

MODULE 2: OPERATING SYSTEMS AND ALGORITHMS

PART I: OPERATING SYSTEMS

An Operating System is system software that controls the overall operation of a computer.

It acts as an interface (middleman) between:

- The user (or application programs)
- The computer hardware

Without an OS, using a computer would be extremely difficult – users would have to directly control hardware using machine-level instructions.

2. Main Functions / Roles of an Operating System

The text highlights three key everyday user-facing roles, but in reality, OS performs many more core functions:

- Provides means to store and retrieve files → File system management
- Provides interface for users to request program execution → User interface (GUI or command-line)
- Provides environment necessary to execute requested programs → Process & resource management

Core Functions of OS :

1. Process Management

- Handles creation, scheduling, execution, and termination of processes (running programs).
- Decides which process gets CPU time (scheduling).

2. Memory Management

- Allocates and de-allocates memory to programs.
- Manages virtual memory, paging, swapping.

3. File System Management

- Organizes, stores, retrieves, deletes files on storage devices.
- Handles directories, permissions, file naming.

4. Device Management (I/O Management)

- Controls hardware devices (keyboard, mouse, printer, disk, network).
- Uses device drivers for communication.

5. User Interface

- Provides Graphical User Interface (GUI) or Command Line Interface (CLI).
- Allows users to interact easily.

6. Security & Protection

- Protects data and resources from unauthorized access.
- User authentication, file permissions, virus protection support.

7. Other Functions

- Error detection & handling
- Job accounting (tracking resource usage)
- Networking support
- Coordination between hardware & software

3.1 THE HISTORY OF OPERATING SYSTEMS

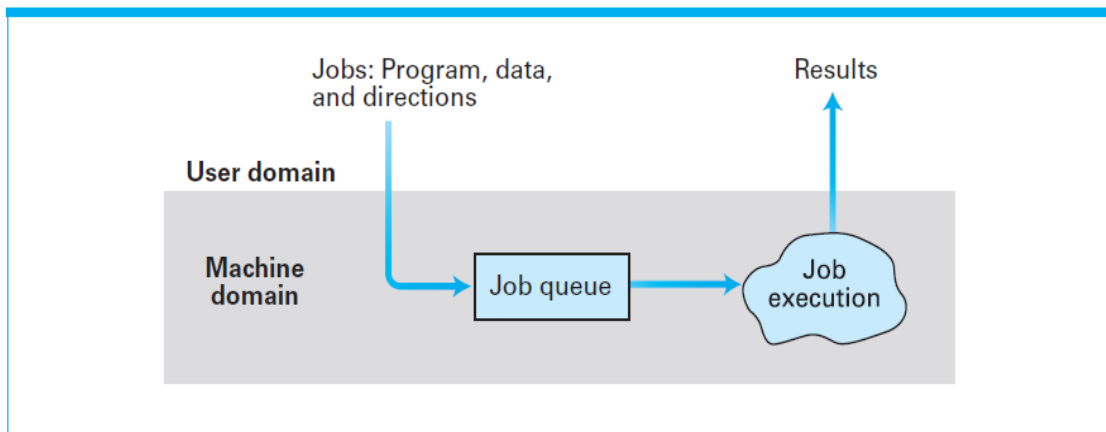
Early Computing (1940s-50s):

- Computers were room-sized machines requiring extensive manual preparation
- Each program execution called a **job** - isolated activity requiring setup and teardown
- Users reserved machine time via sign-up sheets; machine under complete user control during allocated time

Batch Processing:

- Computer operator hired to separate users from equipment
- Users submitted jobs with data and instructions to operator, returned later for results
- **Operating system** program read and executed jobs one at a time from mass storage
- Jobs waited in a **job queue** - first-in, first-out (FIFO) organization
- Some jobs could be bumped by higher-priority jobs
- Each job accompanied by instructions in **Job Control Language (JCL)**
- **Drawback:** No user interaction with job after submission

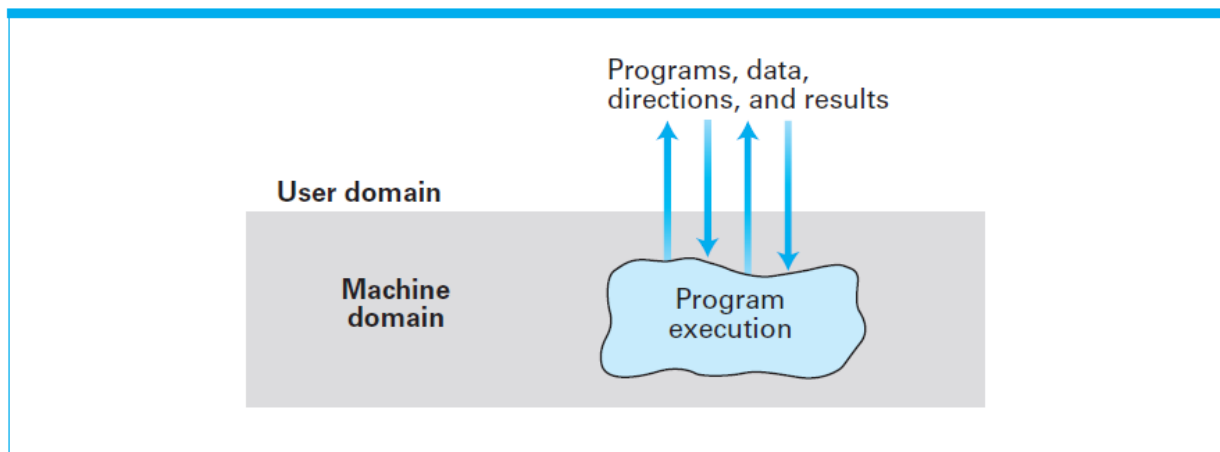
Figure 3.1 Batch processing



Interactive Processing:

- Developed to allow program-user dialogue through remote terminals
- Required **real-time processing** - computer actions fast enough to coordinate with user needs
- Terminals evolved from electronic typewriters to workstations and PCs

Figure 3.2 Interactive processing



NOTE: For successful interactive processing, the computer must respond fast enough to match the user's natural speed and expectations.

- The computer should adapt to the user's timetable
- NOT force the user to wait for the machine's slow schedule

Classic Comparison

Task	Can wait for computer?	Type of processing	Reason / Example
Payroll processing	Yes	Batch processing	Done overnight; no human is waiting; computer can take hours/days
Typing in a word processor	No	Interactive / Real-time	User types → expects immediate display of character; delay = frustration

→ If response is slow → user experience becomes frustrating and unusable.

2. What is Real-Time Processing?

Real-time processing means the computer performs tasks within strict deadlines imposed by the external real world (usually human users or physical processes).

- “Real-time” = actions happen in accordance with real-world timing requirements
- The computer is forced to meet deadlines → otherwise the system fails its purpose

Two Main Meanings of “Real Time” in Computing

Type	Deadline Strictness	Example Applications	Typical Response Time Expectation
Soft Real-Time	Important but missing occasionally is tolerable	Interactive systems, video streaming, online gaming	< 100–500 ms (human perception)
Hard Real-Time	Must never miss deadline (critical)	Aircraft flight control, airbag deployment, medical devices, industrial robots	Microseconds to milliseconds

(Note: The paragraph mainly discusses soft real-time in the context of interactive user systems.)

Why Was Real-Time Processing Easy in Single-User Systems?

4. The Big Challenge: Multi-User Interactive Systems (1960s–1970s)

Historical Context

- Computers in the 1960s and 1970s were extremely expensive
- Mainframes cost millions of dollars
- It was economically impossible to give each user their own computer

Solution adopted → Time-sharing / Multi-user systems

- Many users connected via remote terminals (teletype machines, CRT terminals)
- Several users worked simultaneously on the same central computer

- Goal: Give each user the illusion that they have the whole machine to themselves

Problem Created If the operating system runs only one job/program at a time (like in batch systems):

- Only one user gets good real-time response
- All other users wait → terrible interactive experience
- Defeats the purpose of interactive computing

5. Core Challenge Faced by Operating Systems

Main Obstacle How to provide real-time interactive response to multiple users simultaneously when:

- CPU time is shared
- Only one instruction can execute at a time
- Memory, disk, and I/O devices are also shared

Solution Developed → Time-sharing Operating Systems

Key techniques

Process scheduling (especially round-robin scheduling)

- Time slicing (quantum / time quantum of 10–100 ms)
- Preemptive multitasking
- Priority-based scheduling (interactive jobs get higher priority)
- Efficient I/O handling (interrupt-driven, not polling)

Goal of these techniques: → Give each user frequent small bursts of CPU time → Response time remains fast enough for human perception (< 0.1–0.5 seconds)

Quick Revision Table

Concept	Meaning / Requirement	Single-User System	Multi-User Time-Sharing System	Example
Interactive processing	Computer responds at human speed	Easy	Difficult (resource sharing)	Word processor, text editor
Real-time processing	Meets external (real-world) deadlines	Simple	Requires special OS design	Typing, command response
Batch processing	No human waiting; can run slowly	Common	Used for non-interactive jobs	Payroll, report generation

Concept	Meaning / Requirement	Single-User System	Multi-User Time-Sharing System	Example
Main challenge (1960s-70s)	Expensive hardware → must serve many users interactively at once	Not applicable	Core problem of early OS	Multiple terminals on one mainframe

7. Summary Points to Remember

- Interactive systems demand fast response → computer must follow user's timing, not vice versa
- This requirement is called real-time processing (soft real-time for humans)
- Single-user → easy to achieve real-time response
- Multi-user → very difficult → led to development of time-sharing operating systems
- Goal of time-sharing: Give every user the feeling of having a dedicated fast computer
- Without time-sharing and good scheduling → interactive computing would have remained impractical for most people

Mnemonic "Real-time = Real-world deadlines" "Interactive = User should never wait uncomfortably" "Time-sharing = Many users sharing one expensive computer fairly and quickly"

These concepts are the foundation of modern operating systems (Windows, Linux, macOS – all are time-sharing multi-tasking OS).

Time-Sharing and Multiprogramming:

- **Time-sharing:** Multiple users sharing computer simultaneously
- **Multiprogramming:** Time divided into intervals; each job executes one interval at a time
- Jobs rapidly shuffled to create illusion of simultaneous execution
- **Multitasking:** Single user executing numerous tasks simultaneously
- Early time-sharing systems supported up to 30 users with acceptable real-time processing

Evolution to Present:

- Computer operator role replaced by **system administrator**
- Administrator manages equipment, software installation, user accounts, storage limits
- Modern challenges: **load balancing** (efficient task allocation across processors), **scaling** (breaking tasks into appropriate subtasks)

- Networking extends operating system concepts to manage resources across multiple machines

Smartphone Operating Systems:

- Smartphones require full-fledged OS for resource management and application support
- Competitors: Apple iOS, BlackBerry OS, Windows Phone, Symbian OS, Android

Embedded systems are **computer systems** designed for **specific dedicated tasks** (not general-purpose computing like a laptop or desktop).

- They are **built into** (embedded within) larger devices or products
- Purpose: Control or monitor a specific function in that device
- Examples of devices with embedded systems:
 - Medical devices (pacemakers, infusion pumps)
 - Vehicle electronics (engine control unit/ECU, ABS, infotainment)
 - Home appliances (washing machines, microwaves, smart thermostats)
 - Cell phones & smartphones
 - Hand-held computers / PDAs (older)
 - IoT devices, industrial controllers, robots, etc.

Key Difference from General-Purpose Computers General-purpose (e.g., PC) → Runs many different programs, user can install new software freely Embedded → Fixed function, software is pre-installed and rarely changed by user

2. Special Requirements / Challenges of Embedded Operating Systems

Embedded OS must satisfy strict constraints (unlike desktop OS like Windows/Linux):

Requirement	Why Important?	Typical Constraint / Design Focus
Conserve battery power	Many devices are battery-powered (phones, wearables, remote sensors)	Low power modes, efficient scheduling, sleep states
Meet demanding real-time deadlines	Life-critical or safety-critical (e.g., airbag deployment, medical monitor)	Hard/soft real-time guarantees, deterministic response
Operate continuously with little/no human oversight	Devices run 24/7 (e.g., car ECU, factory machine)	High reliability, fault tolerance, no frequent reboots

Requirement	Why Important?	Typical Constraint / Design Focus
Limited resources	Small memory (KB–few MB), low CPU power, no hard disk	Tiny footprint, minimal overhead
Custom hardware	Specific microcontroller/processor, no standard drivers	Highly tailored drivers & BSP (Board Support Package)

→ Embedded OS are often **Real-Time Operating Systems (RTOS)** or lightweight variants.

NOTE:

- **Embedded systems** = computers inside other devices for **specific tasks**
- Embedded OS must handle: **low power, real-time deadlines, continuous reliable operation**
- Classic successes: **VxWorks** (Mars missions – still relevant), **Windows CE** (legacy handhelds), **Palm OS** (PDA era → evolved to webOS for TVs)
- Modern embedded world: Mix of commercial RTOS (critical apps) + open-source (IoT, low-cost)
- Research direction: Making OS efficient for battery-powered, autonomous, real-time devices everywhere (phones → cars → space)

Mnemonic “Embedded = Specific task + Tiny + Real-time + Reliable + Low power” “VxWorks went to Mars – proof of reliability in extremes!”

This topic connects OS concepts (real-time, scheduling) to real-world devices – very important for future embedded/IoT courses.

3.2 OPERATING SYSTEM ARCHITECTURE

Software Classification

Application Software:

- Programs for tasks specific to machine's utilization
- Examples: spreadsheets, database systems, desktop publishing, accounting systems, games

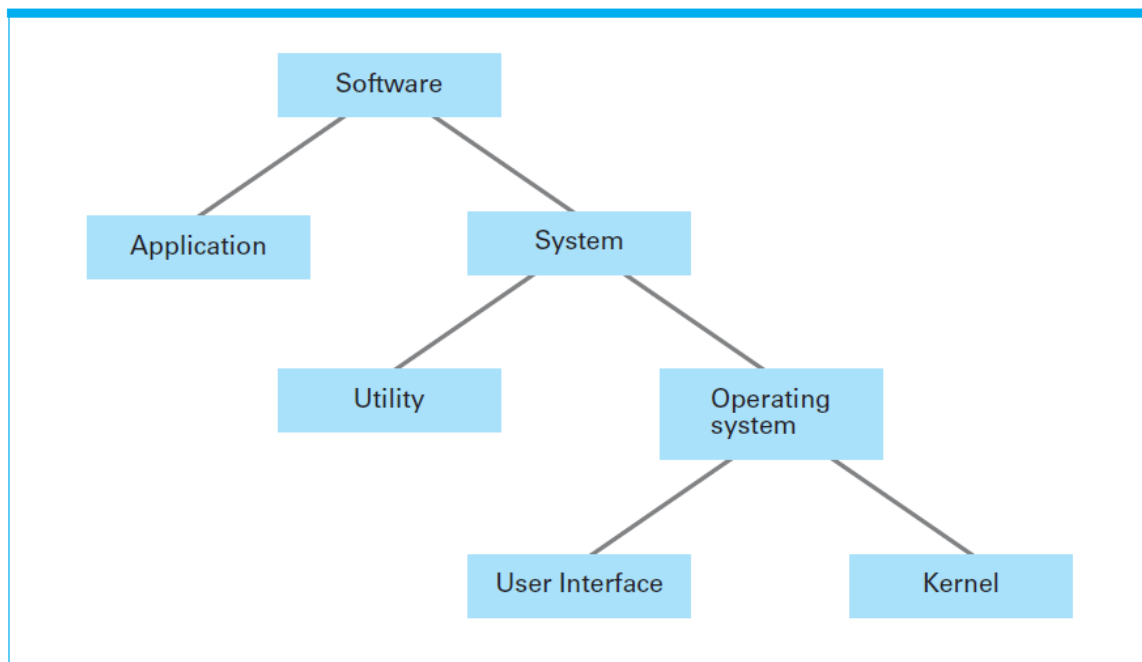
System Software:

- Performs tasks common to all computer systems
- Provides infrastructure for application software

Two Categories of System Software:

1. **Operating System** - Core system software
2. **Utility Software** - Extends/customizes OS capabilities
 - Examples: disk formatting, file copying to CD, data compression, multimedia playback, network communication
 - Distinction between utility and application software is vague; utilities become fundamental tools over time

Figure 3.3 Software classification

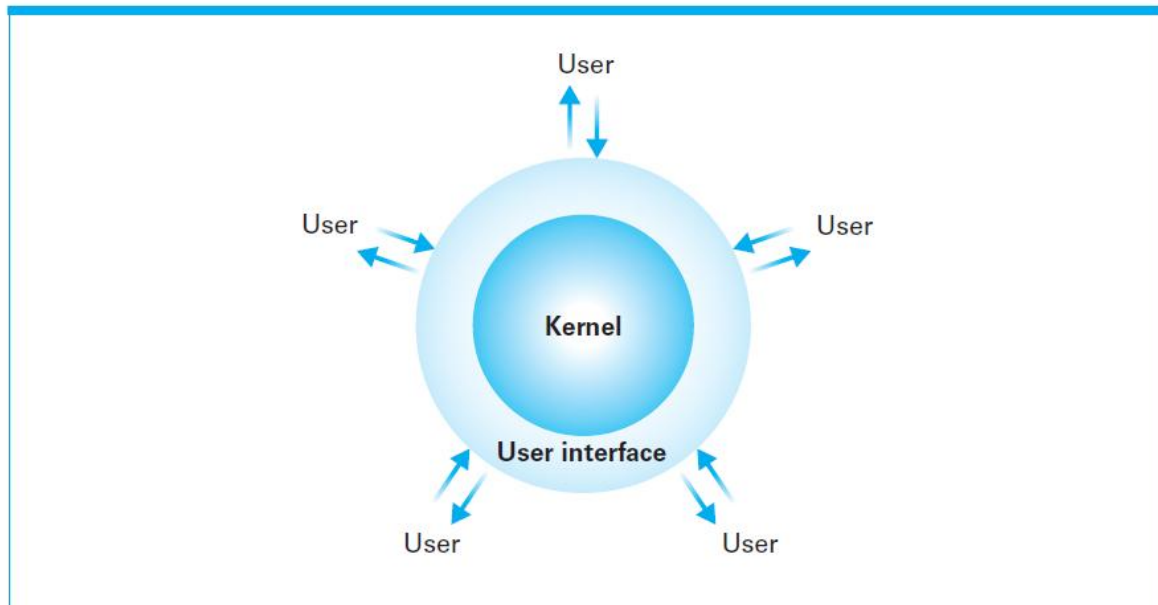


Components of an Operating System

User Interface:

- **Shells:** Older text-based interfaces using keyboard and monitor
- **GUI (Graphical User Interface):** Icons represent files and programs; mouse, stylus, or touch input
- **Window Manager:** Allocates screen space (windows), tracks application associations, processes mouse clicks
- UNIX offers multiple shell options (Bourne, C, Korn) and GUI (X11)
- Windows retains DOS cmd.exe shell; Apple OS X has Terminal (UNIX ancestry)

Figure 3.4 The user interface acts as an intermediary between users and the operating system's kernel



Kernel (Internal Core):

1. File Manager:

- Coordinates mass storage usage
- Maintains records of file locations, access permissions, available storage space
- Allows files to be grouped into **directories/folders**
- Directories can contain **subdirectories** forming hierarchical organization
- **Directory path**: Chain of directories (e.g., animals/prehistoric/dinosaurs)
- Files accessed through **opening** procedure with approval from file manager

2. Device Drivers:

- Communicate with controllers and peripheral devices
- Translate generic requests into device-specific technical steps
- Each driver designed for particular device type (printer, disk drive, monitor)
- Enable generic OS customization through appropriate driver installation

3. Memory Manager:

- Coordinates main memory usage
- In multitasking environments: finds/assigns memory space for multiple programs, ensures programs stay within allotted space
- Tracks and reclaims memory areas no longer occupied

- **Paging:** Creates illusion of larger memory by swapping pages between main memory and mass storage
 - **Virtual memory:** Fictional large memory space created through paging
4. **Scheduler:** Determines which activities are considered for execution
 5. **Dispatcher:** Controls allocation of time to activities

Getting It Started - Booting

The Bootstrap Problem:

- CPU's program counter starts at predetermined address when powered on
- Main memory is **volatile** - loses contents when power is off
- Need program in memory at startup, but memory is empty when machine is off

Solution:

- Small portion of memory constructed from **nonvolatile ROM (Read-Only Memory)**
- **Boot loader** program permanently stored in ROM
- When machine turns on, CPU executes boot loader from ROM
- Boot loader transfers OS from mass storage into volatile main memory
- Boot loader then jumps to that memory area; OS takes control
- This process is called **booting** the computer

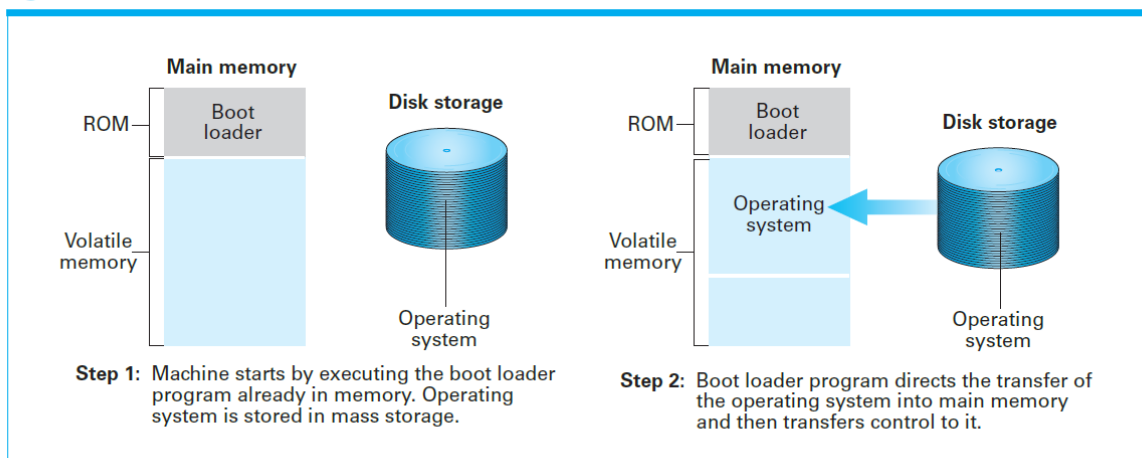
Firmware:

- ROM contains software routines for fundamental I/O activities
- Not hardware (immutable) nor software (readily changeable) - **firmware** is the middle ground
- Examples: BIOS, EFI, Open Firmware, CFE
- Used by boot loader to perform I/O before OS becomes functional

Why Not Store Entire OS in ROM?

- Embedded systems (smartphones) can do this due to small OS size
- General-purpose computers: inefficient to devote large memory blocks to nonvolatile storage
- OS undergoes frequent updates for security and new device drivers
- **Firmware updates** possible but technologically limited

Figure 3.5 The booting process



3.3 COORDINATING THE MACHINE'S ACTIVITIES

The Concept of a Process

Program vs. Process:

- **Program:** Static set of directions (like sheet music on a shelf)
- **Process:** Dynamic activity of executing a program (like musician performing the piece)
- **Process state:** Current status including program counter value, CPU registers, associated memory cells

Task of Operating System:

- Manage multiple processes competing for resources
- Ensure each process has needed resources
- Prevent interference between independent processes
- Enable information exchange between processes that need to communicate

Process Administration

To keep track of all the processes, the scheduler maintains a block of information in main memory called the **process table**. Each time the execution of a program is requested, the scheduler creates a new entry for that process in the process table. This entry contains such information as the memory area assigned to the process (obtained from the memory manager), the priority of the process, and whether the process is ready or waiting. A process is **ready** if it is in a state in which its progress can continue; it is **waiting** if its progress is currently delayed until some external event occurs, such as the completion of a mass storage operation, the pressing of a key at the keyboard, or the arrival of a message from another process.

Scheduler:

- Maintains record of all processes in the system
- Introduces new processes to the pool
- Removes completed processes
- Maintains **process table** in main memory
- Process table entry contains: memory area assigned, priority, ready/waiting status

Process States:

- **Ready:** Process can continue its progress
- **Waiting:** Process delayed until external event occurs (I/O completion, keyboard press, message arrival)

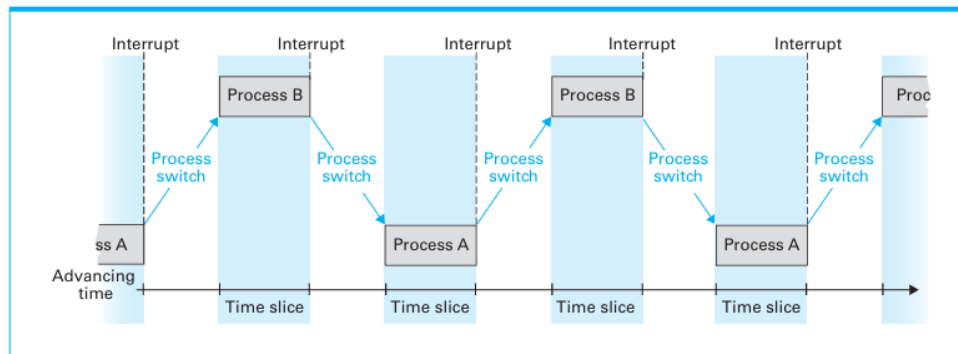
Dispatcher:

The dispatcher is the component of the kernel that oversees the execution of the scheduled processes. In a time-sharing/multitasking system this task is accomplished by **multiprogramming; that is, dividing time into short segments, each called a time slice (typically measured in milliseconds or microseconds)**, and then switching the CPU's attention among the processes as each is allowed to execute for one time slice. **The procedure of changing from one process to another is called a process switch (or a context switch).** Each time the dispatcher awards a time slice to a process, it initiates a timer circuit that will indicate the end of the slice by generating a signal called an interrupt. The CPU reacts to this interrupt signal in much the same way that you react when interrupted from a task. You stop what you are doing, record where you are in the task (so that you will be able to return at a later time), and take care of the interrupting entity. When the CPU receives an interrupt signal, it completes its current machine cycle, saves its position in the current process, and begins executing a program, called an interrupt handler, which is stored at a predetermined location in main memory. **This interrupt handler is a part of the dispatcher, and it describes how the dispatcher should respond to the interrupt signal.** Thus, the effect of the interrupt signal is to preempt the current process and transfer control back to the dispatcher. At this point, the dispatcher selects the process from the process table that has the highest priority among the ready processes (as determined by the scheduler), restarts the timer circuit, and allows the selected process to begin its time slice.

- Oversees execution of scheduled processes
- Implements **multiprogramming** through **time slices** (milliseconds/microseconds)
- **Process switch/context switch:** Changing from one process to another

Below **Figure3.6** shows the Multiprogramming mechanism between process A and B

Figure 3.6 Multiprogramming between process A and process B



Interrupt Mechanism:

- Dispatcher initiates timer circuit at start of time slice
- Timer generates **interrupt** signal at end of slice
- CPU completes current machine cycle, saves process position
- CPU executes **interrupt handler** (part of dispatcher) at predetermined memory location
- Dispatcher selects highest priority ready process, restarts timer, allocates time slice

Importance of Process State Saving:

- Must re-create environment present before interruption
- Requires saving program counter, registers, pertinent memory cells
- Modern CPUs incorporate automatic state saving and reloading instructions
- CPU design influenced by operating system needs

Efficiency of Multiprogramming:

- Without multiprogramming: time spent waiting for I/O is wasted
- With multiprogramming: I/O-bound process marked as waiting, CPU given to other processes
- When I/O completes, process marked ready and competes for time slices
- Overall collection of tasks completes faster despite context switch overhead

3.4 HANDLING COMPETITION AMONG PROCESSES

Semaphores

Problem: Shared Resource Access

- Example: Single printer shared by multiple processes
- Need to prevent simultaneous access (results would be worthless)

Flag Approach and Its Flaw:

- **Flag:** Bit in memory; clear (0) = resource available, set (1) = resource allocated
- Process: Check flag → if clear, grant request and set flag
- **Problem:** Flag testing and setting requires multiple machine instructions
- Process can be interrupted after detecting clear flag but before setting it
- Interrupting process also finds flag clear, also granted access
- **Result:** Two processes granted simultaneous access

Solutions:

1. Disable/Enable Interrupts:

- Start flag-testing routine with **interrupt disable** instruction
- End with **interrupt enable** instruction
- No other activity can interrupt the routine once started

2. Test-and-Set Instruction:

- Single machine instruction: retrieves flag value, notes it, sets flag
- CPU completes instruction before recognizing interrupts
- Task cannot be split

Semaphore:

- Properly implemented flag for resource allocation is called a Semaphore.
- Named after railroad signals controlling track access
- Used to guard **critical regions** - sequences of instructions that should execute by only one process at a time
- **Mutual exclusion:** Requirement that only one process at a time execute a critical region

Semaphore Protocol:

- To enter critical region: find semaphore clear → set semaphore → enter
- To exit critical region: clear semaphore
- If semaphore set: process must wait until cleared

The requirement that only one process at a time be allowed to execute a critical region is known as **mutual exclusion**. In summary, a common way of obtaining mutual exclusion to a critical region is to guard the critical region with a semaphore. To enter the critical region, a process must find the semaphore clear and then set the semaphore before entering the critical region; then upon exiting the critical region, the process must clear the semaphore. If the semaphore is found in its set state, the process trying to enter the critical region must wait until the semaphore has been cleared.

Deadlock

Definition:

- Condition where two or more processes are blocked from progressing
- Each process waiting for a resource allocated to another

Examples:

1. Process A has printer, waiting for CD player; Process B has CD player, waiting for printer
2. **Forking**: Process creates new process (subtask); scheduler has no space left in process table; each process must create another before completing; no process can continue

Three Conditions Required for Deadlock:

1. Competition for **nonsharable resources**
2. Resources requested on **partial basis** (process receives some resources, later requests more)
3. **No forcible retrieval** once resource allocated

Deadlock Detection and Correction:

- Attacks condition #3
- Deadlock considered rare; detect when it occurs, then correct
- Example: **Kill** some processes to release space in process table
- Breaks deadlock, remaining processes can continue

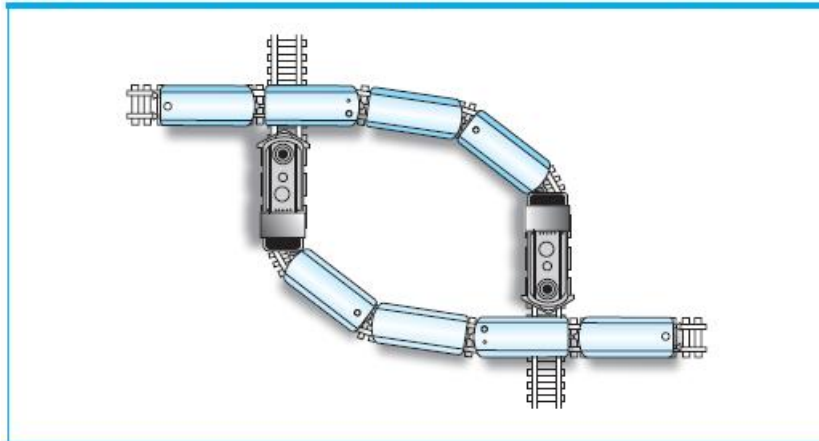
Deadlock Avoidance Schemes:

1. **Attack condition #2**:
 - Require each process to request **all resources at once**
2. **Attack condition #1**:
 - Convert nonsharable resources into sharable ones
 - **Spooling**: Create illusion of multiple resources
 - Example: Grant printer requests but store output in mass storage
 - Process thinks it has printer access
 - Later, OS transfers data from mass storage to actual printer
 - File manager: multiple processes can read same file; only one can write
 - File can be divided so different processes alter different parts concurrently

Python and Operating Systems:

- When Python script executes, OS launches new process
- Python scripts can be applications or utility software
- Python's "**os**" **module** provides system-agnostic functions for accessing OS features
- Functions include forking new processes, executing other utility/application programs

Figure 3.7 A deadlock resulting from competition for nonsharable railroad intersections



3.5 SECURITY

Security Responsibilities of Operating System:

1. Reliability:

- Flaw in file manager → loss of file data
- Defect in dispatcher → system crash, loss of work
- Requires well-designed, dependable operating system

Attacks from the Outside

User Accounts and Authentication:

- Operating system establishes "accounts" for authorized users
- Account contains: user name, password, privileges
- **Login procedure:** User establishes initial contact with OS
- **Super user/Administrator:** Highly privileged access
 - Alters OS settings, modifies critical software, adjusts user privileges
 - Performs maintenance activities denied to normal users

Auditing Software:

- Records and analyzes activities within computer system
- Detects: flood of incorrect password attempts (unauthorized access attempts)

- Identifies activities inconsistent with user's past behavior
- Detects **sniffing software** - records activities, reports to intruder
- Example: Program simulating OS login procedure to capture passwords

User Carelessness:

- Major obstacle to computer security
- Easy-to-guess passwords (names, dates)
- Sharing passwords with friends
- Failure to change passwords regularly
- Transferring offline mass storage between machines
- Importing unapproved software that may subvert security

Attacks from Within

Unauthorized Access Attempts:

- Intruder with general user account tries to exceed privileges
- May trick memory manager into accessing memory outside allotted area
- May trick file manager into retrieving unauthorized files

Hardware Security Features:

1. Memory Limit Registers:

- Special-purpose registers store upper and lower limits of process's memory area
- CPU compares each memory reference to these registers during process execution
- Reference outside limits → CPU transfers control to OS via interrupt

2. Privilege Levels:

- CPUs designed to operate in two modes:
 - **Privileged mode:** CPU can execute all machine language instructions
 - **Nonprivileged mode:** Limited instruction set
- **Privileged instructions:** Available only in privileged mode
 - Examples: changing memory limit registers, changing current privilege mode
- Attempt to execute privileged instruction in nonprivileged mode → interrupt
- Interrupt converts CPU to privileged mode, transfers control to OS

Boot Process:

- CPU starts in privileged mode when first turned on
- OS executes all instructions at startup
- When OS allows process to start time slice, switches CPU to nonprivileged mode via "change privilege mode" instruction
- OS notified if process attempts privileged instruction

Consequences of Security Flaws:

- Process altering timer → extend time slice, dominate machine
- Process accessing peripheral devices directly → read files without file manager supervision
- Process accessing memory outside allotted area → read/alter data used by other processes

PART II: ALGORITHMS

5.1 THE CONCEPT OF AN ALGORITHM

Informal Definition:

- A set of steps that defines how a task is performed
- Examples: cooking recipes, directions, machine cycle, shelling peas

Formal Definition of an Algorithm

An algorithm is an **ordered set** of **unambiguous, executable steps** that defines a **terminating process**.

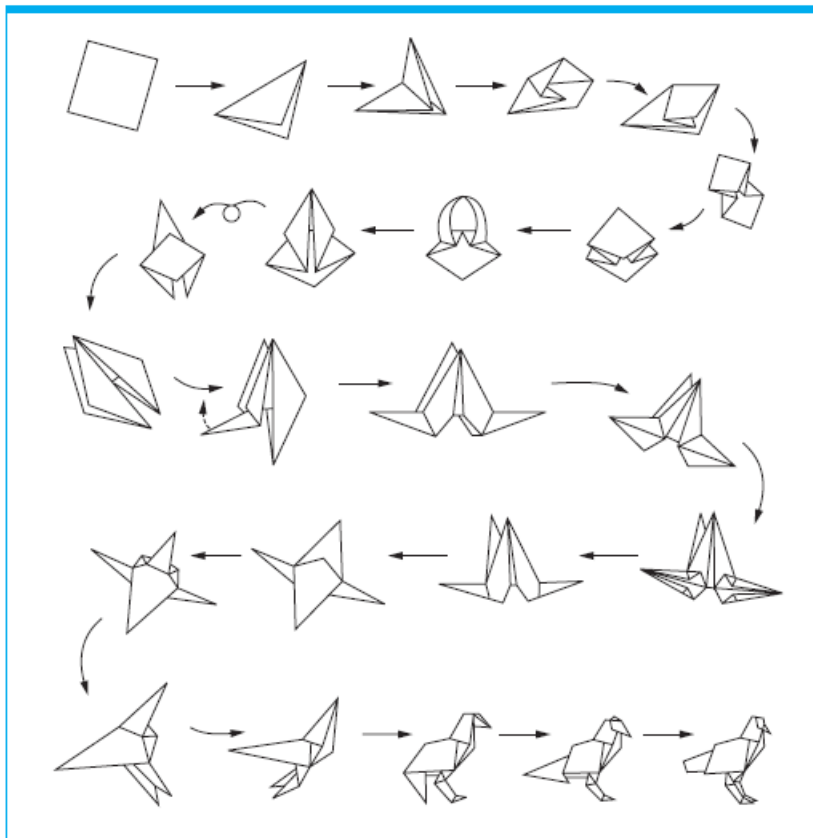
Key Requirements:

1. **Ordered:** Steps must have well-established execution structure
 - Does NOT require sequential execution
 - **Parallel algorithms:** Multiple sequences for different processors
 - Circuits: each gate performs a step, ordered by cause and effect
2. **Executable (Effective):** Steps must be doable
 - Example of non-executable: "Make a list of all positive integers" (infinitely many)
3. **Unambiguous:** Information in process state must uniquely and completely determine actions required by each step
 - Execution does NOT require creative skills
 - Requires only ability to follow directions
 - **Nondeterministic algorithms** (important research topic) do NOT conform to this

4. **Terminating:** Execution must lead to an end

- Origin: Theoretical computer science distinguishes problems with algorithmic solutions vs. those beyond algorithmic capabilities
- Note: Some meaningful applications use nonterminating processes (patient monitoring, aircraft altitude control)
- Applied settings often use "algorithm" loosely for nonterminating processes

Figure 5.2 Folding a bird from a square piece of paper



The Abstract Nature of Algorithms

Algorithm vs. Representation:

- Algorithm is abstract/conceptual
- Representation is physical (like story vs. book)
- Same algorithm can be represented in many ways

- Example: Celsius to Fahrenheit conversion as algebraic formula, English instruction, or electronic circuit

Level of Detail Problem:

- Meteorologists understand "Convert Celsius to Fahrenheit"
- Layperson needs more detailed description
- Problem is representation, not the underlying algorithm
- **Primitives** eliminate ambiguity problems in representation

Programs vs. Algorithms vs. Processes:

- **Program:** Formal representation of algorithm designed for computer application
- **Process:** Activity of executing a program (algorithm)
- **Programs, algorithms, processes** are distinct yet related entities

5.2 ALGORITHM REPRESENTATION

Primitives

Definition:

- Well-defined set of building blocks for constructing algorithm representations
- Assigning precise definitions removes ambiguity
- Establishes uniform level of detail
- **Programming language:** Collection of primitives + rules for combining them

Syntax vs. Semantics:

- **Syntax:** Primitive's symbolic representation
- **Semantics:** Meaning of the primitive

Levels of Primitives:

- Machine language instructions = low-level primitives
- Higher-level primitives = abstract tools constructed from lower-level primitives

The below shown Figure 5.3 presents some of the primitives used in origami.

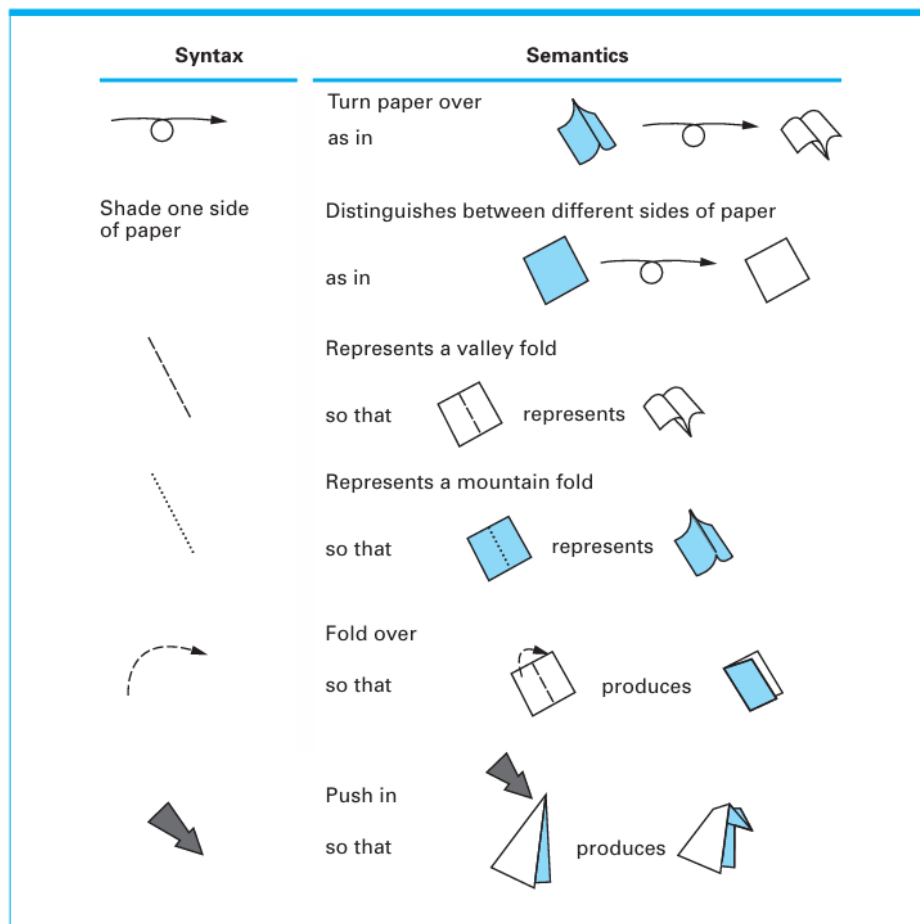


Figure 5.3: Origami Primitives

Pseudocode

Definition:

- Notational system for informally expressing ideas during algorithm development
- Loosened rules of formal programming language
- Must have consistent, concise notation for recurring semantic structures

Common Semantic Structures in Pseudocode:

1. **Assignment Statement** (saving computed value):

text

name = expression

Example: RemainingFunds = CheckingBalance + SavingsBalance

2. **Selection (if-else):**

Text

if (condition):

activity

else:

activity

Example is as follows:

```
if (year is leap year):  
    daily total = total / 366  
else:  
    daily total = total / 365
```

- Colon after condition and else
- Indented activity blocks
- Single alternative: if (condition): activity

We also adopt the shorter syntax

if (condition): Activity

for those cases not involving an else activity. Using this notation, the statement
Should it be the case that sales have decreased, lower the price by 5%.

will be reduced to

Example:

```
if (sales have decreased):  
    lower the price by 5%
```

3. Iteration (while loop):

Another common semantic structure is the repeated execution of a state ment or sequence of statements as long as some condition remains true. Informal examples include As long as there are tickets to sell, continue selling tickets. and While there are tickets to sell, keep selling tickets. For such cases, we adopt the uniform pattern

Text

while (condition):

activity

- Check condition; if true, perform activity and repeat
- If false, move to next instruction after while structure

while (condition)

Activity

Example:

```
while (tickets remain to be sold):  
    sell a ticket
```

Importance of Indentation:

- Essential in Python and Python-derived pseudocode
- Shows nesting structure
- Determines which statements belong to which control structure

In many programming languages, indentation often enhances the readability of a program. In Python, and thus also in our Python-derived pseudocode, indentation is essential to the notation. For example, in the statement

```
if (not raining):  
    if (temperature == hot):  
        go swimming  
    else:  
        play golf  
else:  
    watch television
```

indentation tells us that the question of whether temperature equals hot will not even be asked unless it is not raining. Note the use of double equal signs, “==” to differentiate between assignment (=) and comparison (==). The question about the temperature is nested inside the if-statement, and is, in effect, the activity to be performed if the outer if-statement condition holds true. Similarly, indentation tells us that else: play golf belongs to the inner if-statement, rather than to the outer if-statement. Thus we will adopt the use of indentation in our pseudocode.

Functions in Pseudocode:

text

def name():

statements

- **Parameters:** Generic names listed in parentheses

text

def Sort(List):

statements using List

- Call function: Sort(the organization's membership list)

Below is the function Greetings in pseudocode

```
def Greetings():
    Count = 3
    while (Count > 0):
        print('Hello')
        Count = Count - 1
```

5.3 ALGORITHM DISCOVERY

The Art of Problem Solving

Key Insight:

- Algorithm discovery requires finding method to solve problem
- Cannot be reduced to an algorithm itself (proven impossible - see Chapter 12)
- Problem solving remains an artistic skill, not a precise science

Polya's Four Phases (1945):

Phase	General Problem Solving	Program Development Context
1	Understand the problem	Understand the problem
2	Devise a plan	Get idea of algorithmic function
3	Carry out the plan	Formulate algorithm and program
4	Evaluate solution	Evaluate program for accuracy and reuse

Important Observations:

- Phases are NOT sequential steps to follow
- Successful problem solvers may start Phase 2 before Phase 1 is complete
- Failed strategies lead to deeper understanding → better strategies

- **Incubation period:** Period between conscious work and sudden inspiration
- Subconscious mind appears to continue working on problem

Getting a Foot in the Door

Common Thread: All problem-solving techniques share this concept

General Approaches:

1. **Work the problem backward:**
 - Start with desired output, back up to given input
 - Example: Unfolding completed origami bird to see construction
2. **Look for related/easier/solved problems:**
 - Apply solution to current problem
 - Critical in program development: find GENERAL algorithm for ALL instances, not specific case
 - Example: Sorting specific list (David, Alice, Carol, Bob) doesn't give general algorithm
 - BUT: Solving special cases can reveal general principles
3. **Stepwise Refinement (Top-down methodology):**
 - Don't conquer entire task at once
 - Break problem into subproblems
 - Decompose subproblems into smaller ones
 - Continue until collection of easily solved subproblems
 - Organizational tool; problem-solving attributes from organization
4. **Bottom-up methodology:**
 - Progresses from specific to general
 - Often complements top-down approach in creative problem solving

Caution - Preconceived Notions:

- Preselected tools can mask problem's simplicity
- Example: Ages of children problem - algebra students try simultaneous equations (dead end)
- Example: Hat in river problem - calculator users get trapped in infinite cycles
- **Solution:** Resist urge to write formulas; adjust perspective

Problem-Solving: The "Foot in the Door" Technique

Problem-solving is rarely about finding the entire solution at once; rather, it is about finding a small, logical starting point – a "toehold" – and expanding it into a complete solution.

1. The Core Strategy: "Get Your Foot in the Door"

The most effective way to approach a complex problem is to identify one undeniable truth or eliminate a set of impossibilities. Once this initial "thrust" is established, the rest of the solution often follows through a chain of logical deductions.

2. Case Study:

The Problem:

- **Predictions:**
 - A: "B will win."
 - B: "D will be last."
 - C: "A will be third."
 - D: "A's prediction is correct."
- **The Rules:** Only one prediction is true, and it was made by the winner.

The Solution Process:

1. **Finding the Toehold:** Notice that A and D made identical claims. Since only one prediction can be true, both A and D **must be lying**.
2. **Elimination:** If A and D lied, neither of them won.
3. **Deduction:** If A lied, B did not win. By process of elimination, **C is the winner**.
4. **Verification:** Since C won, C's prediction is true: **A finished 3rd**.
5. **Final Ordering:** The order must be C _ A _.
 - Remaining spots for B and D are 2nd and 4th.
 - If B was 4th (last), B's prediction would be true. But only C's prediction is true.
 - Therefore, D must be 2nd and B must be 4th.
6. **Final Result:** **C-D-A-B**.

1. Top-Down vs. Bottom-Up Methodologies

These are the two primary organizational strategies used to conquer complex tasks.

A. Stepwise Refinement (Top-Down)

- **Definition:** The process of breaking a large, daunting problem into several smaller, more manageable subproblems.
- **The Workflow:** We start with the "general" goal and progress toward the "specific" details. Each subproblem is further decomposed until it reaches a level of simplicity where it can be easily solved.
- **Application:** Crucial for designing large software systems from scratch, where organization is key.

B. Bottom-Up Methodology

- **Definition:** Progressing from the specific to the general.
- **The Workflow:** Starting with small, known components and combining them to build a larger, more complex system.
- **Application:** Increasingly popular in modern software engineering, where systems are built using prefabricated components (libraries, APIs, and existing modules).

2. The Complementary Nature of Design

While theoretically opposites, these two modes often work together:

- An engineer might use top-down refinement to organize a project's goals.
- The same engineer uses bottom-up intuition to recognize which "pre-built tools" can solve those specific sub-steps.

3. The Trap of Preconceived Notions

A significant obstacle to problem-solving is bringing the wrong tools to the task. Often, students default to complex algebraic formulas when a simple shift in perspective would reveal a trivial solution.

Case Study: The "Hat in the River" Problem

- **The Setup:** A hat falls in a river (current 2.5 mph). A boat travels upstream (4.75 mph relative to water). After 10 minutes, the boat turns around to catch the hat.
- **The Algebraic Trap:** Attempting to calculate distances relative to the shore, the speed of the current, and the boat's speed leads to complex equations or infinite cycles of "where the hat will be next."
- **The Perspective Shift:** Change the Frame of Reference.
 - Think of the river as a moving conveyor belt.
 - If you walk away from an object on a conveyor belt for 10 minutes, it will take you exactly 10 minutes to walk back to it, regardless of how fast the belt is moving past the floor.

- **The Solution:** It takes 10 minutes to catch the hat.

4. Problem-Solving as an Art

The text concludes that Algorithm Discovery is not a rigid science but a "challenging art."

- **Creative Nurturing:** Strictly following one methodology can "quash" the creative skills needed to see a simple solution.
- **The "Toolbox" Mentality:** A good problem solver develops a broad perspective and the wisdom to know when to set aside a calculator and simply rethink the problem's environment.

3. General Approaches to Finding a "Toehold"

George Polya and other mathematicians have suggested specific strategies for when you are stuck:

A. Working Backward

- **Concept:** Start with the desired output and reverse-engineer the steps to get back to the initial input.
- **Example:** In origami, if you cannot figure out how to fold a bird, you take a finished bird and **unfold** it to see the sequence of creases.

B. Decomposition

- **Concept:** Break the problem into smaller, manageable sub-problems. Solve the small parts first to see how they connect to the larger whole.

C. Look for a Similar Problem

- **Concept:** Ask if you have solved something similar before. Can the same logic or "algorithm" be adapted to the current situation?

KEY TERMS TO REMEMBER

Term	Definition
Batch Processing	Executing jobs as a batch without user interaction
Job Queue	FIFO storage organization for waiting jobs
Time-sharing	Multiple users sharing computer simultaneously

Term	Definition
Multiprogramming	Interleaving execution of multiple programs
Multitasking	Single user executing multiple tasks
Real-time Processing	Computer performs task within environmental deadlines
Kernel	Internal part of operating system with basic functions
File Manager	Coordinates mass storage usage
Device Driver	Communicates with controllers/peripheral devices
Memory Manager	Coordinates main memory usage
Virtual Memory	Fictional large memory created through paging
Boot Loader	Program in ROM that loads operating system
Firmware	Software in nonvolatile memory (between hardware and software)
Process	Dynamic activity of executing a program
Process State	Current status (program counter, registers, memory)
Scheduler	Maintains process table, adds/removes processes
Dispatcher	Controls allocation of time slices to processes
Context Switch	Changing from one process to another
Interrupt	Signal indicating end of time slice or external event
Semaphore	Properly implemented flag for resource allocation

Term	Definition
Critical Region	Instruction sequence executed by only one process at a time
Mutual Exclusion	Only one process at a time in critical region
Deadlock	Processes blocked waiting for each other's resources
Spooling	Making nonsharable resources appear sharable
Privileged Mode	CPU mode allowing all instructions
Nonprivileged Mode	CPU mode with limited instruction set
Algorithm	Ordered set of unambiguous, executable steps that terminates
Primitive	Well-defined building block for algorithm representation
Pseudocode	Informal notational system for algorithm development
Stepwise Refinement	Top-down problem decomposition methodology