Process Concepts and Process Control Block

Week 3

SDB

Autumn 2025

Agenda: Understanding Process Execution

1. **What is a Process?** *(Defining the Core Unit)*
2. **Process States & Transitions** *(The Lifecycle of Execution)*
3. **The Process Control Block (PCB)** *(The OS's Ledger)*
4. **Context Switching & Queues** *(Enabling Multitasking)*
5. **Real-world Observations & Conclusion**

Agenda: Understanding Process Execution

1. **What is a Process?** *(Defining the Core Unit)*
2. **Process States & Transitions** *(The Lifecycle of Execution)*
3. **The Process Control Block (PCB)** *(The OS's Ledger)*
4. **Context Switching & Queues** *(Enabling Multitasking)*
5. **Real-world Observations & Conclusion**

Think Ahead: The Orchestrator's Challenge

How does an Operating System efficiently manage and keep track of hundreds, or even thousands, of concurrently running processes? What data structures and mechanisms are essential?

What is a Process? From Program to Execution

Definition: The Active Entity

A **process** is an instance of a program in execution. It's not just the code, but also its current state, associated data, and allocated system resources.

- **Program Code (Text Section):** The executable instructions.
- **Data Section:** Global and static variables initialized/uninitialized.
- **Heap:** Dynamically allocated memory during runtime.
- **Stack:** Temporary data (function parameters, return addresses, local variables).
- **CPU Registers:** Current state of the CPU for this process (e.g., Program Counter, stack pointer).
- **Process Metadata:** Information managed by the OS (Process ID (PID), state, priority).
- **Resource Handles:** Open files, network connections, I/O devices.

What is a Process? From Program to Execution

Definition: The Active Entity

A **process** is an instance of a program in execution. It's not just the code, but also its current state, associated data, and allocated system resources.

- **Program Code (Text Section):** The executable instructions.
- **Data Section:** Global and static variables initialized/uninitialized.
- **Heap:** Dynamically allocated memory during runtime.
- **Stack:** Temporary data (function parameters, return addresses, local variables).
- **CPU Registers:** Current state of the CPU for this process (e.g., Program Counter, stack pointer).
- **Process Metadata:** Information managed by the OS (Process ID (PID), state, priority).
- **Resource Handles:** Open files, network connections, I/O devices.

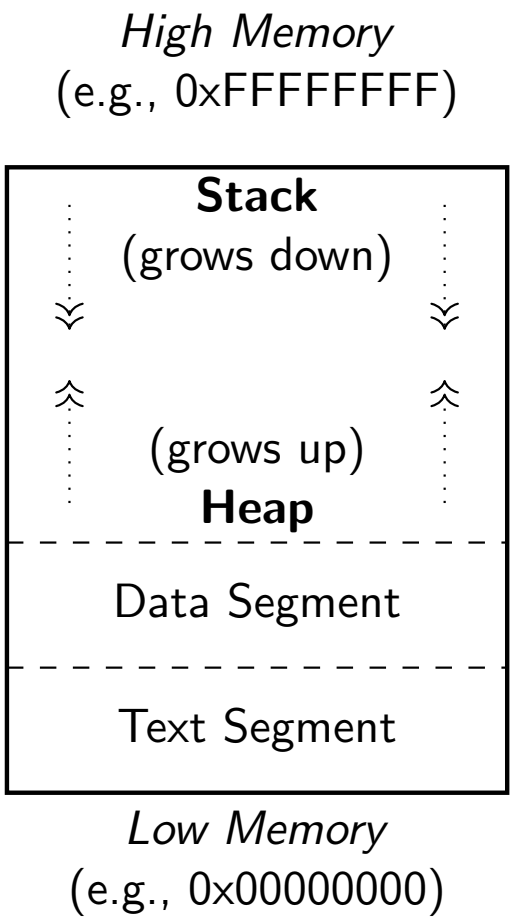
Analogy: The Dynamic Kitchen

Imagine a process as a chef actively cooking a recipe. The recipe itself is the **program code**. The kitchen's ingredients (static data), the current dish being prepared (heap), and the chef's current task list (stack) are all part of the process's memory. The chef's focus and tools (registers) represent the CPU state.

Inside a Process: Memory Layout

Typical Memory Segments of a Process:

- **Text Segment:** Contains compiled code (read-only)
- **Data Segment:** Global/static variables
  - **Initialized Data:** Variables with assigned values
  - **Uninitialized Data (BSS):** Variables declared but not initialized
- **Heap:** Dynamically allocated memory (e.g., malloc)
- **Stack:** Function call frames, local variables



Note: "Low memory" refers to the lower end of a process's virtual address space, starting near address 0. It typically contains the code and data segments. "High memory" is where the stack begins and grows downward.

Process States: The Lifecycle of Execution

The Dynamic Lifecycle of a Process:

- **New:** The process is being created. Resources are being allocated.
- **Ready:** The process is loaded into main memory and awaiting CPU allocation. It is runnable.
- **Running:** Instructions are being executed by the CPU. This is the active state.
- **Waiting (Blocked):** The process is waiting for some event to occur (e.g., I/O completion, a signal, resource availability). It cannot execute immediately.
- **Terminated:** The process has finished execution and is awaiting final cleanup by the OS.

Process States: The Lifecycle of Execution

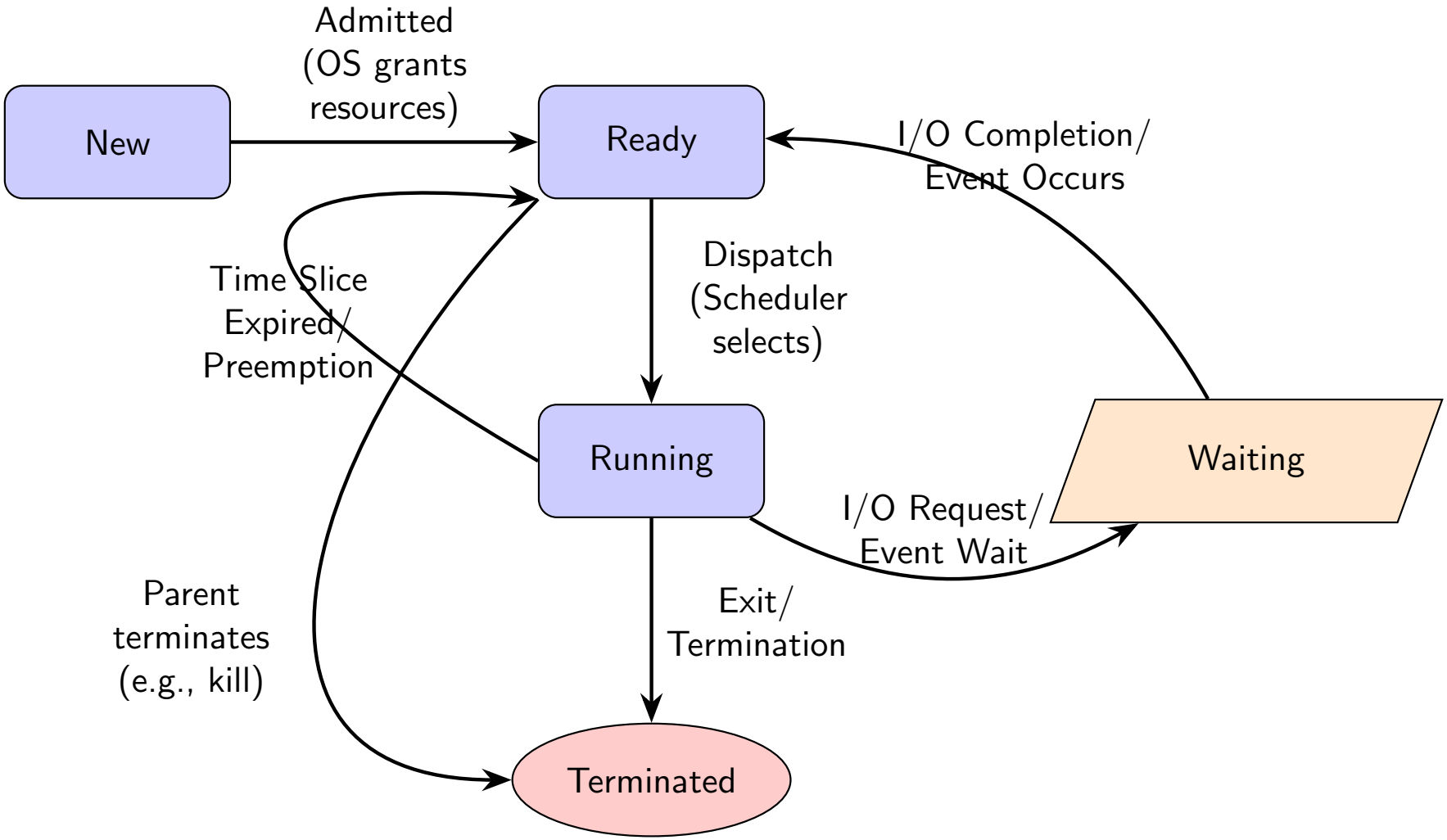
The Dynamic Lifecycle of a Process:

- **New:** The process is being created. Resources are being allocated.
- **Ready:** The process is loaded into main memory and awaiting CPU allocation. It is runnable.
- **Running:** Instructions are being executed by the CPU. This is the active state.
- **Waiting (Blocked):** The process is waiting for some event to occur (e.g., I/O completion, a signal, resource availability). It cannot execute immediately.
- **Terminated:** The process has finished execution and is awaiting final cleanup by the OS.

Discussion Point: State Transitions

Question: Can a process transition directly from the `Running` state back to the `Ready` state? If so, under what circumstances, and why is this critical for multitasking?

Process State Transitions: A Visual Flow



Each transition is a critical OS operation, triggered by specific events or scheduling decisions.

Special Process Types: Beyond the Basics I

Understanding how different process types behave is crucial for system management and debugging.

- **Zombie Process (Z or ‘⟨*defunct*⟩‘)**
  - A process that has **terminated** but whose **Process Control Block (PCB) still exists**.
  - Occurs because its parent has **not yet called ‘wait()‘** to collect its exit status.
  - Consumes minimal resources (mostly PID entry); too many can exhaust PID space.
- **Orphan Process**
  - A **running process** whose **parent process has terminated**.
  - **Adopted by the ‘init‘ process** (PID 1) or ‘systemd‘, which becomes its new parent and will eventually ‘wait()‘ for it.
  - Generally not a problem as ‘init‘ ensures proper cleanup.

Special Process Types: Beyond the Basics II

- **Daemon Process**
  - A **long-running background process** that operates independently of a controlling terminal.
  - Typically performs **system-wide services** (e.g., web server, logging).
  - Often initiated at boot and adopted by ‘init‘ or ‘systemd‘.
- **Foreground vs. Background Processes**
  - **Foreground:** Interacts directly with the terminal, blocking the shell until it completes or is suspended.
  - **Background:** Runs independently of the terminal (using ‘&‘), allowing the shell to accept new commands immediately.

For detailed explanations on each process type, refer to the Appendix.

The Process Control Block (PCB)

Definition: The Central Data Structure

The **Process Control Block (PCB)** is a highly critical data structure maintained by the Operating System for **every process**. It contains all the information needed to manage and control a specific process.

Key Information Stored in a PCB:

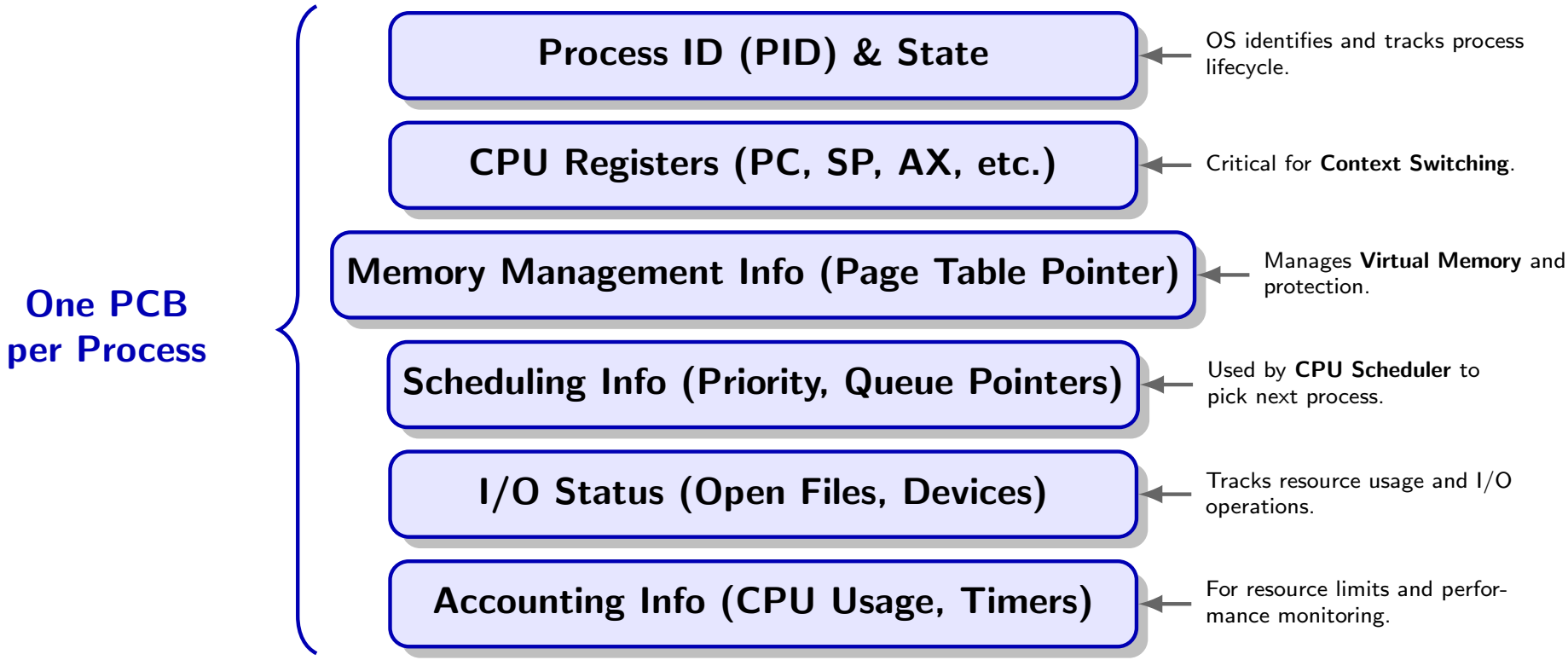
- **Process Identification:** Process ID (PID), Parent PID, User ID, Group ID.
- **Process State:** Current state (New, Ready, Running, Waiting, Terminated).
- **CPU State Information:** CPU registers, Program Counter (PC), Stack Pointer, General Purpose Registers.
- **Memory Management Information:** Pointers to page tables or segment tables, base and limit registers for memory areas.
- **Scheduling Information:** Process priority, pointers to scheduling queues, CPU time used, time limits.
- **I/O Status Information:** List of open files, list of allocated I/O devices, outstanding I/O requests.
- **Accounting Information:** CPU time used, real time used, time limits, process number.

The PCB acts as the Operating System's comprehensive "fingerprint" or "identity card" for each active process, enabling context switching and resource management.

PCB Layout Visualization

Conceptual Structure

The Process Control Block (PCB)



This aggregated data allows the OS to effectively manage, switch, and resume any process at any time.

Context Switching: The Heart of Multitasking

The Mechanism:

- **Phase 1: Save Current State.** The Operating System saves the complete CPU state (registers, program counter, stack pointer, etc.) of the currently running process into its corresponding PCB.
- **Phase 2: Load Next State.** The OS then loads the saved CPU state from the PCB of the process selected to run next.
- **Phase 3: Dispatch.** The CPU then begins execution from the loaded Program Counter of the new process.

Context Switching: The Heart of Multitasking

The Mechanism:

▶ **Phase 1: Save Current State.** The Operating System saves the complete CPU state (registers, program counter, stack pointer, etc.) of the currently running process into its corresponding PCB.

▶ **Phase 2: Load Next State.** The OS then loads the saved CPU state from the PCB of the process selected to run next.

▶ **Phase 3: Dispatch.** The CPU then begins execution from the loaded Program Counter of the new process.

Why it Matters (Costs & Benefits):

▶ **Enables Multitasking:** Allows multiple processes to appear to run concurrently on a single CPU, providing responsiveness and resource sharing.

▶ **Overhead:** This operation is pure overhead. The CPU performs no useful work for the user processes during a context switch. It consumes CPU cycles and memory bandwidth.

Context Switching: The Heart of Multitasking

The Mechanism:

▶ **Phase 1: Save Current State.** The Operating System saves the complete CPU state (registers, program counter, stack pointer, etc.) of the currently running process into its corresponding PCB.

▶ **Phase 2: Load Next State.** The OS then loads the saved CPU state from the PCB of the process selected to run next.

▶ **Phase 3: Dispatch.** The CPU then begins execution from the loaded Program Counter of the new process.

Why it Matters (Costs & Benefits):

▶ **Enables Multitasking:** Allows multiple processes to appear to run concurrently on a single CPU, providing responsiveness and resource sharing.

▶ **Overhead:** This operation is pure overhead. The CPU performs no useful work for the user processes during a context switch. It consumes CPU cycles and memory bandwidth.

OS Optimizations:

▶ Minimize context switch frequency (e.g., larger time slices).

▶ Optimize PCB access and state saving/loading routines.

▶ Hardware support (e.g., dedicated registers for fast context saving).

Process Queues: Organizing the Workflow

Types of Queues in OS Memory:

- **Job Queue (Long-Term Scheduler):** Contains all processes in the system, typically residing on disk. Processes from here are admitted to the ready queue.
- **Ready Queue (Short-Term Scheduler):** Contains processes that are in main memory and are ready and waiting for the CPU. Processes from here are dispatched to the CPU.
- **Device Queues (I/O Queues):** Contains processes waiting for a specific I/O device (e.g., disk, printer, network card). There is typically one queue per device.

Process Queues: Organizing the Workflow

Types of Queues in OS Memory:

■ **Job Queue (Long-Term Scheduler):** Contains all processes in the system, typically residing on disk. Processes from here are admitted to the ready queue.

■ **Ready Queue (Short-Term Scheduler):** Contains processes that are in main memory and are ready and waiting for the CPU. Processes from here are dispatched to the CPU.

■ **Device Queues (I/O Queues):** Contains processes waiting for a specific I/O device (e.g., disk, printer, network card). There is typically one queue per device.

Key Queue Transitions:

- **New Process** → **Job Queue** → (Admitted) → **Ready Queue**
- **Running Process** (Time Slice Expired) → **Ready Queue**
- **Running Process** (I/O Request) → **Device Queue**
- **Device Queue** (I/O Complete) → **Ready Queue**
- **Any Queue** → **Terminated State** (e.g., process completion, killed)

These queues are fundamental for the OS's scheduling policies, ensuring fair and efficient resource allocation.

Practical Session: Observing Processes in Linux

Hands-on: Exploring Process Information

- Connect to a Linux environment (VM, WSL, or native).
- Open a terminal and execute the following commands:

```
# View a snapshot of your
    running processes
ps aux | head -n 10
# Observe real-time process
    activity, CPU, and memory
top
# Examine the status of your
    current shell process
cat /proc/$$/status
```

What to Focus On During Your Observation:

- **Process State (STAT/S column):** Identify processes in 'R' (Running), 'S' (Sleeping/Waiting), 'Z' (Zombie), 'T' (Stopped) states.
- **Process ID (PID) & Parent PID (PPID):** Understand process hierarchy.
- **CPU Usage (%CPU) and Memory Usage (%MEM):** Analyze resource consumption.
- **CPU Time (TIME+):** Total CPU time consumed.

Practical Session: Observing Processes in Linux

Hands-on: Exploring Process Information

- Connect to a Linux environment (VM, WSL, or native).
- Open a terminal and execute the following commands:

```
# View a snapshot of your
    running processes
ps aux | head -n 10
# Observe real-time process
    activity, CPU, and memory
top
# Examine the status of your
    current shell process
cat /proc/$$/status
```

What to Focus On During Your Observation:

- **Process State (STAT/S column):** Identify processes in 'R' (Running), 'S' (Sleeping/Waiting), 'Z' (Zombie), 'T' (Stopped) states.
- **Process ID (PID) & Parent PID (PPID):** Understand process hierarchy.
- **CPU Usage (%CPU) and Memory Usage (%MEM):** Analyze resource consumption.
- **CPU Time (TIME+):** Total CPU time consumed.

Challenge Question

Can you find the PID of your web browser and then locate its process status file in '/proc'? What information does it contain?

Key Takeaways

- A **process** is the fundamental unit of execution in an OS, representing a program with its state and resources.
- The **Process Control Block (PCB)** is the OS's vital data structure, holding all metadata required for process management.
- Processes cycle through distinct **states** (`New`, `Ready`, `Running`, `Waiting`, `Terminated`), driven by events and the OS scheduler.
- **Context switching** is the mechanism that enables multitasking, allowing the CPU to rapidly switch between processes, despite its inherent overhead.
- The OS uses various **queues** (Job, Ready, Device) to organize and manage processes for efficient scheduling and resource allocation.

Key Takeaways

- A **process** is the fundamental unit of execution in an OS, representing a program with its state and resources.
- The **Process Control Block (PCB)** is the OS's vital data structure, holding all metadata required for process management.
- Processes cycle through distinct **states** (`New`, `Ready`, `Running`, `Waiting`, `Terminated`), driven by events and the OS scheduler.
- **Context switching** is the mechanism that enables multitasking, allowing the CPU to rapidly switch between processes, despite its inherent overhead.
- The OS uses various **queues** (Job, Ready, Device) to organize and manage processes for efficient scheduling and resource allocation.

Future Reflection Prompt

Given your understanding of processes and PCBs, how might this knowledge be directly applicable when you are debugging a multi-threaded application or optimizing a system's performance? Consider issues like race conditions or resource contention.

Next Week Preview: Deeper Dive into CPU Scheduling

- **The Dispatcher's Role:** How the OS hands off the CPU to the selected process.
- **Scheduling Metrics:** Understanding performance indicators like Turnaround Time, Waiting Time, and Response Time.
- **Scheduling Goals:** Balancing fairness, efficiency, throughput, and responsiveness in CPU allocation.
- **Common Scheduling Algorithms:**
  - First-Come, First-Served (FCFS)
  - Shortest Job First (SJF) and Shortest Remaining Time First (SRTF)
  - Round Robin
  - Priority-based Scheduling
  - Multilevel Queue and Multilevel Feedback Queue Scheduling
- **Practical Application:** Hands-on simulation or analysis of scheduler behavior.

Next Week Preview: Deeper Dive into CPU Scheduling

- **The Dispatcher's Role:** How the OS hands off the CPU to the selected process.
- **Scheduling Metrics:** Understanding performance indicators like Turnaround Time, Waiting Time, and Response Time.
- **Scheduling Goals:** Balancing fairness, efficiency, throughput, and responsiveness in CPU allocation.
- **Common Scheduling Algorithms:**
  - First-Come, First-Served (FCFS)
  - Shortest Job First (SJF) and Shortest Remaining Time First (SRTF)
  - Round Robin
  - Priority-based Scheduling
  - Multilevel Queue and Multilevel Feedback Queue Scheduling
- **Practical Application:** Hands-on simulation or analysis of scheduler behavior.

Preparation Tip for Next Session

Experiment with simple programs in Linux: use the `time ./a.out` command to measure execution time. Think about how other running processes might affect these measurements.

Outline

Quick Quiz: Test Your Understanding

Self-Assessment Questions:

1. During a context switch, what specific information is saved from the current process, and where is it stored?
2. Describe the sequence of events that allows a process to move from a `Waiting` state back to the `Ready` state.
3. Explain why a process cannot simultaneously be in the `Running` and `Waiting` states. What fundamental principle does this illustrate?

Quick Quiz: Test Your Understanding

Self-Assessment Questions:

1. During a context switch, what specific information is saved from the current process, and where is it stored?
2. Describe the sequence of events that allows a process to move from a `Waiting` state back to the `Ready` state.
3. Explain why a process cannot simultaneously be in the `Running` and `Waiting` states. What fundamental principle does this illustrate?

Hint for Discussion

Refer back to the **Process Control Block (PCB)** and **Process State Transition Diagram** slides. Consider the role of events like I/O completion and the CPU's single execution unit.

Quick Check: Process Fundamentals

Test Your Basic Understanding:

1. **True/False:** A program is the same as a process. Explain your answer briefly.

2. **Multiple Choice:** Which of the following is NOT typically stored in a Process Control Block (PCB)?
   - (a) Process ID (PID)
   - (b) CPU Registers
   - (c) Source Code
   - (d) Memory Management Information

3. **Fill in the Blank:** When a process makes an I/O request, it typically transitions from the _____ state to the _____ state.

4. **Short Answer:** What is the primary purpose of context switching in an operating system?

Quick Check: Process Fundamentals

Test Your Basic Understanding:

1. **True/False:** A program is the same as a process. Explain your answer briefly.

2. **Multiple Choice:** Which of the following is NOT typically stored in a Process Control Block (PCB)?
   - (a) Process ID (PID)
   - (b) CPU Registers
   - (c) Source Code
   - (d) Memory Management Information

3. **Fill in the Blank:** When a process makes an I/O request, it typically transitions from the _____ state to the _____ state.

4. **Short Answer:** What is the primary purpose of context switching in an operating system?

Think & Discuss

Take a moment to formulate your answers. We'll discuss these briefly.

Advanced Conceptual Quiz: Deeper Dive

Apply Your Knowledge:

1. **Scenario Analysis:** A process is currently in the `Ready` state. Describe two distinct events or OS decisions that could cause it to transition into the `Terminated` state without ever entering the `Running` state.

2. **Critical Thinking:** Context switching introduces overhead. If an OS designers wants to minimize this overhead, what specific PCB fields would they focus on optimizing for faster saving and loading, and why?

3. **Analogy Extension:** Revisit our "chef executing a recipe" analogy. Where in this analogy would you place the "Ready Queue" and the "Device Queue"? Explain your reasoning.

4. **Problem Solving:** Imagine you are tasked with designing a very simple OS. You have limited memory for PCBs. Which PCB field(s) would you consider absolutely essential for basic multitasking, and which could potentially be omitted or simplified for a bare-bones system? Justify your choices.

Advanced Conceptual Quiz: Deeper Dive

Apply Your Knowledge:

1. **Scenario Analysis:** A process is currently in the `Ready` state. Describe two distinct events or OS decisions that could cause it to transition into the `Terminated` state without ever entering the `Running` state.

2. **Critical Thinking:** Context switching introduces overhead. If an OS designers wants to minimize this overhead, what specific PCB fields would they focus on optimizing for faster saving and loading, and why?

3. **Analogy Extension:** Revisit our "chef executing a recipe" analogy. Where in this analogy would you place the "Ready Queue" and the "Device Queue"? Explain your reasoning.

4. **Problem Solving:** Imagine you are tasked with designing a very simple OS. You have limited memory for PCBs. Which PCB field(s) would you consider absolutely essential for basic multitasking, and which could potentially be omitted or simplified for a bare-bones system? Justify your choices.

Collaborate & Challenge

Discuss these challenges with a peer. There might be more than one valid approach!

Exercise: Process State & PCB Mapping

Instructions: For each scenario below, identify the likely process state(s) and list which specific PCB fields would be most actively updated or referenced by the OS.

1. **Scenario A: User clicks "Save" in a word processor.**
   - Process State: _____
   - Key PCB fields updated/referenced: _____
2. **Scenario B: The OS timer interrupt fires, signaling end of time slice.**
   - Process State: _____
   - Key PCB fields updated/referenced: _____
3. **Scenario C: A new application is launched from the desktop.**
   - Process State: _____
   - Key PCB fields updated/referenced: _____
4. **Scenario D: A background compilation task completes successfully.**
   - Process State: _____
   - Key PCB fields updated/referenced: _____

Exercise: Process State & PCB Mapping

Instructions: For each scenario below, identify the likely process state(s) and list which specific PCB fields would be most actively updated or referenced by the OS.

1. **Scenario A: User clicks "Save" in a word processor.**
   - Process State: _____
   - Key PCB fields updated/referenced: _____
2. **Scenario B: The OS timer interrupt fires, signaling end of time slice.**
   - Process State: _____
   - Key PCB fields updated/referenced: _____
3. **Scenario C: A new application is launched from the desktop.**
   - Process State: _____
   - Key PCB fields updated/referenced: _____
4. **Scenario D: A background compilation task completes successfully.**
   - Process State: _____
   - Key PCB fields updated/referenced: _____

Consider the OS's Role

Think about what the OS *must* know or change about the process in each event.

Exercise: Context Switching & Performance Implications

Instructions: Answer the following questions, showing your reasoning.

1. **Quantitative Analysis:** Assume a context switch takes $10\,\mu s$ (microseconds). If a CPU has a time slice of $10\,ms$ (milliseconds), what percentage of the CPU's time is spent on context switching if processes always use their full time slice?
   - Calculation: _____
   - Percentage Overhead: _____
2. **Trade-off Discussion:** Based on your calculation above, discuss the trade-offs of having a very short time slice versus a very long time slice in a multitasking OS. Consider responsiveness, throughput, and overhead.
3. **Challenge: "Zombie" Processes:**
   - What is a "zombie" process?
   - Why does it exist (what problem does it solve for the parent)?
   - Why is it generally harmless, but why can a large number of them be problematic?

Exercise: Context Switching & Performance Implications

Instructions: Answer the following questions, showing your reasoning.

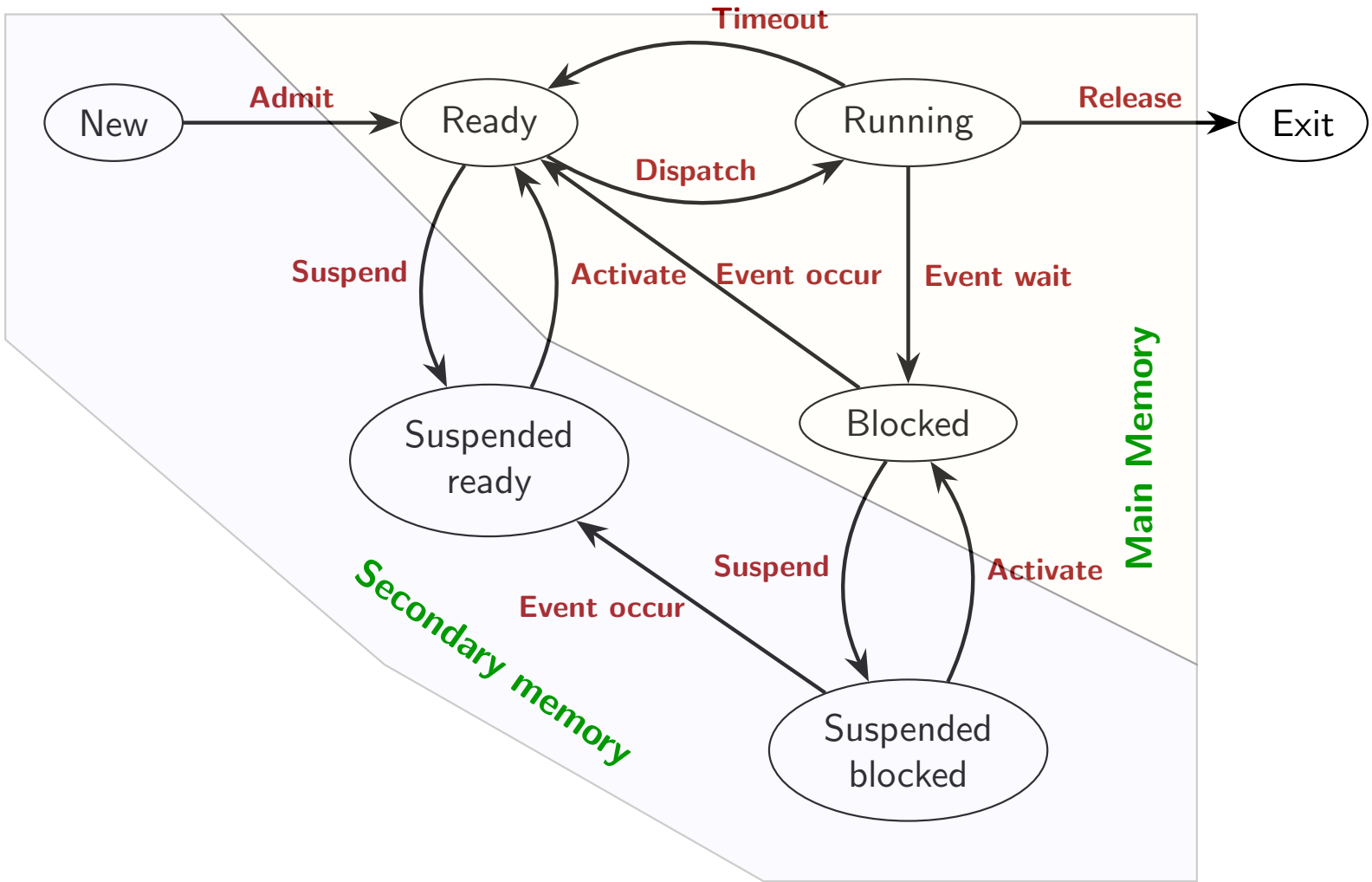
1. **Quantitative Analysis:** Assume a context switch takes $10\,\mu s$ (microseconds). If a CPU has a time slice of $10\,ms$ (milliseconds), what percentage of the CPU's time is spent on context switching if processes always use their full time slice?
   - Calculation: _____
   - Percentage Overhead: _____
2. **Trade-off Discussion:** Based on your calculation above, discuss the trade-offs of having a very short time slice versus a very long time slice in a multitasking OS. Consider responsiveness, throughput, and overhead.
3. **Challenge: "Zombie" Processes:**
   - What is a "zombie" process?
   - Why does it exist (what problem does it solve for the parent)?
   - Why is it generally harmless, but why can a large number of them be problematic?

Hint: Time Conversions!

Remember to convert all time units to be consistent (e.g., all to microseconds or all to milliseconds).

Process State Transitions

Detailed



Special Process Types: Zombie Processes (Z) I

Definition: The Deceased Process

A **Zombie Process** (also known as a "defunct" process) is a process that has completed its execution (terminated), but its entry in the process table (its PCB) still exists because its parent process has not yet called ‘wait()‘ or ‘waitpid()‘ to collect its exit status.

How They Are Created:

- A child process finishes execution (calls ‘exit()‘).
- The OS cleans up most of its resources (memory, open files, CPU state).
- However, the PCB is retained to allow the parent process to retrieve the child's exit status.
- If the parent never ‘wait()‘s, the PCB remains.

Why Are They a Concern?

Special Process Types: Zombie Processes (Z) II

- They consume minimal system resources (mainly just a process ID and a PCB entry).
- A few zombies are harmless.
- **Too many zombies:** Can exhaust the system's process ID (PID) space, preventing new processes from being created. This is a rare but serious issue, typically indicating a bug in a long-running parent process.

How to "Reap" a Zombie Process:

- The parent process must call ‘wait()‘ or ‘waitpid()‘.
- If the parent terminates, all its zombie children are inherited by the ‘init‘ process (PID 1), which automatically ‘wait()‘s for its adopted children, cleaning them up.

You might see them listed as ‘Z‘ or ‘⟨defunct⟩ >‘ in ‘ps‘ output.

Special Process Types: Orphan Processes (O) I

Definition: The Parentless Process

An **Orphan Process** is a running process whose parent process has terminated before the child process.

How They Are Created:

- A parent process creates a child process.
- The parent process then terminates (e.g., crashes, completes its task) while the child process is still running.

How the OS Handles Orphans (The ‘init‘ Process):

- Unlike zombies, orphan processes are fully functional and continue to run.
- The operating system (specifically the ‘init‘ process, which always has PID 1, or ‘systemd‘ in modern Linux) automatically **adopts** orphan processes.
- The ‘init‘ process becomes the new parent of the orphaned child.

Special Process Types: Orphan Processes (O) II

- This is a crucial mechanism for system stability, ensuring no running process is left without a parent to manage it.

Why They Are Not a Problem (Generally):

- Orphans are actively managed by ‘init‘/‘systemd‘.
- ‘init‘ will eventually ‘wait()‘ for these adopted children when they finally terminate, preventing them from becoming zombies.
- This mechanism ensures proper resource cleanup.

You might see their PPID change to 1 in ‘ps‘ output once orphaned.

Special Process Types: Daemon Processes I

Definition: The System's Background Workers

A **Daemon Process** (often simply called a "daemon") is a computer program that runs as a background process, rather than being under the direct control of an interactive user. Daemons are typically initiated at boot time and live for the entire duration of the system's uptime.

Characteristics:

- **Background Execution:** They don't have a controlling terminal and typically don't interact directly with users.
- **System Services:** They perform system-wide functions, such as handling network requests (web servers, SSH daemons), logging, printing, or scheduling tasks.
- **Long-Lived:** Designed to run continuously, waiting for events or performing periodic tasks.

Special Process Types: Daemon Processes II

- **Parent is ‘init‘ (often):** Once daemonized, their original parent often exits, and they are adopted by the ‘init‘ or ‘systemd‘ process (PID 1).

Examples:

- ‘httpd‘ (Apache web server daemon)
- ‘sshd‘ (SSH daemon for remote login)
- ‘syslogd‘ (system logging daemon)
- ‘crond‘ (cron daemon for scheduled tasks)

In Linux, daemonization typically involves forking, detaching from the controlling terminal, and often adopting PID 1.

Foreground vs. Background Processes I

Definition: Interactive Control

This classification describes how a process interacts with the user and the controlling terminal (e.g., a shell).

1. **Foreground Process:**
   - **Direct Interaction:** The process that currently has control of the terminal. It can receive input from the keyboard and send output directly to the screen.
   - **Blocking:** The shell (or terminal) waits for the foreground process to complete or be suspended before accepting new commands.
   - **Creation:** Typically created when you type a command and press Enter in a shell (e.g., ‘ls -l‘, ‘vim‘).

Foreground vs. Background Processes II

2. Background Process:

- **No Direct Control:** The process runs independently of the terminal, meaning the shell does not wait for its completion and immediately returns control to the user.
- **Detached from Input:** It generally cannot receive keyboard input directly from the terminal. Its output might still go to the terminal or be redirected.
- **Creation:** Created by appending an ampersand (‘&‘) to a command in the shell (e.g., ‘firefox &‘, ‘sleep 300 &‘). Can also be moved to background from foreground using ‘Ctrl+Z‘ (suspend) then ‘bg‘.

Practical Implications:

- Essential for multitasking in a command-line environment.
- Understanding their behavior is key for managing long-running tasks or services without tying up your terminal.
- ‘jobs‘ command lists background processes, ‘fg‘ brings them to foreground, ‘bg‘ resumes them in background.

These concepts are closely related to Job Control in Unix-like operating systems.

Explore Further: Advanced Process & Kernel Topics

Beyond the Basics – What Else is in a PCB?

- **Parent/Child Links:** Process hierarchy (PPID, child list)
- **Signal Handlers:** Registered actions for signals (e.g., SIGINT)
- **Resource Descriptors:** File descriptors, memory maps
- **Scheduling Info:** Priority, time slice, CPU affinity
- **Security Context:** User ID, group ID, capabilities

Explore Further: Advanced Process & Kernel Topics

Beyond the Basics – What Else is in a PCB?

- **Parent/Child Links:** Process hierarchy (PPID, child list)
- **Signal Handlers:** Registered actions for signals (e.g., SIGINT)
- **Resource Descriptors:** File descriptors, memory maps
- **Scheduling Info:** Priority, time slice, CPU affinity
- **Security Context:** User ID, group ID, capabilities

Context Switching – What Happens?

- Save current process state (registers, PC, stack pointer) into PCB
- Choose next process via scheduler
- Load next process state from its PCB
- Update memory mappings (page tables, TLB flush if needed)

Explore Further: Advanced Process & Kernel Topics

Beyond the Basics – What Else is in a PCB?

- **Parent/Child Links:** Process hierarchy (PPID, child list)
- **Signal Handlers:** Registered actions for signals (e.g., SIGINT)
- **Resource Descriptors:** File descriptors, memory maps
- **Scheduling Info:** Priority, time slice, CPU affinity
- **Security Context:** User ID, group ID, capabilities

Context Switching – What Happens?

- Save current process state (registers, PC, stack pointer) into PCB
- Choose next process via scheduler
- Load next process state from its PCB
- Update memory mappings (page tables, TLB flush if needed)

Explore More

Try exploring: - How Linux implements ‘task_struct‘ - How context switches are optimized in multicore systems - How real-time OSes reduce context switch latency

Linux's ‘/proc‘ Filesystem - The Kernel Interface I

What is ‘/proc‘ (Procfs)?

- A **virtual filesystem** (sometimes called a "pseudo-filesystem").
- It's **not stored on disk**; it's generated on-the-fly by the Linux kernel when you access it.
- Provides a powerful interface to **kernel data structures** and real-time system information.
- Allows observation and, in some cases, modification of kernel parameters and process states.

Accessing Process Information: ‘cat /proc/$$/status‘

- **‘$$‘**: A special shell variable that expands to the **Process ID (PID)** of the current shell.
- **‘/proc/PID/‘**: For every running process with PID ‘X‘, there's a directory ‘/proc/X/‘.
- **‘/proc/PID/status‘**: A human-readable summary of a process's status, resembling parts of its Process Control Block (PCB).

Linux's '/proc' Filesystem - The Kernel Interface II

Example Output ('cat /proc/$$/status' for a 'bash' shell):

```
Name:   bash              # Process name
State:  S (sleeping)      # Current process state
Pid:    1234              # Process ID
PPid:   1233              # Parent Process ID
VmSize: 15000 kB          # Current virtual memory size
VmRSS:  7000 kB           # Current resident set size (physical memory)
Threads:        1         # Number of threads in the process
Cpus_allowed_list:      0-3 # CPUs the process is allowed to run on
voluntary_ctxt_switches:        1000 # Number of voluntary context switches
nonvoluntary_ctxt_switches:     50   # Number of non-voluntary context switches
```

(Output truncated for brevity)

Deconstructing ‘/proc/PID/status‘ (Key Fields) I

The fields in ‘/proc/PID/status‘ provide direct insight into OS concepts:

1. **Process Identification & State:**
   - **‘Name‘**: The executable name (often truncated).
   - **‘State‘**: Current process state (e.g., ‘R‘ for Running, ‘S‘ for Sleeping, ‘Z‘ for Zombie, ‘T‘ for Stopped). Direct mapping to process states from Week 3.
   - **‘Pid‘**: The process's unique ID.
   - **‘PPid‘**: The Parent Process ID. Crucial for understanding process hierarchies and orphan processes.
   - **‘Tgid‘**: Thread Group ID. For a multi-threaded process, all threads share the same ‘Tgid‘, which is the PID of the main thread.
   - **‘Threads‘**: Number of threads within this process. Directly relates to multithreading concepts from Week 8.

Deconstructing ‘/proc/PID/status‘ (Key Fields) II

2. Resource Usage (Memory):

- **‘VmPeak‘**: Peak virtual memory size.
- **‘VmSize‘**: Current virtual memory size.
- **‘VmHWM‘**: Peak resident set size (highest physical memory usage).
- **‘VmRSS‘**: Current resident set size (actual physical memory used by the process, excluding swapped-out portions).
- **‘VmData‘**, **‘VmStk‘**, **‘VmExe‘**, **‘VmLib‘**: Sizes of data, stack, executable code, and shared library segments.
- These fields offer a glimpse into the process's memory layout and usage, a core part of its PCB's memory management information (linking to future Memory Management topics).

Deconstructing ‘/proc/PID/status‘ (Key Fields) III

3. Scheduling & Control:

- **‘Cpus_allowed‘ / ‘Cpus_allowed_list‘**: A bitmask/list indicating which CPU cores the process is allowed to run on. Directly related to CPU affinity control (Week 11).
- **‘Mems_allowed‘ / ‘Mems_allowed_list‘**: Similar to ‘Cpus_allowed‘, but for NUMA (Non-Uniform Memory Access) nodes, indicating allowed memory nodes.
- **‘voluntary_ctxt_switches‘**: Number of times the process explicitly yielded the CPU (e.g., waiting for I/O, ‘sleep()‘).
- **‘nonvoluntary_ctxt_switches‘**: Number of times the scheduler preempted the process (e.g., time slice expired, higher priority task became ready). These provide insights into scheduler behavior and efficiency (Week 4, 5, 6).

Deconstructing ‘/proc/PID/status‘ (Key Fields) IV

4. Privileges & Signals:

- **‘Uid‘ / ‘Gid‘**: User ID and Group ID (real, effective, saved, filesystem). Essential for understanding process permissions and security (Week 1/2 concepts).
- **‘Sig*‘**: Various fields related to signal handling (pending signals, blocked signals, ignored signals, caught signals). The PCB stores this signal management information.
- **‘Cap*‘**: Capabilities (e.g., ‘CapEff‘ for effective capabilities). Linux capabilities allow breaking down root's super-privileges into smaller, distinct units.

‘/proc/PID/status‘ is like a simplified, real-time snapshot of the process's PCB and its interaction with the kernel's resource managers.

Navigating the ‘/proc‘ Filesystem (Other Key Areas) I

Beyond ‘status‘, the ‘/proc‘ filesystem offers a wealth of information about processes and the system.

Per-Process Information (‘/proc/PID/‘):

- **‘cmdline‘**: The full command-line arguments used to start the process.
- **‘environ‘**: The environment variables of the process.
- **‘cwd‘**: A symbolic link to the process’s current working directory.
- **‘exe‘**: A symbolic link to the executable file of the process.
- **‘fd/‘**: A directory containing symbolic links for all open file descriptors of the process. (E.g., ‘ls -l /proc/$$/fd/‘ to see your shell’s open files).
- **‘maps‘**: A list of memory regions (mappings) in the process’s virtual address space, including shared libraries, heap, and stack. (Important for Memory Management - future topic).
- **‘task/‘**: A subdirectory containing information about each thread within a multi-threaded process.

Navigating the ‘/proc‘ Filesystem (Other Key Areas) II

System-Wide Information (‘/proc/‘ direct files):

- **‘cpuinfo‘**: Detailed information about the CPU(s).
- **‘meminfo‘**: Information about system memory usage (total, free, buffers, cache).
- **‘loadavg‘**: System load average (average number of processes in the run queue or running) over 1, 5, and 15 minutes.
- **‘uptime‘**: System uptime and idle time.
- **‘version‘**: Linux kernel version string.
- **‘filesystems‘**: List of filesystems supported by the kernel.
- **‘net/‘**: Directory containing network stack information.

Navigating the ‘/proc‘ Filesystem (Other Key Areas) III

Kernel Parameters (‘/proc/sys/‘):

- A hierarchical directory to view and **modify kernel parameters at runtime**.
- E.g., ‘cat /proc/sys/vm/swappiness‘ to see swap aggressiveness.
- E.g., ‘echo 1 > /proc/sys/net/ipv4/ip_forward‘ to enable IP forwarding (requires root).

‘/proc‘ is the backbone of many Linux system monitoring and debugging tools.

Practical Exploration - Things to Try Out I

1. **Explore Your Own Shell (‘$$‘):**
   - ‘cat /proc/$$/status‘ (your shell’s status)
   - ‘cat /proc/$$/cmdline‘ (how your shell was started)
   - ‘ls -l /proc/$$/fd/‘ (your shell’s open files, including stdin/out/err)
   - ‘ls -l /proc/$$/exe‘ (the actual executable of your shell)
   - ‘cat /proc/$$/maps‘ (your shell’s memory map)
2. **Observe Other Processes:**
   - Open another terminal and run ‘sleep 600‘.
   - In your original terminal, find its PID: ‘pgrep sleep‘
   - Now, use that PID to investigate: ‘cat /proc/¡sleep_PID¿/status‘, ‘cat /proc/¡sleep_PID¿/cmdline‘
   - Observe its ‘State: S (sleeping)‘.
   - Run ‘top‘ or ‘htop‘ and see if you can match the ‘PID‘ and ‘Name‘ from ‘/proc‘.

Practical Exploration - Things to Try Out II

3. **System-Wide Information:**
   - ‘cat /proc/cpuinfo‘ (CPU details)
   - ‘cat /proc/meminfo‘ (Memory usage)
   - ‘cat /proc/loadavg‘ (System load)
   - ‘cat /proc/uptime‘ (System uptime)
4. **Kernel Parameters (be cautious with ‘sys/‘):**
   - ‘cat /proc/sys/kernel/hostname‘
   - ‘cat /proc/sys/vm/swappiness‘
   - ‘sysctl -a — less‘ (lists all readable kernel parameters)

Remember: Modifying files in ‘/proc/sys/‘ can change kernel behavior. Only do so if you understand the implications or in a safe virtual machine environment.