COMPUTER ORGANIZATION AND OPERATING SYSTEMS

Abstract

This chapter examines the intricate relationship between computer hard- ware architecture and operating system (OS) functionalities, emphasizing how resources are managed to achieve efficiency and reliability. It begins with the von Neumann architecture, detailing the roles of the control unit, arithmetic logic unit (ALU), memory hierarchy (from registers to hard disk), and the critical function of system buses in data transfer [1]. The chapter then explores the OS as an intermediary between hardware and user applications, highlighting core responsibilities such as process scheduling, memory management, and input/out- put operations [2, 3]. Special attention is given to the kernel's role in resource protection, multitasking, and security, as well as to mechanisms like paging and direct memory access (DMA) for efficient data handling. Case studies, including Linux kernel multitasking, and visual aids such as the OSI model, provide practical insights into real-world implementations. By integrating hardware and software perspectives, this chapter equips readers with a holistic understanding of how modern computing systems orchestrate complex tasks and maintain robust performance

Keywords: CPU, Memory hierarchy, Process scheduling, Paging, DMA

Authors

Monu Sharma
Valley Health, Winchester
Virginia, USA.
monufscm@gmail.com;

Amit Dhiman
HCL America Inc.
Dallas, Texas, USA.
amittdhiman91@gmail.com;

Navom Saxena
Senior Machine Learning Engineer
Meta, New York, USA.
navom.saxena@gmail.com;

Anushka Raj Yadav
Department of Computer Science
Chandigarh University, Gharuan
Mohali, 140413, Punjab, India.
ay462744@gmail.com;

Shubneet
Department of Computer Science
Chandigarh University, Gharuan
Mohali, 140413, Punjab, India.
jeetshubneet27@gmail.com;

I. INTRODUCTION

The symbiotic relationship between computer hardware and operating systems (OS) forms the foundation of modern computing. Hardware, comprising physical components like the CPU, memory, and I/O devices, provides the computational power and resources necessary for task execution. The OS, as a software layer, orchestrates these resources to deliver functionality, security, and usability. This interdependence ensures efficient resource allocation-for instance, the OS relies on the CPU's arithmetic logic unit (ALU) for computations, while hardware components depend on the OS for task scheduling and memory management. This synergy enables systems to balance performance, reliability, and user accessibility, from embedded devices to enterprise servers [4].

Historically, computing evolved from rudimentary batch processing to sophisticated multitasking environments. In the 1950s–60s, batch systems processed jobs sequentially using punched cards, requiring minimal user interaction. The 1970s introduced time-sharing, allowing multiple users to access mainframes simultaneously through terminals, a paradigm that prioritized resource fairness over raw speed. By the 1980s, multitasking OSes like UNIX enabled single users to run concurrent applications, leveraging advancements in CPU clock speeds and memory hierarchy. Modern systems integrate preemptive scheduling and virtual memory, with kernels like Linux managing billions of operations daily across diverse hardware architectures [5].

Key milestones in this evolution include

- **Batch Processing:** Jobs executed in sequence without user input (e.g., IBM's OS/360).
- **Time-Sharing:** Interactive access via terminals (e.g., MIT's CTSS).
- **Multitasking:** Concurrent application execution (e.g., Windows NT, Linux).
- **Virtualization:** Hardware abstraction for cloud computing (e.g., VMware, Docker).

This chapter explores these concepts through the following structure:

- **Hardware Components:** Von Neumann architecture, CPU subsystems, and memory hierarchy.
- **OS Functions:** Process scheduling, paging, and I/O management.
- **I/O Systems:** Interrupt handling, DMA controllers, and network protocols.
- **Case Study:** Linux kernel's multitasking implementation.
- **Visual Aids:** Diagrams of hardware-OS interactions and layered architectures.

II. HARDWARE COMPONENTS

Modern computing systems rely on tightly integrated hardware architectures and memory subsystems to balance performance, cost, and energy efficiency. This section examines the von Neumann model, CPU subsystems, memory hierarchies, and key performance metrics.

1. **Von Neumann Architecture:** The von Neumann architecture forms the basis of most general-purpose computers, unifying program instructions and data in a single memory system. As shown in Figure 1, it comprises five components:

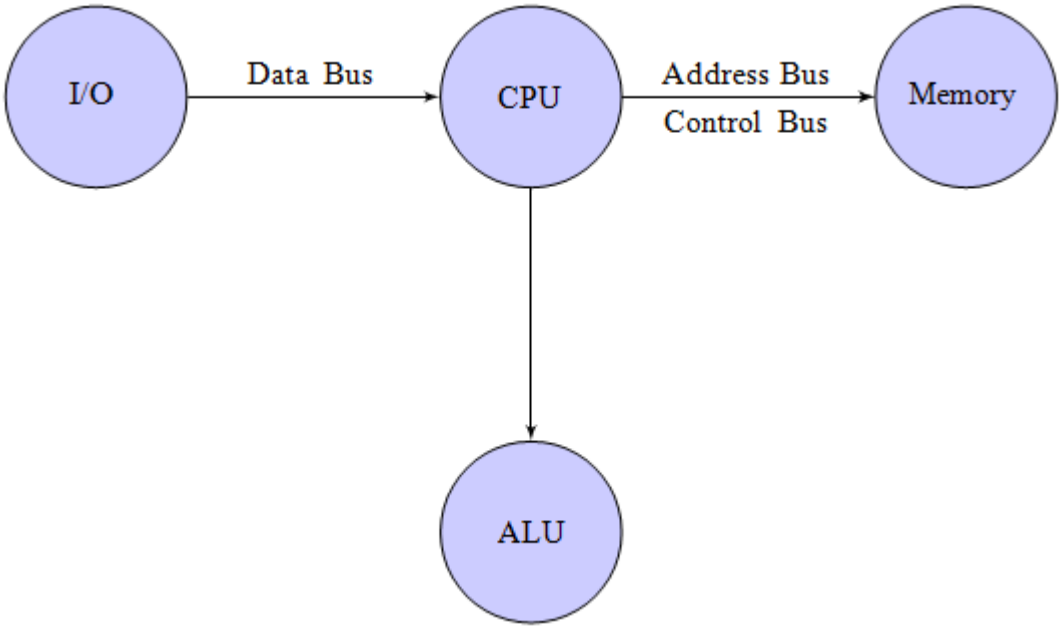

Figure 1: Von Neumann architecture with unified memory and bus structure

- **CPU:** Executes instructions via ALU and control unit
- **Memory:** Stores instructions and data (RAM/SSD/HDD)
- **Buses:**
  - **Data:** Transfers operands/results (bidirectional)
  - **Address:** Specifies memory locations (unidirectional)
  - **Control:** Manages operations (interrupts, timing)

This design introduces the *von Neumann bottleneck*, where shared buses limit concurrent instruction/data access [6, 7].

CPU Subsystems: The CPU executes programs through coordinated operation of two subsystems:

Arithmetic Logic Unit (ALU):
- Performs integer arithmetic (add, subtract)
- Executes logic operations (AND, OR, XOR)
- Handles bit-shifting and comparisons

Control Unit: Manages the *fetch-decode-execute cycle*:
- **Fetch**: Copies instruction address from PC to MAR, retrieves instruction via data bus to MDR/CIR [**?**].
- **Decode**: Splits instruction into opcode (operation) and operand (data/address).
- **Execute**: Routes operands to ALU/memory, stores results in registers/memory.

Memory Hierarchy: Modern systems employ a layered memory hierarchy to optimize speed/cost tradeoffs (Table 1):

Table 1: Memory hierarchy characteristics

Level	Latency	Size	Cost/GB
Registers	0.1ns	1KB	$10,000
L1 Cache	0.5ns	64KB	$1,000
L2 Cache	5ns	512KB	$500
RAM	80ns	16GB	$10
SSD	100 μs	1TB	$0.20

- **Registers:** CPU-integrated storage (e.g., PC, MAR)
- **Cache:** SRAM-based L1/L2/L3 reduce RAM access latency
- **RAM:** DRAM for volatile program/data storage
- **SSD/HDD:** Non-volatile bulk storage [8]

Performance Metrics: Key metrics for evaluating hardware performance:

- **Clock Speed:** GHz rate (e.g., 3.5 GHz = 3.5 billion cycles/sec)
- **IPC:** Instructions per cycle (higher = better parallelism)
- **CPI:** Cycles per instruction (lower = better efficiency)
- **Amdahl's Law:** Speedup = $\frac{1}{(1-P)} + \frac{P}{s}$ where P = optimized fraction, S = speedup factor [9]

For a CPU with 30% vectorized code (10 x faster):

Speedup = $\frac{1}{(1-0.3)+\frac{0.3}{10}} = 1.27x$

III. OPERATING SYSTEM FUNCTIONS

Operating systems serve as intermediaries between hardware and applications, managing resources and providing services. This section examines key OS functions: process scheduling, memory management, file systems, and security mechanisms.

Process Scheduling: The scheduler determines which processes receive CPU time and in what order. Modern operating systems implement various scheduling algorithms to optimize system performance:

Round Robin (RR) Scheduling allocates CPU time to processes in a cyclic manner, with each process receiving a fixed time quantum before being preempted. This approach ensures fairness by preventing any single process from monopolizing the CPU. However, its performance heavily depends on the time quantum value-smaller values improve responsiveness but increase context switching overhead [10].

Key characteristics of Round Robin scheduling include:

- **Fairness:** All processes receive equal CPU time
- **Responsiveness:** Short time slices maintain system responsiveness
- **Overhead:** Context switching between processes consumes CPU cycles

As time quantum increases, RR scheduling approaches FCFS (First-Come-First- Served) behavior; as it approaches infinity, RR becomes identical to FCFS. Most implementations use multilevel queue scheduling, organizing processes into multiple queues based on their characteristics (CPU-bound or I/O-bound) and applying different scheduling algorithms to each queue.

Memory Management: Memory management involves allocating and tracking physical memory resources while providing processes with a consistent addressing scheme.

Paging divides physical and virtual memory into fixed-size blocks (pages), map- ping virtual addresses to physical ones through page tables. Since constant table lookups would slow the system, a specialized cache called the Translation Lookaside Buffer (TLB) stores recent address translations [11].

The TLB functions as follows:

- When translating a virtual address, the MMU first checks the TLB
- On a TLB hit, the physical address is retrieved immediately
- On a TLB miss, the page table in main memory is consulted
- The new translation is added to the TLB for future reference

Virtual memory extends physical RAM by using disk space as an overflow, allowing programs to use more memory than physically available. The OS manages page tables, allocates physical memory, and handles exceptions raised by the Memory Management Unit (MMU).

Segmentation divides memory into variable-sized segments based on logical divisions (code, data, stack). Modern systems often combine segmentation with pag- ing (segmented paging), using pages to describe components of segments for easier management.

File Systems: File systems provide organized storage and retrieval mechanisms for data.

Ext4 vs. NTFS represent two widely-used file systems for Linux and Windows respectively. Ext4 produces less fragmentation than NTFS, enabling faster data reads, while NTFS offers features like online disk checking and user quotas [12].

Key differences include:

- **Structure:** Ext4 uses inodes while NTFS uses Master File Table (MFT)
- **Size Limits:** Ext4 supports volumes up to 1EB and files up to 16TB; NTFS theoretically supports volumes up to $2^{64} - 1$ clusters
- **Performance:** Ext4 generally provides better performance for multiple concurrent file operations
- **Fragmentation:** Ext4 minimizes fragmentation inherently while NTFS requires periodic defragmentation

Inode handling in Ext4 stores metadata including file permissions, timestamps, and pointers to data blocks. Each inode has a unique number that serves as an identifier for the file or directory it represents.

Security: OS security mechanisms protect system integrity by controlling resource access and isolating processes.

Kernel-mode vs. User-mode privileges establish a security boundary. In kernel mode, code has unrestricted access to hardware and can execute any CPU instruction. In user mode, applications run in isolated virtual address spaces with limited hardware access [13].

When a user-mode application requires privileged operations, it makes system calls that temporarily transfer control to kernel-mode code. This separation ensures that:

- Applications cannot directly access critical hardware
- Process crashes in user mode don't affect the entire system
- Malicious code has limited ability to compromise the system

Buffer overflow mitigation is critical in kernel code where memory management errors can lead to system crashes or privilege escalation. Kernel drivers must carefully validate buffer sizes and implement proper bounds checking to prevent attackers from overwriting adjacent memory regions with malicious code.

The growing integration of AI and IoT into critical infrastructure, such as digital payment systems, underscores the need for robust OS-level security. Recent research demonstrates that AI-driven, IoT-enabled platforms can proactively detect and mitigate security threats in real time, leveraging advanced fraud detection algorithms and device-level monitoring to strengthen system resilience against cyberattacks[14].

Modern operating systems employ additional protection mechanisms including Address Space Layout Randomization (ASLR), Data Execution Prevention (DEP), and stack cookies to reduce the exploitability of memory corruption vulnerabilities.

IV. I/O SYSTEMS AND NETWORKING

Input/output (I/O) systems and networking protocols are critical for managing data flow between hardware components and enabling efficient communication across net- works. This section examines interrupt handling, DMA controllers, network models, and modern storage protocols.

1. **Interrupt Handling:** Interrupts signal the CPU to pause execution and handle high-priority events. Key components include:
   - **Interrupt Requests (IRQs)**: Hardware-generated signals (e.g., keyboard input, network packets) assigned priority levels.
   - **Interrupt Service Routines (ISRs)**: Short, time-sensitive code blocks that handle interrupts.
   - **Latency Challenges**: Prolonged ISR execution delays other tasks. Techniques like *First-Level Interrupt Handlers (FLIHs)* quickly log events, deferring complex operations to *Second-Level Interrupt Handlers (SLIHs)* to minimize CPU stall time [15].

DMA Controllers: Direct Memory Access (DMA) controllers bypass CPU involvement in bulk data transfers:

- **GPU Texture Loading:** Transfers texture data from SSD to GPU memory via PCIe lanes, leveraging DMA for 7 GB/s throughput.
- **Operation Steps**
  - Device sends DMA request.
  - Controller arbitrates bus access.
  - Data moves directly between device and memory.
- **Efficiency:** Reduces CPU load by 80% compared to programmed I/O [16].

OSI Model and TCP/IP: The OSI model standardizes network communication across seven layers, while TCP/IP prioritizes practicality:

TCP/IP's streamlined design enables faster deployment but sacrifices granularity in error handling and encryption [15].

Table 2: OSI vs. TCP/IP Models

OSI Layer	Purpose	TCP/IP Layer
7: Application	HTTP, FTP	Application
6: Presentation	Encryption	(Merged)
5: Session	Connection management	(Merged)
4: Transport	TCP/UDP	Transport
3: Network	IP routing	Internet
2: Data Link	MAC addressing	Link
1: Physical	Hardware signaling	Physical

Case Study: NVMe Protocol for SSDs: Non-Volatile Memory Express (NVMe) optimizes SSD communication:

- **Parallelism:** Supports 64K command queues vs. SATA's single queue.
- **Latency:** Reduces read/write delays to 2.8 μs (vs. SATA's 30–100 μs).
- **DMA Integration:** Uses Physical Region Page (PRP) lists to map host memory directly to SSD controllers, bypassing CPU data copying.

NVMe's PCIe interface achieves 7 GB/s throughput, making it ideal for AI training and real-time analytics [16]

V. CASE STUDY: LINUX KERNEL MULTITASKING

The Linux kernel's multitasking capabilities rely on sophisticated algorithms and memory management techniques to balance performance, fairness, and resource efficiency across thousands of concurrent processes. This case study examines four pivotal components enabling this functionality.

Completely Fair Scheduler (CFS): CFS ensures equitable CPU time distribution using a Red-Black tree to organize tasks by vruntime (virtual runtime). Key features:

- **Red-Black Tree:** Tasks are sorted by vruntime, with the leftmost node (lowest vruntime) scheduled next. Insertions/deletions occur in $O(\log n)$ time.
- **vruntime Calculation:**

$$\text{vruntime} = \text{actual\_runtime} \times \frac{NICE\_0\_LOAD}{task-weight}$$

Higher-priority tasks (lower task_weight) accumulate vruntime slower, gaining more CPU time.

- **Fairness:** Tasks waiting longer are prioritized, preventing starvation [16].

Memory Management: Linux combines two allocators for efficient memory utilization:

- **Buddy Allocator:** Manages physical memory in power-of-two blocks. Splits blocks to fulfill requests (e.g., 4KB → 2×2KB), coalescing freed blocks to avoid fragmentation.
- **Slab Cache:** Pre-allocates frequently used kernel objects (e.g., inodes, task structs) in contiguous memory slabs. Reduces initialization overhead by reusing initialized objects [17].

Inter-Process Communication (IPC): Linux supports three primary IPC mechanisms:

- **Pipes:** Unidirectional channels between related processes (e.g., shell command chaining). Implemented as kernel-managed circular buffers.
- **Sockets:** Bidirectional network/domain communication (e.g., TCP/IP). Supports asynchronous data transfer across machines.
- **Shared Memory:** Multiple processes access the same memory region via shmget()/shmat(), synchronized using semaphores.

These methods enable efficient data sharing while maintaining process isolation [18].

Kernel Modules: Loadable kernel modules (LKMs) dynamically extend kernel functionality:

- **DMA Controllers:** Modules like dmaengine. ko manage direct memory access, offloading bulk transfers (e.g., NVMe SSD I/O) from the CPU.
- **Benefits:** Modules can be loaded/unloaded without rebooting, reducing down-time. Custom drivers add hardware support (e.g., GPUs, NICs).

The modprobe tool handles module dependencies and version checks [2].

VI. VISUAL AIDS

Visual representations are invaluable for understanding complex computer system concepts. This section presents three key diagrams: a CPU-memory hierarchy, the OSI network model, and a page table walk flowchart.

CPU and Memory Hierarchy

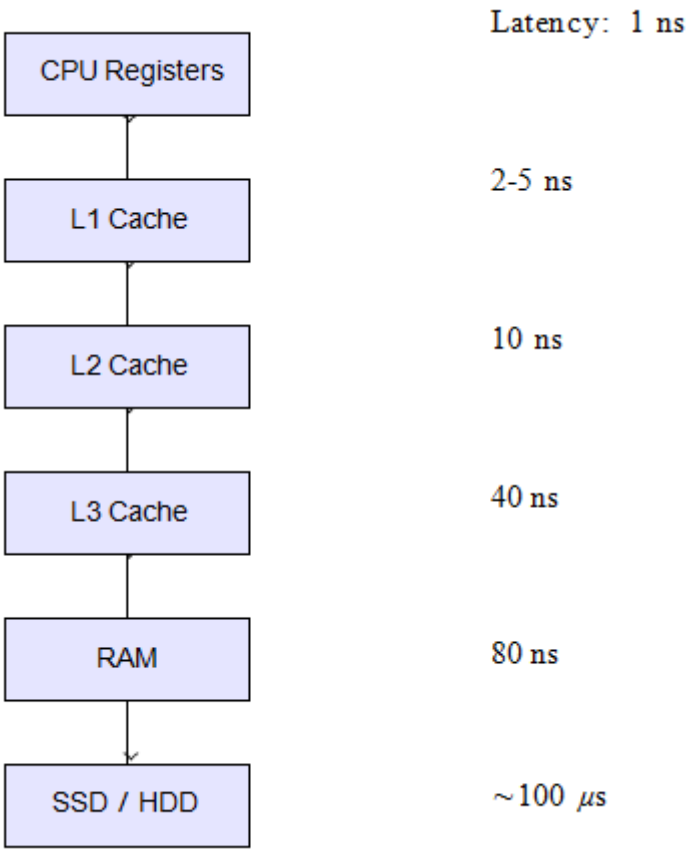

Figure 2: CPU and memory hierarchy with typical access latencies

OSI Model

Layer 7: Application (HTTP, FTP)
Layer 6: Presentation (TLS/SSL)
Layer 5: Session (RPC, SIP)
Layer 4: Transport (TCP, UDP)
Layer 3: Network (IP, ICMP)
Layer 2: Data Link (Ethernet, MAC)
Layer 1: Physical (RJ45, WiFi)

Figure 3: OSI model layers with protocol examples

Page Table Walk

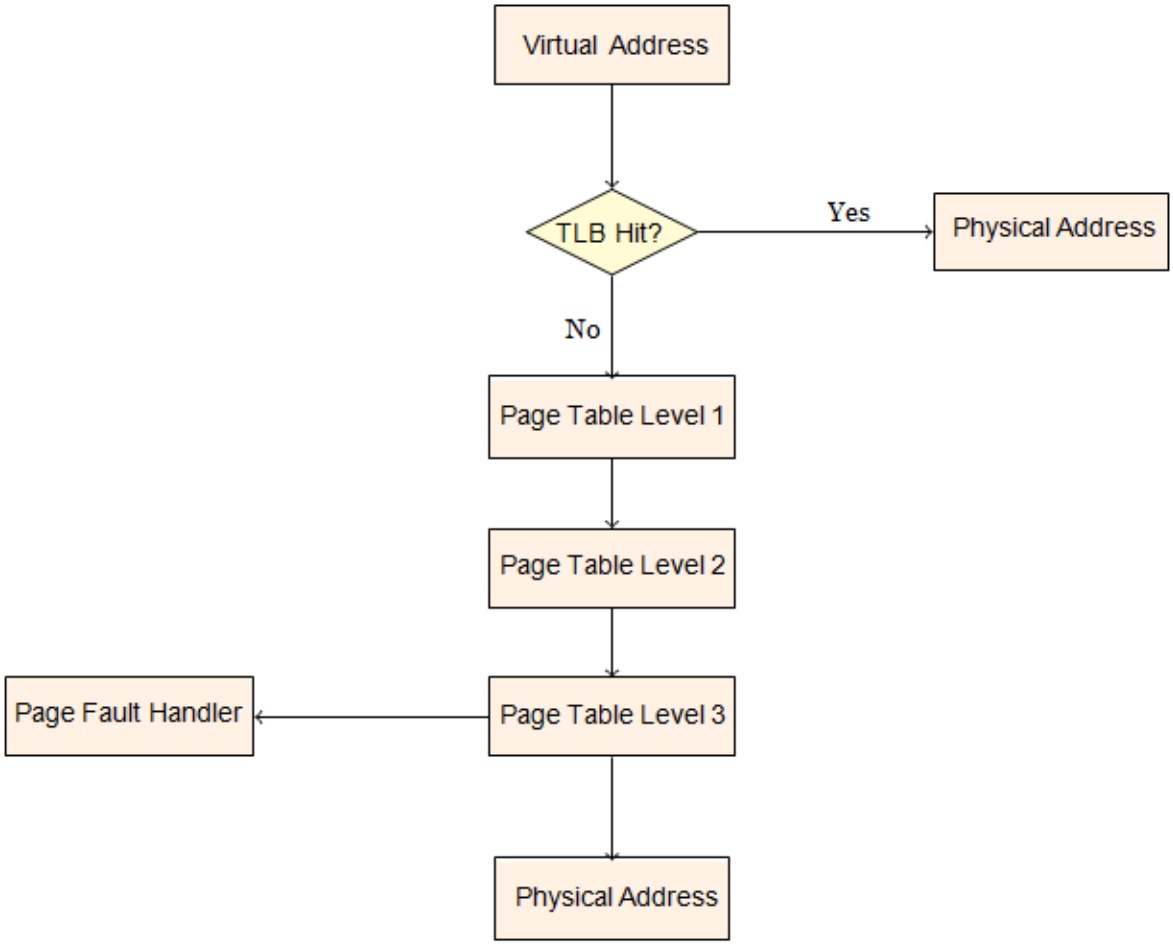

Figure 4: Flowchart for virtual-to-physical address translation (page table walk)

Visual diagrams like these help bridge abstract theory and practical understanding, making it easier to grasp architectural and operational details in computer systems [7].

VII. EXERCISES

This section provides practical exercises on core operating system concepts, focusing on process scheduling, memory management, and high-performance I/O.

1. Simulate Round-Robin Scheduling with a Python Queue

Problem: Implement a round-robin scheduler for three processes with burst times of 8, 10, and 6 ms, using a time quantum of 2 ms.

Solution:

```
from collections import deque class Process:

def _init_(self, name, burst):
```

```
        self.name = name
        self.burst = burst

def round_robin(processes, quantum):
    queue = deque(processes)
    t = 0
    while queue:
        p = queue.popleft()
        exec_time = min(quantum, p.burst)
        print(f"{p.name} runs from {t} to {t+exec_time} ms")
         t += exec_time
        p.burst -= exec_time
        if p.burst > 0:
                queue.append(p)
        else:
                print(f"{p.name} completes at {t} ms")

# Example
plist = [Process('P1', 8), Process('P2', 10), Process('P3', 6)]
round_robin(plist, 2)
```

Explanation: Each process receives up to 2 ms per turn. If unfinished, it rejoins the queue. This simulates fair CPU sharing and highlights context switching overhead typical of round-robin scheduling [3].

2. Calculate Effective Memory Access Time with TLB Hit Ratios

Problem: Given a TLB hit ratio of 90%, TLB lookup time of 10 ns, and memory access time of 100 ns, what is the effective memory access time (EAT) for a single-level page table?

Solution:
EAT = (TLB hit ratio) × (TLB time + Memory time)
+ (TLB miss ratio) × (TLB time + 2 × Memory time)
= 0.9 × (10 + 100) + 0.1 × (10 + 200)
= 0.9 × 110 + 0.1 × 210
= 99 + 21 = 120 ns

Explanation: On a TLB miss, the system must access both the page table and the actual data, resulting in two memory accesses [19].

3. DMA's Role in a PCIe-based NVMe SSD Case Study

Problem: Explain how DMA improves data transfer efficiency in PCIe-based NVMe SSDs.

Solution: Direct Memory Access (DMA) allows the NVMe SSD to transfer data directly between its internal storage and the host system's RAM without involving the CPU for each byte. When a large file is read, the NVMe driver initiates a DMA transaction; the SSD

streams data into system memory at high speed via PCIe lanes. The CPU is free to perform computation (such as decrypting data) while the DMA engine handles the transfer. This parallelism maximizes throughput and minimizes CPU idle time. Modern NVMe SSDs use advanced DMA engines capable of scatter- gather operations, efficiently handling non-contiguous memory and supporting deep command queues for concurrent transfers [20].

Explanation: DMA offloads repetitive I/O tasks from the CPU, enabling high- bandwidth, low-latency storage operations essential for modern workloads.

VIII. SUMMARY AND FURTHER READING

This chapter has explored the foundational principles of computer organization and operating systems, emphasizing the interplay between hardware architecture and sys- tem software. We examined how the CPU, memory hierarchy, and I/O subsystems work in concert to deliver efficient computation, and how the operating system man- ages resources through process scheduling, memory management, file systems, and security mechanisms. Key concepts such as the von Neumann architecture, paging, DMA, and process scheduling algorithms were illustrated with diagrams and case studies. We also discussed the importance of modern protocols and standards, including the OSI model and NVMe for high-speed storage.

For readers seeking to deepen their understanding, several authoritative resources are recommended. "Computer Organization and Design" by Patterson and Hennessy is a comprehensive textbook that covers hardware/software interfaces, instruction set architectures, and the latest developments in RISC-V and cloud/mobile computing environments [21]. For operating systems, "Operating Systems: Three Easy Pieces" by Arpaci-Dusseau and Arpaci-Dusseau provides clear explanations of virtualization, concurrency, and persistence, with practical examples and historical context.

To stay current with research trends, recent IEEE and ACM articles on heterogeneous memory architectures offer insights into the challenges and opportunities of integrating multiple memory technologies within a single system [22]. These papers discuss data management strategies, performance trade-offs, and hardware/software co-design for emerging memory systems.

For structured learning, MIT’s 6.004 "Computation Structures" course covers dig- ital logic, computer architecture, and system design, while Stanford’s "Introduction to Operating Systems" lectures provide a deep dive into process management, synchronization, and file systems [23, 24]. These courses combine theory with hands-on exercises, making them ideal for both self-study and academic use.

By engaging with these books, articles, and courses, readers can build a robust foundation in computer systems and stay abreast of the latest developments in hardware and operating system design.

REFERENCES

[1] Guru, C.S.G.: Von Neumann Architecture. https://www.computerscience.gcse. guru/theory/von-neumann-architecture
[2] Wikipedia: Operating System. https://en.wikipedia.org/wiki/Operating_ system
[3] TutorialsPoint: Operating System Architecture. https://www.tutorialspoint.com/operating_system/os_ architecture.htm
[4] Rocket, S.:Relationship Between Hardware and Software. https://studyrocket.co.uk/revision/ a-level-computer-science-aqa/fundamentals-of-computer-systems/relationship-between-hardware-and-software
[5] Wikipedia: Timeline of Operating Systems. https://en.wikipedia.org/wiki/ Timeline_of_operating_systems
[6] BBC Bitesize: Von Neumann Architecture. https://www.bbc.co.uk/bitesize/ guides/zhppfcw/revision/3
[7] ScienceDirect: Von Neumann Architecture Overview. https://www.sciencedirect. com/topics/computer-science/von-neumann-architecture
[8] Business, University, T.: Memory Hierarchy. https://btu.edu.ge/wp-content/ uploads/2024/04/Lesson-3_-Memory-Hierarchy_-RAM-Cache-and-ROM.pdf
[9] Fiveable: Computer Performance Metrics. https://library.fiveable.me/lists/ computer-performance-metrics
[10] Unstop: Scheduling Algorithms in Operating System. https://unstop.com/blog/ scheduling-algorithms-in-operating-system
[11] Forum, E.S.: Paging Vs Segmentation: Core Differences Explained. https://www. enterprisestorageforum.com/hardware/paging-and-segmentation/
[12] TechLensFocus: Understanding Linux Filesystem and the Direc- tory Structure. https://techlensfocus.com/index.php/2024/01/12/understanding-linux-filesystem-and-the-directory-structure/
[13] Labs, W.K.: Windows Kernel Buffer Overflow. https://whiteknightlabs.com/ 2025/03/31/windows-kernel-buffer-overflow/
[14] Singh, N., Jain, N., Jain, S.: Ai and iot in digital payments: Enhancing security and efficiency with smart devices and intelligent fraud detection
[15] Online, S.: OSI Vs TCP/IP Model: What's the Difference? https://www.shiksha. com/online-courses/articles/osi-vs-tcp-ip-model-whats-the-difference/
[16] IBM: NVMe Vs. SATA: What's the Difference? https://www.ibm.com/think/ topics/nvme-vs-sata
[17] Juneja, I.: Understanding Kernel Memory Allocation. https://hackernoon.com/ understanding-kernel-memory-allocation
[18] Topics, S.: IPC in Linux. https://www.scaler.com/topics/ipc-in-linux/
[19] Vidyalay, G.: Effective Memory Access Time Calculation. https://www. gatevidyalay.com/paging-in-os-practice-problems-set-03/
[20] CTF.re: PCIe Part 2 - All About Memory: MMIO, DMA, TLPs, and More! https://ctf.re/kernel/pcie/tutorial/dma/mmio/tlp/2024/03/26/pcie-part-2/
[21] Patterson, D.A., Hennessy, J.L.: Computer Organization and Design RISC-V Edition: The Hardware Software Interface, 5th edn. Morgan Kaufmann, ??? (2021)
[22] Olson, M.B., Kammerdiener, B., Jantz, M.R., Doshi, K.A., Jones, T.: Online application guidance for heterogeneous memory systems. IEEE International Conference on Networking, Architecture, and Storage (2022)
[23] OpenCourseWare, M.: Computation Structures (6.004). https://ocw.mit.edu/ courses/6-004-computation-structures-spring-2009/
[24] University, S.: Operating Systems Lecture Notes. https://www.freebookcentre.net/ComputerScience-Books-Download/ Operating-Systems-Lecture-Notes-by-Stanford-University.html