Execution Management – Wrap-Up and Review

Week 12

SDB

Autumn 2025

Agenda: Execution Management - Review & Forward Path

1. **Recap: The Execution Management Journey**
2. **Module Interlink: The Big Picture**
3. **Connecting Theory to Practice: Lab Experience**
4. **Concept Integration Challenge**
5. **Advanced Exploration: Projects & Research**
6. **Open Discussion & Critical Thinking**
7. **Looking Ahead: Transition to Other OS Modules**
8. **Concluding Reflections**

Why This Review Matters

This module provided the bedrock for understanding how operating systems make applications run. Grasping execution management is crucial for debugging performance issues, designing efficient concurrent systems, and appreciating the intricate dance between hardware and software. It's the engine room of the OS.

Recap: The Execution Management Journey I

Part 1: Fundamentals & Process Lifecycle

- **Week 1: Introduction to Operating Systems**
  - Roles: Resource allocator, control program. Types: Batch, Time-sharing, Real-time, Distributed.
  - Kernel vs. User Mode: Privilege levels, protection. Abstraction: Hiding hardware complexity.
- **Week 2: System Calls & OS Structures**
  - System Calls: Interface between user-mode applications and kernel services (e.g., ‘fork()‘, ‘exec()‘, ‘open()‘, ‘read()‘).
  - OS Structures: Monolithic kernels (Linux), Layered approach, Microkernels (Minix, Mach), Hybrid systems (Windows).
- **Week 3: Processes & Process Control Block (PCB)**
  - Process: A program in execution. The fundamental unit of resource ownership.
  - Process States: New, Running, Waiting, Ready, Terminated.
  - PCB: The repository of all process-specific information (PID, state, registers, memory limits, open files, etc.).

Recap: The Execution Management Journey II

- **Week 4: Context Switching & Dispatcher**
  - Context Switch: Saving the state of the current process and loading the state of another. High overhead, pure overhead.
  - Dispatcher: The kernel module responsible for giving control of the CPU to the process selected by the scheduler. Dispatch latency is key.
- **Week 5: CPU Scheduling Algorithms (I)**
  - Metrics: Throughput, Turnaround Time, Waiting Time, Response Time, CPU Utilization.
  - Algorithms: First-Come, First-Served (FCFS), Shortest-Job-First (SJF) - (preemptive and non-preemptive), Round Robin (RR).
- **Week 6: CPU Scheduling Algorithms (II)**
  - Priority Scheduling: Potential for starvation, mitigated by aging.
  - Multilevel Queue/Feedback Queue Scheduling: Categorizing processes and using different algorithms for different queues.
- **Week 7: Real-Time Scheduling & OS Case Studies**
  - Real-Time: Hard vs. Soft RT. Rate Monotonic Scheduling (RMS), Earliest Deadline First (EDF).

Recap: The Execution Management Journey III

- Linux CFS: Completely Fair Scheduler for ‘SCHED_OTHER‘ tasks.
- Windows Priority Classes: Dynamic priority boosting.

- **Week 8: Threads & Multithreading Models**
  - Thread: Lightweight unit of execution within a process, sharing resources.
  - Models: One-to-One (Linux, Windows), Many-to-One, Many-to-Many.
  - User-level vs. Kernel-level threads: Performance vs. OS awareness.
- **Week 9: Semaphores & Mutual Exclusion**
  - Critical Section Problem: Ensuring atomic access to shared resources.
  - Semaphores: ‘wait()‘ (P/decrement) and ‘signal()‘ (V/increment) operations for synchronization. Binary vs. Counting.
- **Week 10: Classical Synchronization Problems**
  - Dining Philosophers: Deadlock and starvation with multiple resource acquisition.
  - Readers-Writers: Concurrent reads, exclusive writes; prioritizing one over other can cause starvation.

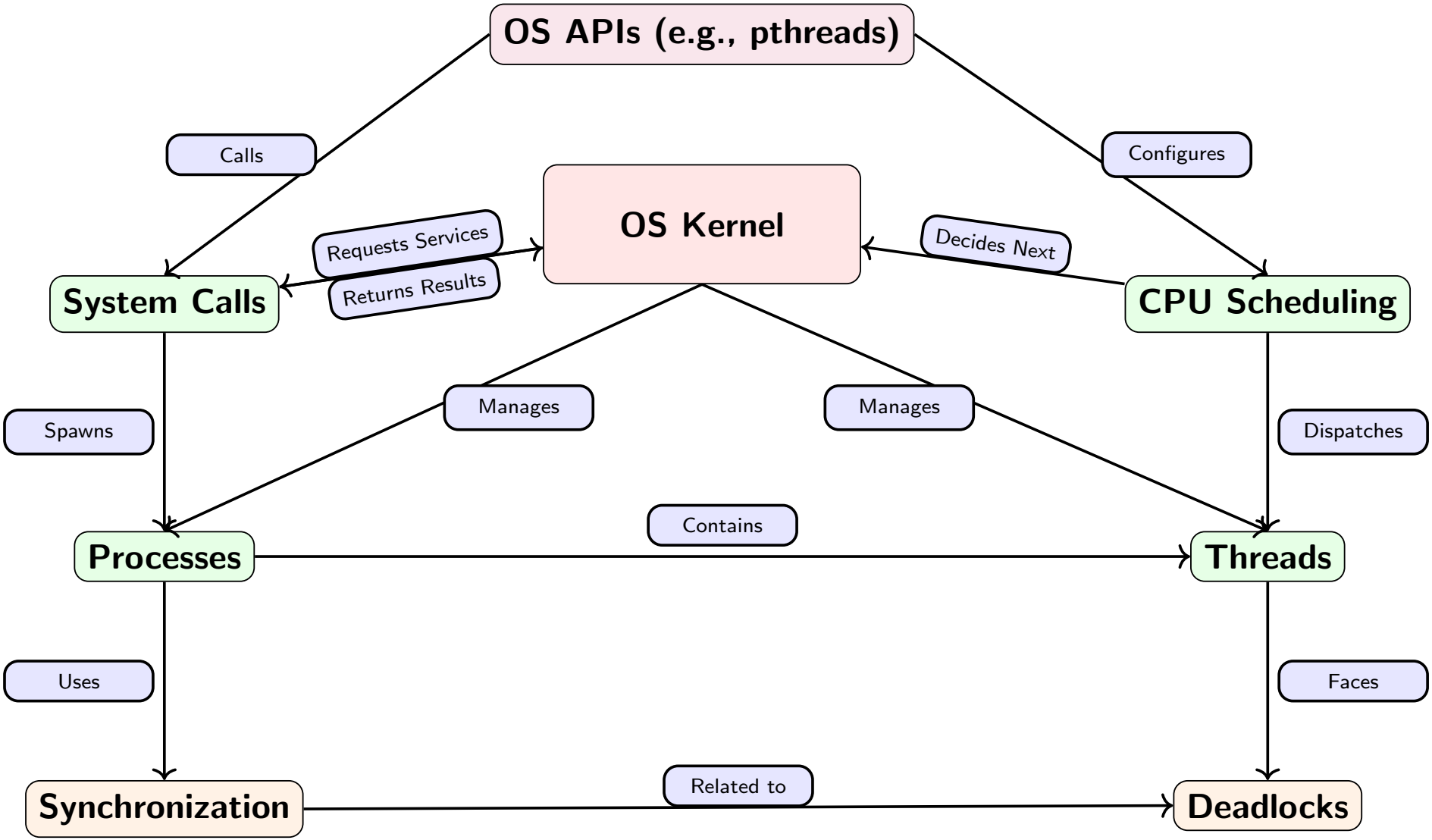
Recap: The Execution Management Journey IV

  - Bounded Buffer (Producer-Consumer): Using semaphores for coordination.

- **Week 11: Linux & Windows API Interfaces**
  - Linux: ‘fork()‘, ‘exec()‘, ‘wait()‘ for processes; ‘pthread‘ for threads, ‘clone()‘ for low-level control.
  - Windows: ‘CreateProcess()‘ for processes; ‘CreateThread()‘ for threads.
  - Control: Priority (‘nice‘, ‘SetThreadPriority‘), CPU Affinity (‘taskset‘, ‘SetThreadAffinityMask‘).
- **Week 12: Review & Integration**
  - Holistic view of execution flow.
  - Connecting theory to practical lab implementations.

Outcome: Students now understand and can manipulate the full lifecycle of execution in an OS—from process creation to scheduling and synchronization.

Module Interlink: The Execution Flow Ecosystem



This diagram illustrates the interconnected nature of the concepts covered, all orchestrated by the OS Kernel.

Connecting Theory to Practice: How Labs Mapped to Topics I

- **Lab 1 & 2: Process Creation & Execution**
  - *Concepts:* ‘fork()‘, ‘exec()‘, ‘wait()‘, process states, parent-child relationships, system calls.
  - *Outcome:* Gained hands-on experience with the Unix process model and built a rudimentary shell to understand process lifecycle control.
- **Lab 3: CPU Scheduling Simulation**
  - *Concepts:* FCFS, SJF, Round Robin algorithms, turnaround time, waiting time, context switching.
  - *Outcome:* Understood the mechanics of different scheduling policies and their performance implications through direct simulation.
- **Lab 4 & 5: Pthread Threading & Context Switching Analysis**
  - *Concepts:* Thread creation (‘pthread_create‘), joining (‘pthread_join‘), shared memory, thread overhead.

Connecting Theory to Practice: How Labs Mapped to Topics II

- *Outcome:* Experienced multithreading, observed context switching effects, and began to understand thread synchronization challenges.

- **Lab 6: Semaphores & Producer-Consumer Simulation**
  - *Concepts:* Mutual exclusion, critical sections, ‘wait()‘/‘signal()‘ operations, classical synchronization problem (Bounded Buffer).
  - *Outcome:* Implemented synchronization primitives to protect shared resources and coordinate concurrent processes/threads.

- **Lab 7+: Shell Miniproject & Dispatcher Log Tracing**
  - *Concepts:* Integration of processes, pipes (IPC), system call tracing, understanding dispatcher behavior from logs.
  - *Outcome:* Consolidated knowledge by building a more complex system, demonstrating how various OS features interoperate.

Concept Integration Challenge: Quick Quiz

1. **Round Robin Efficiency vs. Fairness:** Why is Round Robin considered more fair than FCFS, but sometimes less efficient in terms of overall throughput or average turnaround time?

2. **Many-to-Many Threading Tradeoffs:** What are the primary advantages and disadvantages of a Many-to-Many threading model compared to a One-to-One model?

3. **Starvation in Scheduling:** Define starvation in the context of CPU scheduling. Provide an example of a scheduling algorithm where it can occur, and describe a common mitigation technique.

4. **Windows vs. Unix Process Creation:** Which single Windows API call conceptually combines the functionality of Linux's 'fork()' and 'exec()' system calls? What additional control does it provide compared to the Unix pair?

5. **Synchronization Primitive Choice:** When would you definitively choose a **mutex** over a **binary semaphore** for protecting a critical section, and why?

Quick Quiz: Discussion Points I

Let's review the answers and common pitfalls:

1. **Round Robin Efficiency vs. Fairness:**
   - **Fairness:** RR provides better fairness and response time, especially for interactive tasks, by ensuring all processes get a slice of CPU time. No single long task can monopolize the CPU.
   - **Efficiency/Throughput:** Can be less efficient due to frequent context switches, especially with a very small time quantum. Each switch incurs overhead (saving/loading registers, flushing caches), reducing the actual time spent on useful work.

2. **Many-to-Many Threading Tradeoffs:**
   - **Advantages:** Combines benefits of both: multiple user-level threads can map to fewer (or equal) kernel threads. Allows concurrent execution on multicore systems while giving programmer control over user-level concurrency. Blocking one user-level thread doesn't block the entire process.

Quick Quiz: Discussion Points II

- **Disadvantages:** More complex to implement and manage (requires a sophisticated user-level thread library and kernel cooperation). Performance can be unpredictable due to two levels of scheduling.

3. **Starvation in Scheduling:**
   - **Definition:** A situation where a process is indefinitely postponed from gaining access to a resource (e.g., CPU) because other processes always take precedence.
   - **Example:** Pure Priority Scheduling (without aging). A low-priority process might never run if a continuous stream of high-priority processes keeps arriving.
   - **Mitigation: Aging** (gradually increasing the priority of processes that have been waiting for a long time).

4. **Windows vs. Unix Process Creation:**
   - **Windows API:** ‘CreateProcess()‘.

Quick Quiz: Discussion Points III

- **Additional Control:** ‘CreateProcess()‘ directly specifies the executable, command-line arguments, security attributes, environment block, current directory, and even the initial window appearance (‘STARTUPINFO‘). It offers much more granular control over the new process’s environment at creation time, whereas ‘fork()‘ duplicates most of the parent’s state and ‘exec()‘ then replaces it.

5. **Synchronization Primitive Choice:**
   - You would definitively choose a **mutex** over a binary semaphore for protecting a critical section when **ownership semantics** are important.
   - **Why:** Mutexes have the concept of an ”owner” (the thread that locked it). Only the owning thread can unlock a mutex. This prevents common errors like one thread accidentally releasing a lock held by another, or a thread forgetting to unlock. Semaphores lack this ownership; any thread can call ‘signal()‘ on a binary semaphore. Mutexes often also include features like recursion, priority inversion avoidance, and cleaner integration with condition variables.

Capstone Projects and Research Paths I

Your understanding of Execution Management opens doors to fascinating projects and research.

Hands-on Development Projects:

- **User-space Scheduler/Runtime:** Implement a simplified version of a CFS-like scheduler (e.g., for user-level threads or for managing a pool of tasks). This involves designing data structures (e.g., a balanced tree for run-queue), context switching (if managing user-level threads directly), and dispatching logic.
- **Kernel Call Tracer/Visualizer:** Develop a tool that hooks into system calls (e.g., using ‘ptrace‘ or ‘strace‘ output on Linux) and visualizes the sequence of calls made by a process or workload, identifying patterns or bottlenecks related to execution.
- **Simplified Thread Library (M:N Model):** Create a basic Many-to-Many thread management runtime. You’d manage user-level thread contexts and map them onto a smaller pool of kernel threads, implementing your own user-level scheduling logic.

Capstone Projects and Research Paths II

- **Mini Container Execution Manager:** Build a rudimentary container runtime that uses Linux ‘namespaces‘ (e.g., ‘CLONE_NEWPID‘, ‘CLONE_NEWNET‘, ‘CLONE_NEWNS‘) and ‘cgroups‘ (for resource limits like CPU and memory) to isolate and manage the execution of a new process.

Advanced Research and Exploration Topics:

- **Real-time Scheduler Simulation & Analysis:** Simulate and analyze advanced real-time scheduling algorithms (e.g., EDF with resource sharing protocols) to understand their predictability and performance under various loads.
- **Cgroup-based System Monitoring & Control:** Explore how ‘cgroups‘ are used in modern Linux systems to define resource limits (CPU shares, quotas, memory limits) and how you can programmatically interact with them to fine-tune application performance or resource isolation.

Capstone Projects and Research Paths III

- **Hardware-Assisted Virtualization for Execution:** Investigate how modern CPUs provide hardware support (e.g., Intel VT-x, AMD-V) for virtualization, enabling efficient execution of guest OSes and their processes.
- **Performance Impacts of OS Scheduling Parameters:** Conduct empirical studies to measure the impact of different scheduling policies, time quanta, and CPU affinity settings on real-world application performance (e.g., database servers, web servers).

Open Discussion Topics: OS Design & Evolution I

Let's engage in some critical thinking about operating system design principles based on our module.

- **Scheduling Strategy Redesign for Modern CPUs:** Given the prevalence of multi-core processors, non-uniform memory access (NUMA) architectures, and heterogeneous cores (e.g., ARM big.LITTLE), which scheduling strategy or existing scheduler component would you prioritize redesigning, and why? What new metrics or considerations would be crucial?

- **Threading Models & Multicore Performance:** How do the different threading models (1:1, M:1, M:N) inherently influence an application's ability to fully utilize multiple CPU cores? Are there scenarios where a user-level threading library could outperform kernel-level threads on a multi-core system, or vice-versa?

Open Discussion Topics: OS Design & Evolution II

- **Semaphores vs. Condition Variables:** Can semaphores be entirely eliminated in favor of condition variables (always used with a mutex) for general synchronization? What are the advantages and disadvantages of such a shift in programming paradigm? Consider simplicity, expressiveness, and error-proneness.

- **OS Design Choices & Energy Efficiency:** Beyond simply putting idle CPUs to sleep, what specific design choices within the OS's execution management (e.g., scheduling algorithms, context switching frequency, thread management) can significantly affect the overall energy consumption of a device (from a smartphone to a data center server)?

Brainstorm & Share

There are no single "right" answers here, but thoughtful discussion helps solidify your understanding of trade-offs.

Looking Ahead: Transition to Other OS Modules I

Execution Management is just one piece of the puzzle. It's the engine, but it needs fuel (memory) and a place to store its work (file system).

Connecting to Subsequent Modules:

- **Faculty B: Memory Management**
  - You've learned about process address spaces. Next, you'll delve into how the OS allocates and manages physical memory, virtual memory, paging, segmentation, and caching strategies. This is crucial for performance and protection.
- **Faculty C: File Systems & I/O Systems**
  - Processes need to store and retrieve data. You'll explore how file systems organize data on storage devices, manage access, and ensure integrity. I/O systems cover device drivers and efficient data transfer.
- **Faculty C (Continued): Advanced Synchronization & Deadlocks**

Looking Ahead: Transition to Other OS Modules II

- While we covered classical problems, the next module will provide a deeper dive into deadlock detection, avoidance, and recovery strategies, as well as more advanced synchronization primitives and concurrency models.

- **Faculty C (Continued): Security & Virtualization**
  - Execution management forms the basis of process isolation, which is fundamental to OS security. Virtualization extends this isolation to entire operating systems.

Stay curious about how execution intertwines with resource management and coordination layers. These modules build upon each other to form a complete understanding of operating systems.

Closing Reflections: The OS as the Foundation I

Your Journey Continues...

- **The OS is More Than Just Software:** It's a complex set of policies and mechanisms that enforce rules, protect resources, and arbitrate access to the underlying hardware. It's the ultimate resource manager.
- **Every Line of Code Matters:** From a simple "Hello World" to a complex web server, every instruction you write as a programmer eventually relies on the fundamental execution logic provided by the operating system. Understanding this helps you write better, more efficient, and more reliable code.
- **Foundational for Future Roles:** A solid grasp of execution management prepares you for:
  - **Kernel Development:** Contributing to operating systems themselves.
  - **Embedded Systems:** Optimizing code for resource-constrained environments.
  - **Distributed Systems:** Understanding how individual processes/threads interact across networks.

Closing Reflections: The OS as the Foundation II

- **Performance Engineering:** Identifying and solving bottlenecks in complex applications.
- **Security Analysis:** Recognizing vulnerabilities stemming from improper process isolation or synchronization.

"The operating system is the program that orchestrates the symphony of computation, ensuring every process gets its moment on stage, and every resource plays its part."

Thank you for your engagement throughout this module!

Outline