Module 2: Pages 2.2 to 2.11 (First 10 pages)

Page 2.2 — Module Scope and Topics

- Title and scope:
  - System Calls: What they are and how programs invoke OS services.
  - System/Application Call Interface: The layer through which applications use OS functionality.
  - Protection: Mechanisms to protect OS, processes, and resources.
  - User/Kernel Modes: Two privilege levels that separate normal apps from OS code.
  - Interrupts: Events that alter normal CPU flow to handle urgent tasks or hardware signals.
  - Processes: The core unit of execution for the OS.
  - Process Control Block (PCB): The data structure the OS uses to track each process.
  - Process States: Lifecycle states (new, ready, running, waiting, terminated).
  - Process Management in UNIX: How UNIX handles process creation, execution, and termination.
  - Threads: Multiple flows of execution, within or across processes.
  - User-level threads, Kernel-level threads, Thread Models: Where threads are managed and how mappings work.

Page 2.3 — System Calls: Definition and APIs

- Concept of system calls:
  - System calls are the programming interface to OS services.
  - They provide controlled entry points for applications to request operations that require privileges.
- Typical language and access route:
  - Usually invoked from high-level languages such as C or C++.
  - Most applications do not call raw system calls directly; they use high-level APIs.
- Common APIs:
  - Win32 API for Windows.
  - POSIX API for UNIX-like systems (UNIX, Linux, macOS).
  - Java API for programs running on the JVM.
- Naming and generality:
  - System-call names presented in the text are generic for teaching clarity.
  - Actual names vary by OS/API, but the concepts match.

Page 2.4 — Example: System Call Sequence for File Copy

- Purpose of the example:
  - Demonstrates how copying one file to another uses a sequence of system calls through an API.
- Typical steps explained simply:
  - Open source file for reading.

    - Create or open destination file for writing.
    - Loop to read a block from source.
    - Write the block to destination.
    - Repeat until read returns end-of-file.
    - Close both files.
- Key takeaways:
    - Each open, read, write, and close is a system call or library call that triggers a system call.
    - Error handling typically checks return codes after each call.
    - The OS enforces permissions and provides buffering and file positioning.

Page 2.5 — Relationship: API, System Call, and OS

- Layers involved:
    - Application calls a library function (API call).
    - The API function translates into one or more system calls.
    - The OS kernel executes the privileged operations.
- Benefits of an API:
    - Hides OS-specific details and calling conventions.
    - Simplifies development and improves portability.
    - Provides error codes and standardized behavior.
- Execution flow summary:
    - App → API function → System-call interface → Kernel service → Return status and data → App.

Page 2.6 — System Call Implementation Details

- System call numbering and tables:
    - Each system call has an associated number.
    - The OS maintains a system-call table indexed by these numbers.
- System-call interface role:
    - Receives the call, finds the appropriate kernel function, passes arguments.
    - Returns results and status to the calling process.
- Transparency to the caller:
    - Applications need not know the internal mechanisms.
    - They must follow the API contract and understand expected results and errors.
- Runtime and libraries:
    - Many details are managed by runtime support libraries provided with the compiler or system.
    - These libraries prepare parameters, issue the trap to kernel mode, and handle results.

Page 2.7 — System Call Parameter Passing Methods

- Why parameters matter:
    - Most calls need arguments beyond just the call identity.
    - Types and amounts differ per OS and per call.
- Three general methods:
    - Registers:
        - Parameters are placed directly into CPU registers.

        - Fast but limited by the number of registers.
    - Block in memory:
        - Place arguments in a memory block or table and pass the block's address via a register.
        - Used by Linux and Solaris; removes register count limitations.
    - Stack:
        - Push parameters onto the stack; OS pops them when handling the call.
        - Flexible for variable numbers and sizes of parameters.
- Advantages of block/stack:
    - No hard limit on parameter count or size.
    - Easier to expand arguments without changing CPU conventions.

Page 2.8 — Types of System Calls: Process Control

- Process control operations:
    - Create process, terminate process.
    - End, abort: ways to stop execution and return control.
    - Load, execute: start a specific program image in a process context.
    - Get/set process attributes: priorities, scheduling parameters, identity info.
    - Wait for time: sleep or delay the process.
    - Wait event, signal event: synchronization primitives to coordinate processes.
    - Allocate and free memory: basic memory management requests.
    - Dump memory on error: aid debugging by capturing process state.
    - Debugging support: single-stepping, breakpoints, inspecting memory and registers.
    - Locks for shared data: mechanisms for safe access to shared state.

Page 2.9 — Types of System Calls: File and Device Management

- File management calls:
    - Create and delete files.
    - Open and close files.
    - Read, write, and reposition (seek) within files.
    - Get and set file attributes (permissions, timestamps, sizes).
- Device management calls:
    - Request and release a device (exclusive or shared use).
    - Read, write, and reposition for device streams.
    - Get and set device attributes (modes, configurations).
    - Logical attach/detach devices (associate a driver or make devices available).

Page 2.10 — Types of System Calls: Information and Communication

- Information maintenance:
    - Get or set system time and date.
    - Get or set system data (tunable parameters, identifiers).
    - Get or set attributes of processes, files, or devices.
- Communications system calls:
    - Create and delete communication connections (sockets, pipes).
    - Send and receive messages in message-passing models; identify endpoints by host or process names.

    - Shared-memory model: create and map shared memory regions for direct access.
    - Transfer status information: return codes, error info, connection state.
    - Attach and detach remote devices: integrate remote resources as if local.

Page 2.11 — Types of System Calls: Protection

- Protection-related operations:
    - Control access to resources (files, devices, IPC objects).
    - Get and set permissions (read, write, execute, ownership).
    - Allow or deny user access (authorization decisions).
- Why this matters:
    - Ensures confidentiality, integrity, and proper isolation.
    - Prevents unauthorized use and enforces multi-user security policies.

Module 2: Pages 2.12 to 2.21

Page 2.12 — Standard C Library Example

- Purpose:
    - Shows how a high-level C library function (printf) ultimately invokes a lower-level system call (write) to perform output.
- Flow explained:
    - printf formats the string in user space using the C standard library.
    - After formatting, the library issues a write request to send bytes to a file descriptor (e.g., stdout).
    - The library transitions to kernel mode via a system-call entry (software interrupt/trap).
    - The kernel validates the file descriptor, checks permissions, and writes to the device or file buffer.
    - Control returns to user space with a result (e.g., number of bytes written or an error code).
- Key point:
    - Most applications do not call write directly; they use higher-level functions like printf that wrap system calls.

Page 2.13 — Example: MS-DOS (Single-Tasking System)

- Characteristics:
    - Single-tasking: only one program runs at a time.
    - Shell is invoked at boot and provides a command interface.
    - Simple run model: running a program does not create a separate process abstraction.
    - Single memory space: the loaded program overwrites most of memory except the resident kernel.
    - On program exit: control returns to the shell, which is reloaded if necessary.
- Implications:
    - No concurrent processes; no preemptive multitasking.
    - Limited protection and isolation as all run in a flat space.
    - Simplicity comes at the cost of flexibility, concurrency, and safety.

Page 2.14 — Protection: Dual-Mode Operation

- Concept:
  - Two privilege modes enforced by hardware: user mode and kernel mode.
  - A mode bit indicates current privilege level.
- Behavior:
  - When user code runs, the mode bit indicates user mode; privileged instructions are disallowed.
  - When kernel code executes (e.g., during a system call or interrupt), the mode bit indicates kernel mode; privileged instructions are allowed.
- Safety guarantee:
  - User programs cannot directly set the mode to kernel; elevation happens only via controlled entry points (system calls, interrupts).
  - On system-call entry, hardware switches to kernel mode; on return, the OS restores user mode.
- Privileged instructions:
  - Operations like changing the timer, manipulating device registers, or updating page tables are restricted to kernel mode.

Page 2.15 — Transition from User to Kernel Mode (Figure Explanation)

- What the figure conveys:
  - An application executes in user mode.
  - It requests a service that requires privileges (e.g., file I/O).
  - A controlled trap instruction or software interrupt transfers control to the OS entry stub.
  - The CPU switches to kernel mode; the kernel's system-call handler runs.
  - After completing the service, the kernel prepares return values, restores user context, and returns to user mode.
- Key points:
  - The transition preserves process state.
  - The boundary crossing is the foundation of protection and controlled access.

Page 2.16 — Protection: Timer

- Purpose of the timer:
  - Prevents processes from monopolizing the CPU (infinite loops, starvation).
  - Ensures the OS regains control periodically to enforce scheduling and fairness.
- Mechanism:
  - The OS sets a counter (privileged operation) that is decremented by the physical clock.
  - When the counter reaches zero, a timer interrupt occurs.
  - The interrupt forces a transition to kernel mode so the scheduler can run.
- Usage:
  - The OS sets the timer before dispatching a process.
  - If a program exceeds its time slice, the timer interrupt preempts it.
  - The OS may also use the timer to terminate or manage misbehaving programs.

Page 2.17 — Processes (Section Overview Slide)

- This slide introduces three subtopics:
  - Process Concept: what a process is and how it is represented.
  - Process Scheduling: how the OS chooses which process runs.
  - Operations on Processes: creation, termination, and related management actions.
- Note:
  - Subsequent pages expand each of these areas in detail.

Page 2.18 — Process: Objectives

- Learning goals:
  - Understand a process as a program in execution and the fundamental unit of computation.
  - Learn about key process features: scheduling, creation, termination, and communication.
  - Explore interprocess communication (IPC) using shared memory and message passing.
- Importance:
  - Processes are the primary abstraction by which OS allocates CPU and resources.
  - IPC is essential for building cooperative and concurrent applications.

Page 2.19 — Process Concept (Structure of a Process)

- What a process is:
  - A program in execution that progresses sequentially.
- Types of programs executed:
  - In batch systems: jobs.
  - In time-shared systems: interactive user programs or tasks.
- Components that make up a process:
  - Text (code) section: the compiled instructions.
  - Current activity: program counter and CPU registers.
  - Stack: temporary data such as function parameters, return addresses, and local variables.
  - Data section: global/static variables.
  - Heap: memory dynamically allocated at runtime.
- Key point:
  - The OS tracks and manages all these components as part of the process's execution context.

Page 2.20 — Process Concept (Program vs Process; Creation)

- Program vs process:
  - Program is passive, stored on disk as an executable file.
  - Process is active, created when the executable is loaded into memory.
- How processes start:
  - User actions like GUI clicks or command-line invocations start execution.
- Multiple processes from one program:
  - The same program can be run by multiple users simultaneously, each instance a separate process.

    - Each process has its own state, stack, heap, and resources even if code is the same.

Page 2.21 — Process in Memory (Figure Explanation)

- What the figure typically shows:
    - A process's address space layout with distinct regions.
    - Code (text) at one end, followed by data, heap (grows upward), and stack (grows downward).
- Purpose:
    - Visualizes how the OS organizes a process's memory.
    - Highlights separation of concerns: code is read-only, data/heap is writable, stack is for call frames.
- Practical implications:
    - Memory protection enforces access rules for each region.
    - Dynamic behavior: heap expansion via allocation, stack changes with function calls/returns.

Module 2: Pages 2.22 to 2.31

Page 2.22 — Process State

- Purpose:
    - Defines the standard lifecycle states a process moves through during execution.
- States and meanings:
    - New: The process is being created; resources and data structures are being initialized.
    - Running: The CPU is actively executing the process's instructions.
    - Waiting: The process is blocked, waiting for some event (e.g., I/O completion, a signal).
    - Ready: The process is prepared to run and is waiting for CPU time from the scheduler.
    - Terminated: The process has finished execution and is awaiting cleanup and removal by the OS.
- Key points:
    - Transitions occur due to events such as I/O requests, interrupts, scheduling decisions, or process exit.
    - Only processes in the ready queue can be chosen to enter the running state.

Page 2.23 — Diagram of Process State (Figure Explanation)

- What the diagram conveys:
    - It shows nodes labeled with states (new, ready, running, waiting, terminated) and arrows indicating possible transitions.
- Typical transitions explained:
    - New → Ready: After creation and admission by the OS.
    - Ready → Running: CPU scheduler dispatches the process.
    - Running → Waiting: Process requests I/O or waits for an event.
    - Waiting → Ready: Event completes (e.g., I/O done).

    - Running → Ready: Preemption by the scheduler (time slice expires or higher-priority process arrives).
    - Running → Terminated: Process finishes or aborts.
- Practical takeaway:
    - The diagram helps visualize scheduling behavior and why processes aren't always running.

Page 2.24 — Process Control Block (PCB)

- Purpose:
    - Central OS data structure that stores all information needed to manage a process.
- Information in the PCB:
    - Process state: Current lifecycle state (running, waiting, ready, etc.).
    - Program counter: Address of the next instruction to execute.
    - CPU registers: Saved register values for context switching.
    - CPU scheduling information: Priority, scheduling parameters, and queue pointers.
    - Memory-management information: Page tables, segment tables, base/limit registers.
    - Accounting information: CPU usage, elapsed time, job/account numbers, time limits.
    - I/O status information: Allocated I/O devices, list of open files, pending I/O operations.
- Significance:
    - On a context switch, the OS saves the old process's context in its PCB and loads the new one's context from its PCB.

Page 2.25 — CPU Switch From Process to Process (Figure Explanation)

- What the figure conveys:
    - A timeline-like depiction of switching the CPU from one process to another.
- Steps in a context switch:
    - Interrupt or trap occurs (e.g., timer, I/O completion, system call).
    - OS saves the current process's CPU state (program counter, registers) into its PCB.
    - OS selects a new process to run (dispatch decision).
    - OS loads the chosen process's CPU state from its PCB.
    - Control transfers to the new process at its saved program counter.
- Key idea:
    - Context switch is necessary for multitasking but incurs overhead because no useful work is done during the switch itself.

Page 2.26 — Process Scheduling

- Objective:
    - Maximize CPU utilization and ensure fair, responsive time sharing among processes.
- Roles:
    - Process scheduler: Selects which ready process to run next on the CPU.

- Scheduling queues:
  - Job queue: All processes in the system (regardless of state).
  - Ready queue: Processes in main memory, ready to execute when scheduled.
  - Device queues: Processes waiting for specific I/O devices to become available or complete requests.
- Process migration:
  - Processes move among ready, running, and device queues as they request I/O, get preempted, or complete events.

Page 2.27 — Ready Queue and Various I/O Device Queues (Figure Explanation)

- What the figure conveys:
  - Visual layout of one ready queue feeding the CPU and multiple device queues for different I/O devices.
- Explanation:
  - The ready queue holds processes prepared to execute.
  - Each device has its own waiting queue for processes that issued I/O to that device.
  - After I/O completion, processes return from a device queue to the ready queue.
- Core concept:
  - Separate queues allow the OS to manage CPU-bound and I/O-bound operations efficiently and independently.

Page 2.28 — Representation of Process Scheduling (Queueing Diagram)

- What the diagram conveys:
  - A flow model of processes moving among queues, the CPU, and I/O devices.
- Flow summary:
  - From job queue to ready queue upon admission.
  - From ready queue to CPU when dispatched.
  - From CPU to device queue on I/O request; from device queue back to ready queue after completion.
  - From CPU to terminated state on completion.
- Purpose:
  - Highlights the dynamics of scheduling and the interplay between compute and I/O activities.

Page 2.29 — Schedulers (Short-term, Long-term) and Process Mix

- Short-term scheduler (CPU scheduler):
  - Chooses the next ready process to run.
  - Invoked very frequently (milliseconds), so it must be very fast.
- Long-term scheduler (job scheduler):
  - Admits processes into the system (into the ready pool).
  - Invoked infrequently (seconds to minutes).
  - Controls the degree of multiprogramming (total number of processes in the system).
- Process types:
  - I/O-bound: Many short CPU bursts, spends more time waiting for I/O.

    - CPU-bound: Few long CPU bursts, spends more time computing.
- Goal of long-term scheduling:
    - Maintain a good mix of I/O-bound and CPU-bound processes to keep both CPU and I/O devices busy.

Page 2.30 — Addition of Medium-Term Scheduling (Swapping)

- Motivation:
    - If the system is overcommitted, temporarily reduce the degree of multiprogramming.
- Mechanism:
    - Swap out: Remove a process from main memory and save it to disk (suspended state).
    - Swap in: Bring the process back into memory later to continue execution.
- Benefits:
    - Frees main memory for active processes, improving responsiveness.
    - Allows the OS to manage memory pressure without terminating processes.

Page 2.31 — Example PCB (Field-by-Field Illustration)

- Purpose:
    - A concrete PCB example that shows typical stored values and why they matter.
- Fields and meanings:
    - PID: Unique identifier of the process (e.g., 1042).
    - Process State: Current lifecycle state (e.g., Running).
    - Program Counter: Address of the next instruction (e.g., 0x00403A22).
    - Stack Pointer: Top of the current stack (e.g., 0x7FFFAB34), used for calls/returns and locals.
    - Registers: CPU register values (e.g., EAX, EBX), capturing computation state.
    - Memory Pointers: Pointers to code, heap, and stack regions in the process's address space.
    - Open Files: A list or table indicating which descriptors are open (e.g., stdin, stdout).
    - Parent Process: PID of the parent (e.g., a shell), used for signaling and wait/exit semantics.
    - Priority: Scheduling priority (e.g., 5), influences dispatch order.
    - CPU Scheduling Info: Time-slice length and last used time to support time-sharing logic.
    - Accounting Info: Start time and accumulated CPU time for monitoring and quotas.
    - Call Stack Snapshot: Current call chain helpful for debugging and profiling.
    - Exit Code: Return status once the process terminates; null if still running.
- Takeaway:
    - The PCB centralizes everything the OS needs to pause and resume execution consistently and safely.

Module 2: Pages 2.32 to 2.41

Page 2.32 — Threads (Introduction)

- Context:
  - Up to now, a process has been treated as having a single execution thread.
- Key idea:
  - A process can have multiple program counters active at once, meaning different parts of the same program execute concurrently.
- Terminology:
  - Multiple threads of control inside one process are called “threads.”
- Requirements:
  - If a process has multiple threads, the OS must store per-thread details (e.g., thread IDs, stacks, registers).
  - The PCB must be extended or associated with thread-specific structures to hold multiple program counters and thread states.
- Note:
  - A subsequent chapter will discuss threads in more depth; this slide introduces the concept as a bridge.

Page 2.33 — Process Representation in Linux

- Representation:
  - Linux represents processes using a C structure called task_struct.
- Typical fields shown:
  - pid t_pid: Process identifier.
  - long state: Encodes the current state of the process.
  - unsigned int time_slice: Scheduling information (e.g., time quantum).
  - struct task_struct *parent: Pointer to the parent process’s task_struct.
  - struct list_head children: Linked list of this process’s child processes.
  - struct files_struct *files: Pointer to open file table for this process.
  - struct mm_struct *mm: Pointer to the process’s memory descriptor (address space).
- Implication:
  - task_struct consolidates scheduling, hierarchy, file descriptors, and memory mappings, enabling the kernel to manage processes effectively.

Page 2.34 — Multitasking in Mobile Systems

- iOS (early versions) model:
  - Only one foreground process is active and directly controlled by the user interface.
  - Other processes are suspended or heavily limited in background.
  - Background allowances include:
    - Handling single, short tasks.
    - Receiving and reacting to notifications.
    - Executing specific long-running tasks such as audio playback.
- Android model:
  - Allows foreground and background processes with fewer restrictions.
  - Background work is commonly implemented via Services:
    - A Service can keep running even when the associated app process is not in the foreground.
    - Services have no UI and generally consume small memory.
- Rationale:

    - Mobile UIs and limited screen real estate encourage a single active foreground app.
    - Battery and performance constraints influence background execution policies.

Page 2.35 — Context Switch

- Definition:
    - Switching the CPU from one process to another requires saving the old process's state and loading the new process's state.
- PCB role:
    - The process's context (registers, program counter, etc.) is stored in the PCB.
- Overhead:
    - Context-switch time is pure overhead; no useful computation occurs while switching.
    - The more complex the OS and PCB state, the longer context switching can take.
- Hardware support:
    - Some CPUs provide multiple register sets so that switching between contexts is faster.
    - Hardware features can significantly reduce the overhead of context switches.

Page 2.36 — Operations on Processes (Overview)

- Provided mechanisms:
    - Process creation: How new processes are spawned.
    - Process termination: How processes finish and are cleaned up.
    - Other related operations detailed on following pages (e.g., waiting, signaling).
- Importance:
    - These operations allow building complex applications that manage subprocesses and coordinate tasks.

Page 2.37 — Process Creation

- Parent-child relationship:
    - A parent process can create child processes, forming a tree.
- Identification:
    - Processes are managed using a unique process identifier (PID).
- Resource sharing options:
    - Parent and child share all resources.
    - Child shares a subset of the parent's resources.
    - Parent and child share no resources.
- Execution options:
    - Parent and child can execute concurrently.
    - Parent may wait for the child to terminate before proceeding.
- Use cases:
    - Building pipelines of tasks.
    - Launching helpers or workers from a main program.

Page 2.38 — Process Creation (Cont.)

- Address space options:
  - The child may be a duplicate of the parent (copy of address space).
  - The child may have a new program loaded into it.
- UNIX examples:
  - fork():
    - Creates a new process by duplicating the calling process.
    - Parent and child continue execution from the point after the fork.
  - exec():
    - Replaces the process's memory space with a new program image.
    - Typically used in the child after fork to run a different program.
- Practical pattern:
  - Common UNIX pattern is fork() followed by exec() to start a new program as a child process.

Page 2.39 — C Program Forking Separate Process (Figure/Code Explanation)

- What it shows:
  - A C code example illustrating fork() usage.
- Expected behavior:
  - After fork(), two nearly identical processes run:
    - The parent receives the PID of the child from fork() and can continue or wait.
    - The child receives 0 as the return value from fork() and can exec() a new program or continue with child-specific logic.
- Purpose:
  - Demonstrates how a program can create a separate process and manage different execution paths in parent and child.

Page 2.40 — Process Termination

- Normal exit:
  - The process executes its last statement and requests deletion via exit().
  - The exit status is returned to the parent, commonly via wait().
  - The OS deallocates all resources held by the process.
- Forced termination by parent:
  - Parent may terminate child with abort() (or OS-specific kill mechanisms).
  - Typical reasons:
    - Child exceeded allocated resources.
    - Child's task is no longer needed.
    - Parent is exiting and the OS policy disallows children to continue without their parent.
- Cleanup:
  - Proper termination ensures memory and OS resources (files, devices) are released.

Page 2.41 — Process Termination (Cascading, Wait, Zombies, Orphans)

- Cascading termination:
  - Some OSes enforce that if a parent terminates, all descendants must also terminate.

    - The OS initiates termination of children, grandchildren, etc.
- Waiting for child:
    - The parent can wait for its child using wait(), which returns the child's PID and status.
    - Example call form: pid = wait(&status);
- Zombies:
    - If a child terminates and no parent is waiting (no wait() called), the child becomes a zombie.
    - A zombie retains an entry for its exit status until the parent collects it.
- Orphans:
    - If a parent terminates without waiting, the child becomes an orphan.
    - Orphans are typically reparented to a special system process (e.g., init) which will eventually reap them.
- Key point:
    - Proper use of wait() prevents resource leaks associated with zombie processes.

Module 2: Pages 2.42 to 2.51

Page 2.42 — Multiprocess Architecture: Chrome Browser

- Motivation for multiprocess browsers:
    - Older browsers ran as a single process, so a faulty website or plugin could crash or hang the entire browser.
- Chrome's architecture (separate processes):
    - Browser process:
        - Manages the user interface, tabs, windows, and high-level control.
        - Handles disk I/O (history, cache) and network I/O (requests, responses).
    - Renderer process:
        - Renders web pages and executes HTML, CSS, and JavaScript.
        - Usually one renderer per website or tab, improving fault isolation.
        - Runs in a sandbox with restricted disk and network access to reduce impact of security exploits.
    - Plug-in process:
        - Each plugin type runs in its own process.
        - Isolates plugin crashes and improves stability and security.
- Benefits:
    - Fault isolation (one bad tab or plugin doesn't crash everything).
    - Security via sandboxing and reduced privileges for renderers.
    - Better responsiveness and recovery from failures.

Page 2.43 — Parameter Passing via Table (Figure Explanation)

- Purpose of the figure:
    - Illustrates the "block/table" method for system call parameter passing.
- Explanation:
    - The user program places arguments into a memory block (a parameter table).
    - It passes the address of that block to the system call interface (e.g., via a register).

    - The kernel, upon entry, reads the parameters from this table in a validated, structured way.
- Why it's used:
    - Avoids limitations on the number of registers.
    - Supports variable-length parameter sets and larger argument data.
    - Improves portability and simplifies ABI conventions.

Page 2.44 — Threads (Section Divider)

- Purpose:
    - Introduces the next major topic: threads.
- Implication:
    - Subsequent pages detail thread concepts, models, benefits, libraries, and programming on multicore systems.

Page 2.45 — Threads: Section Outline

- Topics listed:
    - Overview: What threads are and why they're used.
    - Multicore Programming: Challenges and opportunities when multiple cores are available.
    - Multithreading Models: Mapping between user threads and kernel threads.
    - Thread Libraries: APIs and implementations developers use.
    - Implicit Threading: Techniques where systems/libraries manage threading automatically.
    - Threading Issues: Synchronization, scheduling, and correctness considerations.
    - Operating System Examples: How different OSes implement or expose threading.
- Guidance:
    - This page is a roadmap; the following pages expand each point.

Page 2.46 — Threads: Objectives

- Learning goals:
    - Understand a thread as the fundamental unit of CPU utilization within a process.
    - Recognize that modern systems and applications are multithreaded for performance and responsiveness.
    - Get familiar with common thread APIs:
        - POSIX Pthreads (C-based).
        - Windows Threads (Win32).
        - Java Threads (JVM-based).
- Key emphasis:
    - Distinguish between specification (API) and implementation (user-level vs kernel-level support).

Page 2.47 — Motivation for Threads

- Why multithreading:

    - Most modern applications decompose their work into concurrent tasks.
    - Threads allow separate tasks to proceed seemingly in parallel within one process.
- Example tasks implemented as separate threads:
    - Updating the graphical display.
    - Fetching data from disk or network.
    - Spell checking or background computation.
    - Serving inbound network requests.
- Efficiency benefits:
    - Creating a process is heavy-weight; creating a thread is lighter (fewer resources to duplicate).
    - Kernels themselves are typically multithreaded to handle concurrent system activities.
- Code and performance:
    - Threading can simplify code structure for concurrency and improve throughput and responsiveness.

Page 2.48 — Thread Basics

- Per-thread state:
    - Thread ID: Unique identifier within the process or system scope.
    - Program Counter (PC): Next instruction address for the thread's execution.
    - Register Set: CPU registers specific to the thread's current context.
    - Stack: Private call stack for function calls, local variables, and return addresses.
- Shared among threads in the same process:
    - Code (text) section.
    - Data section (globals, statics).
    - Other OS-level resources like open files and signal dispositions.
- Terminology:
    - Traditional or heavyweight process: Single thread of control.
    - Multithreaded process: More than one thread, enabling intra-process multitasking.

Page 2.49 — Single and Multithreaded Processes (Figure Explanation)

- What the figure conveys:
    - A single-threaded process has one execution context tied to one stack and one PC.
    - A multithreaded process shows multiple execution contexts (each with its own PC and stack) sharing the same code and data segments.
- Key insight:
    - Multithreading enables concurrent operations within a single address space, avoiding the overhead of multiple processes while sharing resources.

Page 2.50 — Benefits of Multithreading

- Responsiveness:
    - If one thread blocks (e.g., waiting for I/O), other threads can continue, which is crucial for interactive applications.

- Resource sharing:
    - Threads share process resources, simplifying coordination compared with inter-process communication.
- Economy:
    - Thread creation and switching overhead is lower than process creation and context switching.
- Scalability:
    - Threads allow programs to utilize multiple processors/cores, increasing parallel performance when designed appropriately.

Page 2.51 — Multithreaded Server Architecture (Figure Explanation)

- What the figure conveys:
    - A server architecture where each client request is handled by a dedicated thread.
- Operational flow:
    - A main thread listens for incoming client connections or requests.
    - For each connection, the server creates or assigns a worker thread to handle request processing.
    - Multiple worker threads execute concurrently, sharing server resources (cache, DB connections) safely with synchronization.
- Advantages:
    - Improved throughput and responsiveness under concurrent loads.
    - Simplifies request handling logic by using per-request or per-connection threads.

Module 2: Pages 2.52 to 2.61

Page 2.52 — Multicore Programming: Challenges and Concepts

- Context:
    - Multicore and multiprocessor systems pressure programmers to parallelize applications effectively.
- Key challenges:
    - Dividing activities:
        - Break the overall task into smaller concurrent tasks that can run in parallel.
    - Balance:
        - Ensure tasks have comparable workloads so no core is idle while others are overloaded.
    - Data splitting:
        - Partition data so each thread/core works on its own subset, minimizing contention and sharing.
    - Data dependency:
        - Identify and manage dependencies to avoid race conditions and ensure correct ordering.
    - Testing and debugging:
        - Concurrency bugs are intermittent and harder to reproduce; tools and careful design are needed.
- Definitions:

    - Parallelism:
        - System can perform more than one task simultaneously (requires multiple cores/CPUs).
    - Concurrency:
        - System supports more than one task making progress; even on a single core, the scheduler interleaves execution to create the illusion of simultaneous progress.

Page 2.53 — Multicore Programming (Cont.): Types of Parallelism and Hardware Support

- Types of parallelism:
    - Data parallelism:
        - Distribute different subsets of the same data across multiple cores; each core performs the same operation on its assigned data.
    - Task parallelism:
        - Distribute threads across cores where each thread executes a different operation or stage of a pipeline.
- Hardware trends:
    - As thread count grows, hardware provides more support (more cores and more hardware threads).
    - Example platform mentioned:
        - Oracle SPARC T4 with 8 cores and 8 hardware threads per core, illustrating deep hardware multithreading capacity.

Page 2.54 — Concurrency vs. Parallelism (Figure Explanation)

- What the figure conveys:
    - Concurrency on a single-core system:
        - Multiple tasks advance by time-sliced interleaving; at any instant only one runs, but over time they all make progress.
    - Parallelism on a multi-core system:
        - Multiple tasks run literally at the same time on different cores, achieving true simultaneous execution.
- Practical takeaway:
    - Concurrency is about structure (how tasks are composed) and interleaving, while parallelism is about simultaneous execution leveraging hardware.

Page 2.55 — Amdahl's Law: Limits to Parallel Speedup

- Purpose:
    - Quantifies speedup potential of parallelizing an application with both serial and parallel parts.
- Variables:
    - S: Fraction of the program that is serial (cannot be parallelized).
    - N: Number of processing cores.
- Insights:
    - If an application is 75% parallel and 25% serial, going from 1 to 2 cores yields roughly 1.6× speedup.
    - As $N \to \infty$, maximum speedup approaches 1/S (the serial fraction limits overall speed).

    - The serial portion disproportionately constrains benefits from adding cores.
- Consideration:
    - Modern multicore systems, memory hierarchies, and overheads can influence real-world results beyond the idealized model.

Page 2.56 — User Threads and Kernel Threads

- User threads:
    - Managed by a user-level thread library in user space.
    - Common libraries:
        - POSIX Pthreads.
        - Windows threads (API).
        - Java threads (JVM).
- Kernel threads:
    - Supported and managed directly by the OS kernel.
    - Examples of OS with kernel threads:
        - Windows, Solaris, Linux, Tru64 UNIX, macOS.
- Distinction:
    - User threads can be very fast to create/switch but require mapping to kernel entities to run on CPUs.
    - Kernel threads are scheduled by the OS and can run truly in parallel on multicore systems.

Page 2.57 — Multithreading Models: Overview

- Mapping strategies between user-level threads and kernel-level threads:
    - Many-to-One.
    - One-to-One.
    - Many-to-Many.
- Purpose:
    - Each model balances performance, concurrency, and implementation complexity differently.
- Implication:
    - The chosen model affects concurrency, blocking behavior, and scalability on multicore hardware.

Page 2.58 — Many-to-One Model

- Definition:
    - Many user-level threads are mapped onto a single kernel thread.
- Characteristics:
    - If one user thread blocks (e.g., on I/O), all user threads block because the single kernel thread is blocked.
    - Multiple user threads cannot run in parallel on a multicore system since only one kernel thread exists for the process.
- Usage:
    - Few systems use this model today due to limitations.
- Examples:
    - Solaris Green Threads.
    - GNU Portable Threads.

Page 2.59 — One-to-One Model

- Definition:
  - Each user-level thread maps to a separate kernel thread.
- Characteristics:
  - Greater concurrency than many-to-one; blocking of one thread does not stall others.
  - True parallelism is possible on multicore systems.
- Trade-offs:
  - Creating a user thread also creates a kernel thread; this can introduce overhead.
  - Some systems limit the number of threads per process to manage overhead.
- Examples:
  - Windows, Linux, Solaris 9 and later.

Page 2.60 — Many-to-Many Model

- Definition:
  - Many user-level threads are mapped to many kernel threads.
- Advantages:
  - The OS can create an appropriate number of kernel threads based on workload and system resources.
  - Allows user-level thread libraries to schedule user threads flexibly while still enabling kernel-level concurrency.
- Examples:
  - Solaris prior to version 9.
  - Windows with the ThreadFiber package (mechanism to combine fibers and threads).

Page 2.61 — Two-level Model

- Definition:
  - Similar to Many-to-Many, but allows binding a specific user thread to a specific kernel thread when needed.
- Benefits:
  - Offers flexibility of M:M plus the ability to optimize certain threads (e.g., latency-sensitive) via binding.
- Examples:
  - IRIX, HP-UX, Tru64 UNIX, Solaris 8 and earlier.

Great — we'll finish the module with the final 4 pages. Below are detailed, page-by-page notes for pages 2.62 to 2.65, covering everything on each slide with clear headings and flat bullets.

Module 2: Pages 2.62 to 2.65

Page 2.62 — Thread Libraries

- Purpose:

    - Provide a programmer-facing API to create, manage, and synchronize threads without dealing directly with low-level OS internals.
- What a thread library offers:
    - Functions to create and join threads.
    - Primitives for synchronization (mutexes, condition variables, sometimes semaphores and read–write locks).
    - Facilities to set thread attributes (stack size, detach state, scheduling parameters where supported).
- Implementation styles:
    - Entirely in user space:
        - Thread operations are handled by a user-level runtime.
        - Faster creation/switching and flexible scheduling policies inside a process.
        - Limitations: blocking system calls can block the whole process if not integrated with non-blocking I/O or special wrappers; true parallel execution across cores typically requires kernel support.
    - Kernel-level library:
        - Backed by kernel threads scheduled by the OS.
        - True parallelism on multicore systems.
        - Blocking calls do not stall other threads in the same process.
        - Typically higher overhead for creation and context switches than pure user-level.
- Practical note:
    - Many modern systems provide kernel-backed thread libraries, while some language runtimes combine user-level scheduling concepts on top of kernel threads for efficiency.

Page 2.63 — Pthreads (POSIX Threads): Overview

- Nature and scope:
    - POSIX standard (IEEE 1003.1c) specifying a C-based API for thread creation, management, and synchronization.
    - It defines the interface (specification), not a single implementation; different UNIX-like systems provide compatible implementations.
- Where it’s common:
    - Widely used on UNIX, Linux, and macOS, and available on many other POSIX-compliant platforms.
- Deployment model:
    - May be provided as user-level threads, kernel-level threads, or a combination, depending on the OS and implementation.
- Why Pthreads matter:
    - Portability: code using Pthreads can compile and run across many POSIX systems.
    - Completeness: includes essential threading and synchronization primitives used in systems and high-performance applications.

Page 2.64 — Pthreads: Key API Elements

- Core types and functions:
    - pthread_t:

        - Opaque type representing a thread ID (handle for thread operations).
    - pthread_create:
        - Creates a new thread within the calling process.
        - Parameters typically include a pthread_t*, optional attributes, the start routine (function pointer), and a single void* argument passed to the start routine.
        - Returns 0 on success; non-zero error codes on failure.
    - pthread_join:
        - Waits for a specific joinable thread to terminate.
        - Retrieves the thread's return value (void*) if provided.
        - Ensures resources for that thread are cleaned up.
    - pthread_exit:
        - Terminates the calling thread explicitly and optionally returns a value for a joiner to collect.
        - If the main thread calls pthread_exit, the process can continue running other threads.
    - pthread_mutex_t:
        - Mutex type for mutual exclusion, guarding critical sections.
        - Typically initialized statically with PTHREAD_MUTEX_INITIALIZER or dynamically with pthread_mutex_init.
    - pthread_mutex_lock / pthread_mutex_unlock:
        - Acquire and release the mutex around critical sections to prevent data races.
    - pthread_cond_t:
        - Condition variable for signaling between threads.
        - Used with a mutex to wait for and signal state changes (e.g., producer–consumer coordination).
- Additional common pieces (implicitly related):
    - pthread_attr_t for thread attributes (stack size, detach state).
    - pthread_detach to mark a thread as detached (no join needed; resources reclaimed automatically on termination).
    - pthread_cond_wait / pthread_cond_signal / pthread_cond_broadcast for condition-based coordination.
    - pthread_rwlock_t for read–write locks (when available).
    - pthread_barrier_t for barrier synchronization (when available).

Page 2.65 — Pthreads Example (Code Walkthrough)

- What the example demonstrates:
    - Minimal end-to-end flow of creating a thread, running a function in that thread, and waiting for it to finish with join.
- Code behavior explained:
    - Includes headers for Pthreads and standard I/O.
    - Defines a thread start routine print_message(void*):
        - Casts the incoming void* to char* and prints a message.
        - Returns NULL upon completion.
    - In main:
        - Declares pthread_t thread_id to hold the new thread's handle.

        - Sets a const char* message = "Hello from pthread!" to pass to the thread.
        - Calls pthread_create(&thread_id, NULL, print_message, (void*)message):
            - If non-zero is returned, thread creation failed; prints an error and exits with status 1.
            - On success, a new thread is running concurrently and will execute print_message with the provided argument.
        - Calls pthread_join(thread_id, NULL):
            - Blocks until the created thread completes.
            - Discards the return value (passing NULL for the second parameter).
        - Returns 0 to indicate successful completion.
- Key takeaways:
    - Thread functions must match the signature expected by pthread_create: a function returning void* and taking a void* argument.
    - Arguments to threads are passed by pointer; ensure the pointed-to data remains valid for the duration of thread use.
    - Use join for joinable threads to prevent resource leaks; alternatively, use detach if join is not needed.