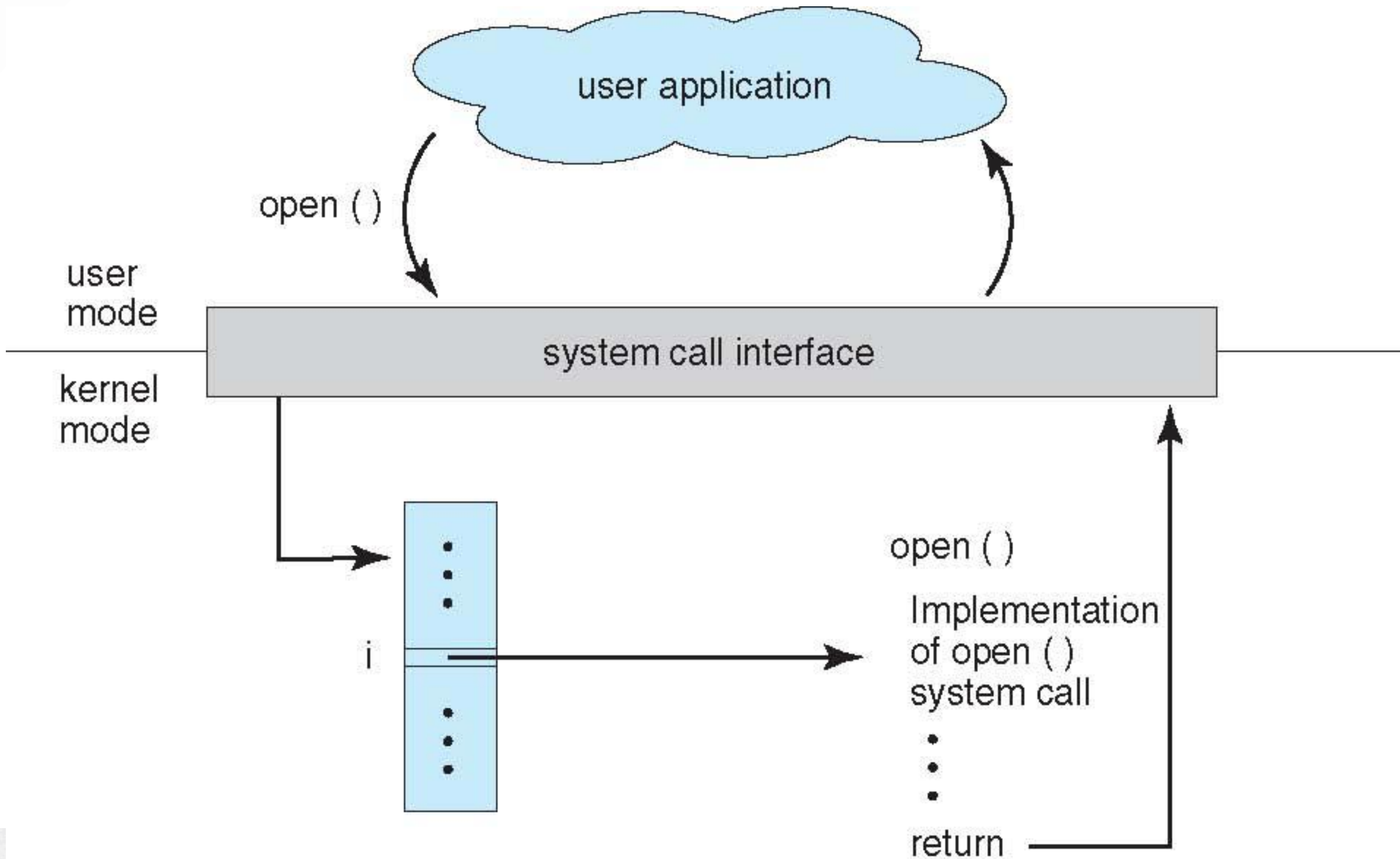
- Typically, a number associated with each system call
  - **System-call interface** maintains a table indexed according to these numbers

- The system call interface invokes the intended system call in OS kernel and returns status of the system call and any return values

System Call Implementation

- *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



Types of System Calls

- **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

- **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
  - transfer status information
  - attach and detach remote devices

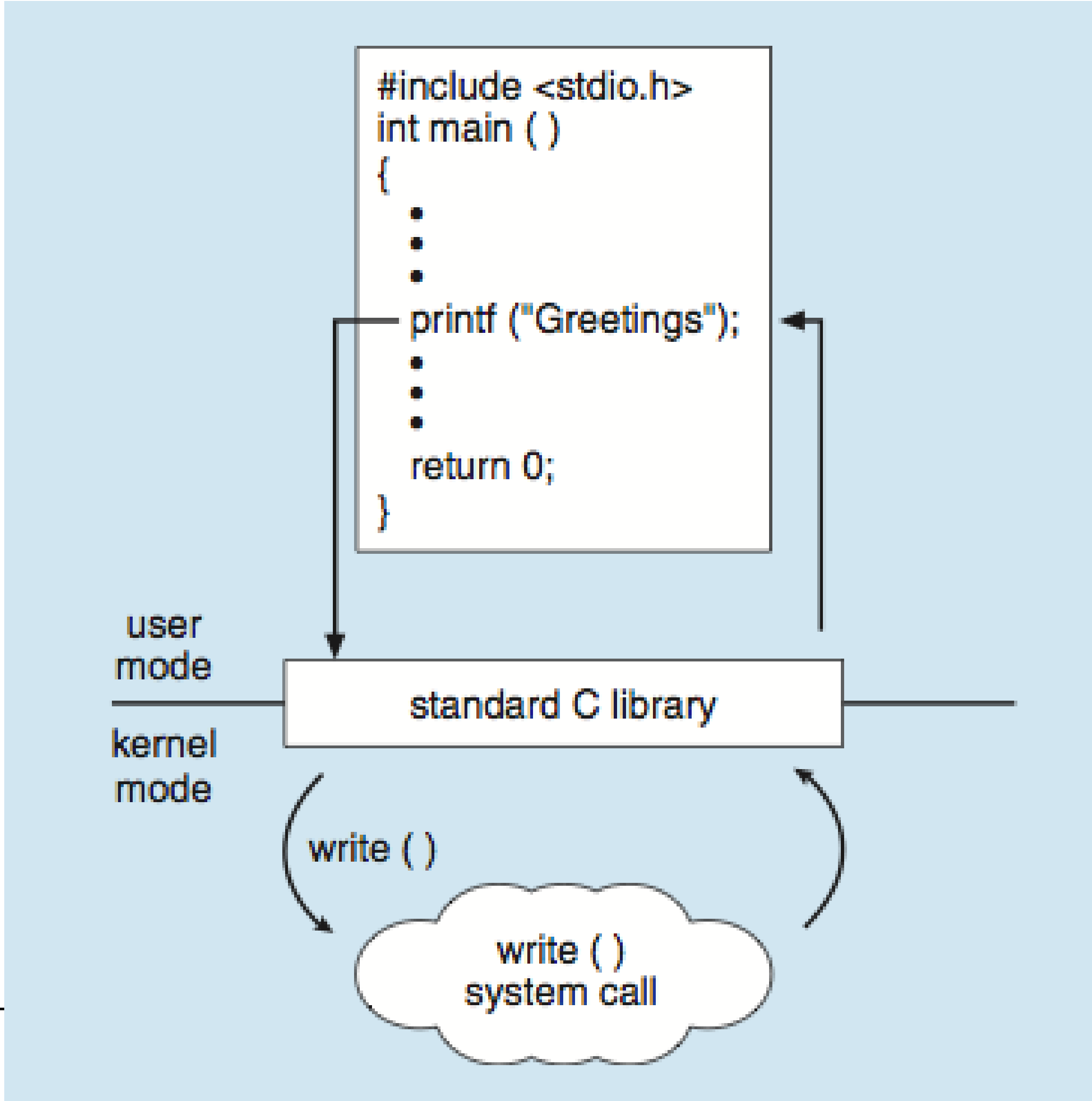
Types of System Calls

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

Examples of Windows and Unix System Calls

	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()

Standard C Library Example



References

- Operating Systems: Internals and Design Principles by William Stallings (6th Edition)

- Operating System Concepts by Silberschatz, Galvin and Gagne (9th Edition)