# Lecture 1: Operating Systems

Yonghao Lee

March 31, 2025

## What is a Computer System

A computer system can be conceptualized as having several distinct layers:

- Hardware: Physical components such as processors, memory, storage devices, input/output devices (keyboard, mouse, display), and network interfaces.
- Operating System: The foundational software layer that manages hardware resources.
- Applications: Software programs designed to perform specific tasks for users.
- Users: Individuals who interact with the computer system through applications.

This structure applies to all computing devices, from embedded systems in aircraft to personal laptops and enterprise servers.

The Operating System (OS) occupies a critical intermediate position between the hardware and the application/user layers, serving as a mediator between them. The OS is a specialized program that, rather than performing end-user tasks directly, creates and maintains the environment in which other programs operate.

The OS possesses two fundamental characteristics:

1. It is **reactive**: The OS responds immediately to events and stimuli, including user input, hardware interrupts, and application requests. Unlike conventional programs that execute a predetermined sequence of operations, the OS continuously monitors the system and reacts to various triggers as they occur.
2. It is **resident**: The OS remains permanently loaded in memory while the computer is running. Its core components (the kernel) stay resident in memory at all times, even when not actively processing tasks. When a computer boots up, the OS is the first software to load, and it continues running until shutdown. This contrasts with application programs that typically launch, perform specific functions, and terminate when their tasks are complete.

Beyond these core properties, the OS fulfills several crucial roles:

- It provides abstraction layers and convenient interfaces that simplify interactions between applications and hardware.

- It enforces security and protection mechanisms to maintain system integrity and prevent unauthorized access.
- It manages system resources efficiently, including processor time, memory allocation, storage access, and peripheral devices.
- It establishes a standardized environment for application development and execution.

These aspects of operating systems will be explored in greater depth throughout this course.

## The OS is More Complex

While the operating system serves as an essential intermediary between hardware and applications, this mediation comes with costs in terms of performance overhead and resource utilization. In certain scenarios, such as when writing specialized programs that require minimal hardware interaction, direct access to the CPU and memory might be preferable to avoid OS intervention.

Modern computing also presents more complex scenarios—multiple operating systems coexisting on a single machine, or a single OS managing multiple CPUs. These configurations will be explored later in the course.

This complexity prompts a fundamental question: Do we really need an operating system?

To answer this, let's consider what would happen without an OS:

Imagine we want to run a program. First, we need to ensure the code reaches the CPU. We know the CPU executes instructions located at specific memory addresses, so we could theoretically write code that places our program at those addresses. But what about interacting with peripherals—displaying output on a monitor, accepting input from a mouse, or accessing a file system? Without an OS, we would need detailed knowledge of exactly which hardware devices are connected and their specific interfaces, making even simple operations exceedingly complex.

Even if we overcame these challenges for one machine, our solution would lack portability. Moving the program to a different computer with different hardware would require extensive rewrites.

Furthermore, without an OS, how would we run multiple programs simultaneously? In a bare-metal environment, we would be limited to executing a single program at a time—a severe limitation for modern computing needs.

Let's examine a concrete example of file access:

With an operating system, reading a file is as simple as: "' return-code = read(fd, buf, nbytes) "'

Without an OS, the same operation becomes enormously complex: 1. Identify linear sector 17403 on disk 2 2. Convert the linear sector number to physical parameters: cylinder, plate, track, and sector 3. Physically move the disk arm to the requested cylinder and plate 4. Wait for the proper sector to rotate into position 5. Read the data 6. Handle any errors or exceptions

This example illustrates how the OS abstracts away tremendous complexity, allowing programmers to focus on application logic rather than hardware details.

The operating system, therefore, serves not merely as an optional intermediary but as an essential layer that enables: - Hardware abstraction and device independence - Resource sharing among multiple applications - Memory protection and security - Standardized interfaces for common operations - Efficient utilization of system resources

These benefits far outweigh the performance costs of OS mediation for the vast majority of computing scenarios.

## Abstraction

One of the most fundamental functions of an operating system is to provide abstraction—converting complex hardware mechanisms into simple, intuitive interfaces. The OS creates logical constructs that do not physically exist in hardware but provide users and applications with conceptual models that are far easier to understand and utilize. In doing so, it effectively hides the underlying complexities of hardware interaction.

This abstraction is not merely cosmetic; the intricate low-level operations still need to be performed. The operating system takes responsibility for translating between the simplified abstractions it presents and the complex reality of hardware operations.

Consider file storage as a concrete example:

- **Hardware Reality:** Storage devices (like hard drives or SSDs) organize data in fixed-size blocks, addressed by physical locations on the storage medium (sectors, tracks, cylinders).
- **OS Abstraction:** The operating system presents storage as a collection of named files of arbitrary sizes. Users can address data by byte offset and length, and files are organized in a hierarchical directory structure.

This abstraction transforms the raw, physical characteristics of storage hardware into a logical model that is intuitive for humans to understand and for software to manipulate. Users never need to consider cylinder-head-sector addressing or block sizes; they simply interact with files and folders.

## The Evolution of Computer Architecture and Operating Systems

This section examines the historical development of computer architecture and operating systems, focusing on how design decisions evolved in response to technological advancements.

### First Generation Computers (1945 - 1955)

During the era of first generation computers, operating systems were nonexistent due to the primitive nature of the hardware. These early machines utilized magnetic drums for memory storage and vacuum tubes for processing. The simplicity of these systems meant there were no concurrent processes or resources requiring management.

## Second Generation Computers (1955 - 1965)

The second generation marked a significant technological shift with the introduction of transistors, which replaced vacuum tubes. These computers implemented batch processing systems capable of executing multiple programs stored on magnetic disks. This advancement in capability necessitated the development of resource management systems.

Operation of a second generation computer required specialized personnel and equipment: a dedicated operator (distinct from the end user) managed the system; a card reader processed program instructions; and a complex workflow involved transferring data from input media to system tapes, then directing output to printers for final processing.

### Spooling

Simultaneous Peripheral Operation On-Line (SPOOLING) represented a critical advancement in resource utilization. This technique enabled overlap between input/output operations and computational processes. SPOOLING emerged from the observation that after completing a computational job and initiating output printing, the central processing unit remained idle despite being fully operational.

The innovation involved reading subsequent jobs from punch cards directly into disk storage (forming a job queue) while simultaneously printing results from previous jobs from disk to printer. This mechanism created a persistent job pool from which the operating system could intelligently select the next task for processing, significantly enhancing CPU utilization and overall system throughput.

Since we have been talking about CPU utilization, it naturally indicates that we might be interested in doing some optimization, for this, we need to understand some terminology.

### Performance Terminology

1. **Latency/Response Time:** The duration required to complete a specific job or process a unit of data, typically measured in seconds or milliseconds.

2. **Throughput:** The rate at which a system performs work or processes data, commonly expressed as units processed per time interval (e.g., bytes/second, transactions/minute).

3. **Utilization:** The proportion of time during which a resource (particularly the CPU) is engaged in productive work. This specifically excludes time spent executing operating system routines.

4. **Overhead:** The additional computational resources, memory, or time consumed by auxiliary processes necessary to accomplish a primary task, but not directly contributing to the end result.

5. **CPU Usage:** The total fraction of time the CPU spends executing instructions, comprising both utilization (application processing) and overhead (system processes). Expressed mathematically as: CPU Usage = Utilization + Time Overhead. Note that in this course, we assume continuous CPU operation and will not extensively address this metric.

## Third Generation Computers (1965 - 1980)

The third generation introduced memory architectures substantially similar to contemporary designs. These systems maintained multiple jobs concurrently in main memory alongside the operating system.

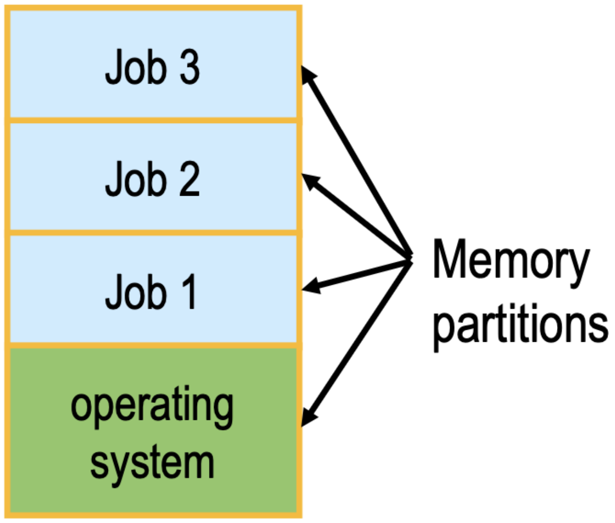


Figure 1: Memory Partitioning in Third Generation Systems

While a sequential execution approach (completing job 1, then job 2, then job 3) was possible, this methodology suboptimized system performance. The primary objectives became minimizing average response time while maximizing CPU utilization. To achieve these goals, two fundamental techniques were introduced:

1. **Multi-programming:** This technique allowed the system to switch execution to another program whenever the current program encountered an instruction requiring peripheral wait time. By alternating between ready processes, CPU utilization increased significantly.

2. **Time-sharing (multi-tasking/time-slicing):** This approach addressed scenarios where process execution times varied dramatically (e.g., a 10-hour computation versus a 1-second task). Rather than allowing a long-running process to monopolize resources, the system would forcibly switch between processes after predetermined time intervals, even when no peripheral wait occurred.

The implementation of multi-programming and time-sharing relied on several critical assumptions:

- CPU resources should be continuously utilized to prevent inefficiency
- CPU processing speed significantly exceeds I/O operation speed
- Memory capacity is sufficient to accommodate multiple concurrent programs
- Peripherals possess direct memory access (DMA) capabilities
- The system must manage multiple concurrent tasks

These assumptions necessitated the development of several essential operating system features:

- **I/O Primitives:** System calls that facilitate controlled access to peripheral devices
- **Memory Management:** Mechanisms to preserve process state during context switches. For example, when the operating system switches from job 1 to job 2, it must save job 1's register values (such as variables x and y in a computation) and restore them when job 1 regains CPU access.
- **Scheduler:** A subsystem that governs CPU allocation, determining which processes enter execution, which are suspended, and for what duration. Various scheduling algorithms emerged, including uniform time-slice allocation and priority-based approaches.
- **Spooler:** A subsystem designed to manage interactions with slower I/O devices, optimizing their utilization.

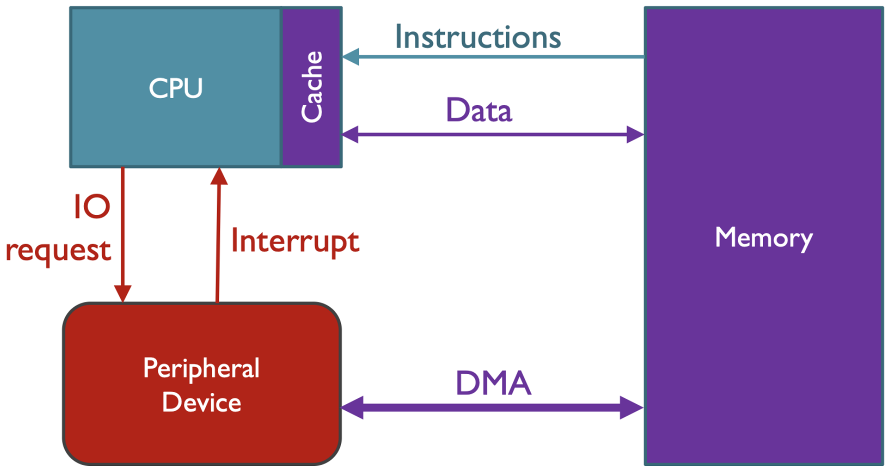


Figure 2: von-Neumann Architecture

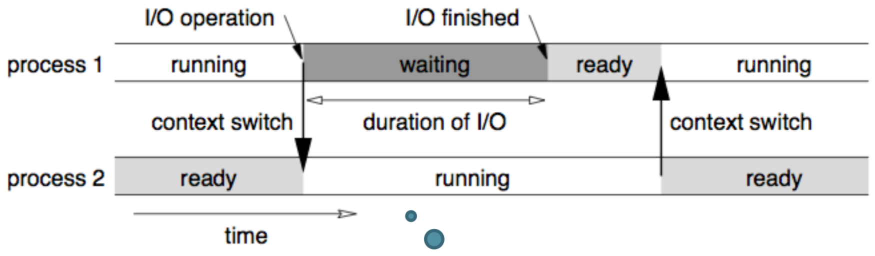


Figure 3: Multi-Programming

Note that the dark arrows are the actions of the OS.

### Fourth Generation Computers (1980 - 2000)

The fourth generation of computing introduced several transformative technologies that fundamentally altered the computing landscape. Key innovations included graphical user interfaces (GUIs), which revolutionized human-computer interaction; networked computing architectures, which enabled distributed processing and resource sharing; and web-based computing paradigms, which laid the foundation for the modern internet ecosystem.

### Fifth Generation Computers (2000 - Present)

Fifth generation computing has been characterized by two major architectural shifts: the rise of multiprocessor and multicore architectures, which dramatically increased parallel processing capabilities; and cloud computing, which abstracted hardware resources into virtualized, on-demand services accessible via networks. These innovations have enabled unprecedented computing scale and flexibility.

## Main Components of Contemporary Operating Systems

Modern operating systems provide a comprehensive suite of services and capabilities that have evolved to address the complex requirements of today's computing environments:

- **Program Execution:** The fundamental ability to load and run application software
- **Process Management:** Sophisticated control of multiple concurrent processes, including scheduling, synchronization, and communication mechanisms
- **Memory Management:** Advanced techniques for memory allocation, protection, and optimization. This includes virtual memory systems that create the illusion of abundant memory resources even when physical memory is limited (e.g., satisfying a `malloc()` request for millions of bytes despite physical constraints)
- **I/O and Device Management:** Standardized interfaces and device drivers that abstract hardware complexity while optimizing peripheral performance

- **File System:** Hierarchical organization of data with consistent access methods, permissions, and metadata management
- **User Interface:** Multiple interaction paradigms including command-line interfaces (CLI), shells, and graphical user interfaces (GUI)
- **Networking:** Protocol implementations and services supporting both local and wide-area communications
- **Security:** Authentication, authorization, encryption, and other protective mechanisms to ensure data integrity and system safety
- **Error Detection and Recovery:** Monitoring, logging, and fault-tolerance mechanisms to maintain system stability
- **Resource Accounting:** Tracking of system resource usage for performance optimization and billing purposes
- **Resource Allocation:** Intelligent distribution of computational resources among competing processes and services

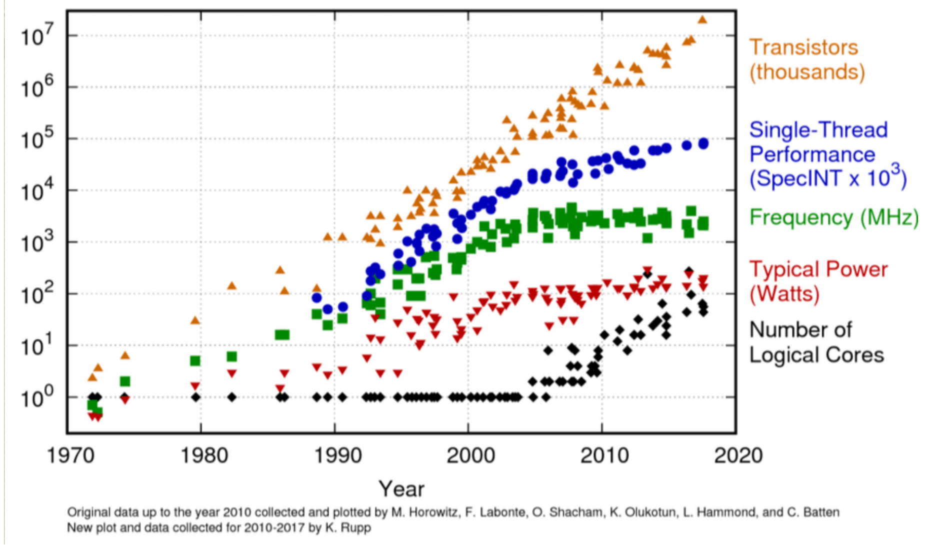


Figure 4: Processor Performance Growth and the Shift to Multi-core Architectures (Moore's Law Transition)

The figure illustrates a critical inflection point in computing hardware evolution. Until approximately 2005, processor performance exhibited exponential growth primarily through clock speed increases and transistor density improvements, closely following Moore's Law. However, this trajectory reached physical limitations as semiconductor manufacturing approached atomic-scale constraints, where silicon transistors could not be made substantially smaller without encountering quantum effects and thermal dissipation challenges.

Consequently, the industry pivoted toward parallelism rather than continued single-core performance scaling. This architectural shift is evident in the graph's plateau, where manufacturers began increasing the number of computational cores rather than enhancing individual core performance.

## Multi-Core Architectures

As single-processor performance scaling reached physical limitations, the computing industry shifted toward parallel processing through multi-core architectures. Contemporary processors incorporate multiple independent computational cores on a single die, with even entry-level mobile devices now commonly featuring dual or quad-core configurations.

This architectural evolution presents several significant challenges for operating system design:

- **Resource Sharing Complexity:** Some hardware components (such as L3 cache and memory bus) are shared across cores, while others (such as L1/L2 caches) remain dedicated to individual cores, requiring sophisticated resource management
- **Scheduling Complexity:** The OS must determine optimal task distribution across available cores while considering factors such as cache locality, thermal constraints, and power efficiency
- **Synchronization Requirements:** Multi-core environments necessitate robust mechanisms for coordinating concurrent access to shared resources
- **Power Management:** Balancing performance and energy efficiency requires dynamic activation and deactivation of cores based on computational demands

## Virtualization

Virtualization technology creates abstracted representations of computing resources that decouple logical structures from their physical implementations. This paradigm allows a single physical resource to manifest as multiple logical entities, or conversely, multiple physical resources to appear as a single logical entity.

This concept manifests in numerous contexts:

- **Storage Virtualization:** Physical storage devices are partitioned into logical volumes with independent addressing and management
- **Memory Virtualization:** The operating system presents applications with contiguous memory address spaces regardless of physical memory fragmentation
- **Network Virtualization:** Virtual Private Networks (VPNs) create secure, logically isolated network connections across shared physical infrastructure

The most prominent virtualization application is the virtual machine (VM), facilitated by hypervisor software (e.g., VMware, Hyper-V, KVM). The hypervisor creates and manages multiple isolated virtual environments, each perceiving itself as operating on dedicated hardware. Modern virtualization extends to containers, which provide application isolation with lower overhead than full VMs by sharing the host OS kernel.

Virtualization delivers several critical operational benefits:

- **Snapshots:** The system state can be captured at precise moments, enabling rapid recovery to known-good configurations
- **Migration:** Running environments can be transferred between physical hosts with minimal service interruption
- **Dynamic Scaling:** Computing resources can be elastically provisioned to match fluctuating workload demands
- **Security Isolation:** Compromised virtual environments remain contained, preventing lateral movement to other systems

It is important to note that virtualization incurs performance overhead due to the additional abstraction layer, though hardware-assisted virtualization features in modern processors have significantly reduced this penalty.

## Cloud Computing

Cloud computing represents a service delivery model in which computing infrastructure, platforms, and applications are accessed as on-demand services via network connections. This paradigm shifts capital expenditure to operational expenditure by allowing organizations to utilize provider-managed infrastructure without direct ownership.

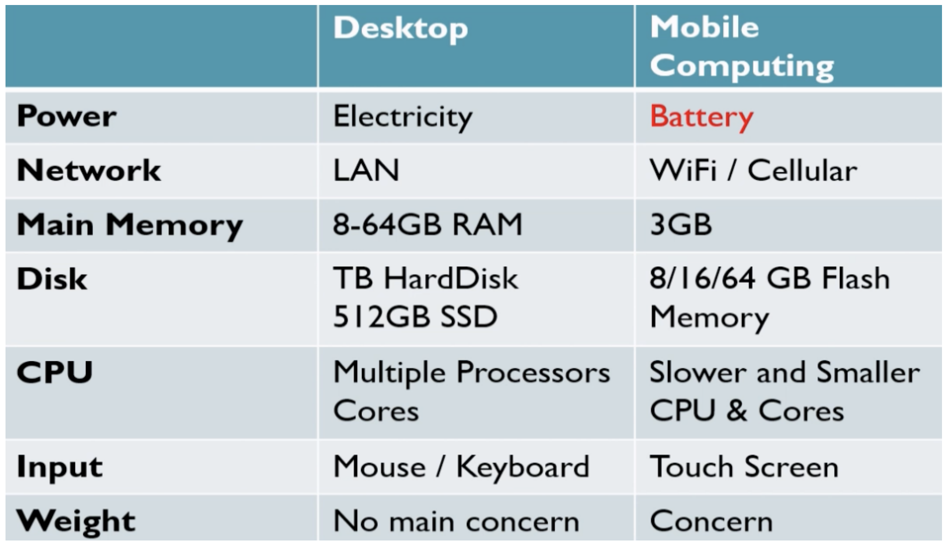
The cloud model encompasses several service tiers:

- **Infrastructure as a Service (IaaS):** Providing virtualized computing resources
- **Platform as a Service (PaaS):** Offering development and deployment environments
- **Software as a Service (SaaS):** Delivering fully functional applications

Cloud architectures leverage virtualization extensively to achieve resource pooling, rapid elasticity, and measured service delivery.

## Mobile Computing

Mobile operating systems address fundamentally different design constraints than their desktop counterparts. Power efficiency, thermal management, and form factor considerations take precedence in these environments.

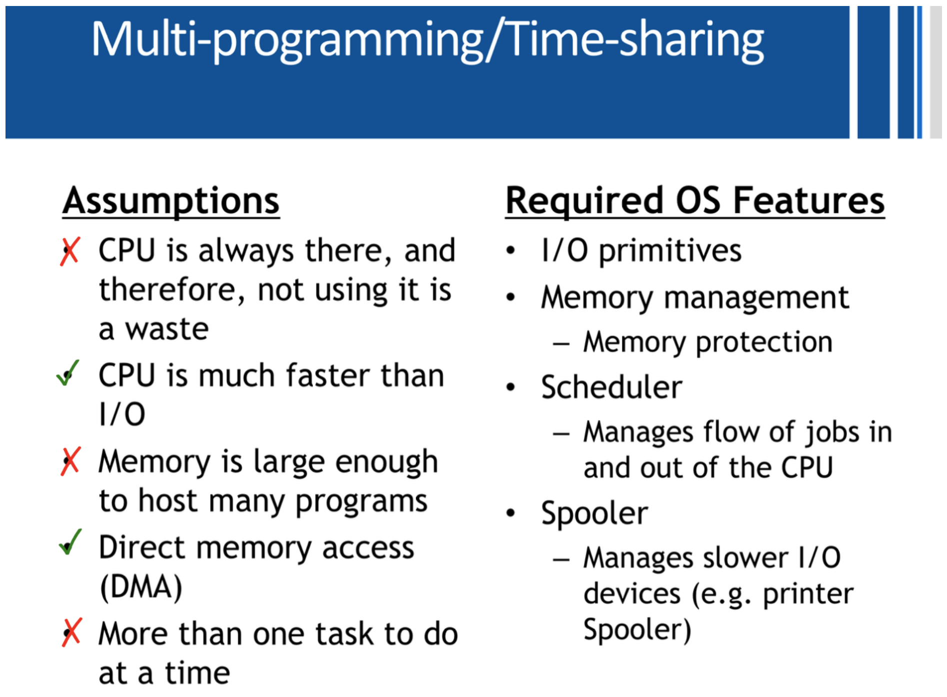


| | Desktop | Mobile Computing |
|---|---|---|
| **Power** | Electricity | Battery |
| **Network** | LAN | WiFi / Cellular |
| **Main Memory** | 8-64GB RAM | 3GB |
| **Disk** | TB HardDisk 512GB SSD | 8/16/64 GB Flash Memory |
| **CPU** | Multiple Processors Cores | Slower and Smaller CPU & Cores |
| **Input** | Mouse / Keyboard | Touch Screen |
| **Weight** | No main concern | Concern |

Figure 5: Mobile Computing Architecture

The assumptions that underpin desktop operating system design must be reconsidered for mobile platforms:

- **Power Management:** Unlike continuously powered desktop systems, mobile devices operate on finite battery capacity, necessitating aggressive power conservation
- **Processor Availability:** Mobile processors frequently transition between active and low-power states to conserve energy, invalidating the desktop assumption of continuous CPU availability
- **Memory Constraints:** Limited physical memory requires more aggressive reclamation policies
- **Thermal Limitations:** Compact form factors with minimal cooling capacity impose strict thermal constraints



## Multi-programming/Time-sharing

**Assumptions**

- ✗ CPU is always there, and therefore, not using it is a waste
- ✓ CPU is much faster than I/O
- ✗ Memory is large enough to host many programs
- ✓ Direct memory access (DMA)
- ✗ More than one task to do at a time

**Required OS Features**

- I/O primitives
- Memory management
  - Memory protection
- Scheduler
  - Manages flow of jobs in and out of the CPU
- Spooler
  - Manages slower I/O devices (e.g. printer Spooler)

Figure 6: Mobile-Optimized Multi-programming Model

These constraints have driven the development of specialized mobile operating systems with distinctive architectural features optimized for resource-constrained environments.