Threads and Multithreading Models

Week 8

SDB

Autumn 2025

Agenda: Threads and Multithreading Models

Understanding Concurrent Execution within a Process

1. **Threads vs. Processes** *(Lighter-weight concurrency)*
2. **Benefits of Multithreading** *(Why use threads?)*
3. **User-level vs. Kernel-level Threads** *(Who manages them?)*
4. **Thread Libraries (e.g., POSIX `pthreads`)** *(How to create and manage threads)*
5. **Multithreading Models: 1:1, M:1, M:N** *(Mapping user code to kernel support)*

Agenda: Threads and Multithreading Models

Understanding Concurrent Execution within a Process

1. **Threads vs. Processes** *(Lighter-weight concurrency)*
2. **Benefits of Multithreading** *(Why use threads?)*
3. **User-level vs. Kernel-level Threads** *(Who manages them?)*
4. **Thread Libraries (e.g., POSIX `pthreads`)** *(How to create and manage threads)*
5. **Multithreading Models: 1:1, M:1, M:N** *(Mapping user code to kernel support)*

Think Ahead: Beyond Single-Tasking

Modern applications need to perform many tasks concurrently (e.g., a web browser rendering a page, playing audio, and downloading files simultaneously). How can an operating system enable this fine-grained parallelism efficiently, without incurring the high overhead of multiple separate processes?

Threads vs. Processes: Units of Concurrency I

Definition: Process

A **Process** is an executing instance of a program. It's an independent unit of resource allocation and protection (e.g., its own memory space, file handles). It's the primary unit of OS scheduling.

Definition: Thread

A **Thread** (or lightweight process) is a basic unit of CPU utilization within a process. Threads share the process's resources (code, data, open files) but have their own program counter, stack, and register set.

Threads vs. Processes: Units of Concurrency II

Key Differences:

Feature	Process	Thread
Independence	Independent execution environment	Part of a process; can share/access other threads' data
Memory	Separate address space, isolated	Shared address space within the same process
Resource Sharing	Achieved via IPC (pipes, shared memory)	Direct access to shared data (global variables)
Overhead (Creation/Context Switch)	High (heavyweight)	Low (lightweight)
Scheduling	Managed by OS kernel	Managed by OS kernel (kernel threads) or user library (user threads)
Termination	If one process terminates, others run	If one thread terminates, others run (unless fatal error)
Fault Isolation	High (crash of one doesn't affect others)	Low (crash of one thread affects entire process)

Benefits of Multithreading: Efficiency and Responsiveness I

Why build applications with multiple threads?

- **Responsiveness:**
  - A program can remain responsive to user input while performing a long-running operation in a separate thread (e.g., a GUI doesn't freeze during a file save).
- **Resource Sharing:**
  - Threads within the same process share code, data, and resources (e.g., open files, memory heap). This is more efficient than IPC.
- **Economy (Lower Overhead):**
  - Creating and switching between threads is significantly faster and consumes fewer resources than processes. (e.g., 10-100x faster context switch).
  - Kernel threads: 100-200 instructions for context switch vs. 1000-2000 for processes.

Benefits of Multithreading: Efficiency and Responsiveness II

- **Scalability (Multi-core CPUs):**
  - On multi-core or multiprocessor systems, multiple threads can execute truly in parallel, significantly speeding up computation-bound tasks.

Caution: Synchronization is Critical!

Sharing data between threads requires careful synchronization mechanisms (e.g., mutexes, semaphores) to prevent **race conditions** and **data corruption**. This introduces new complexities for the programmer.

User-level vs. Kernel-level Threads: Management & Visibility I

- **User-level Threads (ULTs):**
  - Managed entirely by a **user-level library** (e.g., POSIX ‘pthreads‘ implementation on some older systems, Green Threads in Java).
  - The OS kernel is **unaware** of the existence of individual user threads; it only sees the containing process.
  - All threads of a process share a single kernel thread.
- **Kernel-level Threads (KLTs):**
  - Managed directly by the **Operating System kernel**.
  - The kernel is **aware** of and directly schedules each individual kernel thread.
  - Each user-level thread can be mapped to its own kernel thread.

User-level vs. Kernel-level Threads: Management & Visibility II

Trade-offs:

Feature	**User-level Threads (ULTs)**	**Kernel-level Threads (KLTs)**
Creation/Switching	Very Fast (no kernel mode switch)	Slower (requires kernel mode switch)
True Parallelism	No (one thread blocks, entire process blocks)	Yes (multiple threads can run concurrently on multi-core)
System Calls	Blocking system call blocks entire process	Blocking system call blocks only that thread
Scheduling	User-defined (library-managed)	OS-managed (system-wide scheduling)
OS Awareness	None (kernel sees only one "thread" per process)	Full (kernel sees and manages all threads)
Portability	More portable (library can run on different OS)	Less portable (dependent on OS kernel API)
Example OS	Older Solaris Green Threads, some specialized runtimes	Most modern OS (Linux, Windows, macOS)

POSIX Threads (pthreads) Example I

What are pthreads?

- POSIX Threads ('pthreads') is a **standard API (Application Programming Interface)** for thread creation and synchronization.
- It's widely used in Unix-like operating systems (Linux, macOS) and also available on Windows.
- It typically provides a 1:1 mapping (each 'pthread' corresponds to a kernel thread on modern OSes).

Basic Thread Creation in C:

```c
#include <pthread.h> // For pthreads API
#include <stdio.h>   // For printf
#include <unistd.h>  // For sleep

// Function that the new thread will execute
void* my_thread_function(void* arg) {
    printf("Hello from the new thread! Arg received: %s\n", (char*)arg);
    sleep(1); // Simulate some work
    printf("New thread exiting.\n");
    return NULL; // Thread returns NULL pointer
}
```

POSIX Threads (pthreads) Example II

```c
int main() {
    pthread_t thread_id; // Variable to store thread ID
    char* message = "Hello from main thread!";

    printf("Main thread: Creating a new thread.\n");

    // Create a new thread
    // Args: thread_id pointer, attributes (NULL for default),
    //       start routine, argument to start routine
    int ret = pthread_create(&thread_id, NULL, my_thread_function, (void*)message);
    if (ret != 0) {
        perror("pthread_create failed");
        return 1;
    }

    // Wait for the created thread to finish
    // Args: thread_id, pointer to store return value (NULL if not needed)
    printf("Main thread: Waiting for new thread to complete.\n");
    ret = pthread_join(thread_id, NULL);
    if (ret != 0) {
        perror("pthread_join failed");
        return 1;
    }

    printf("Main thread: New thread has finished. Exiting.\n");
    return 0;
}
```

POSIX Threads (pthreads) Example III

Compilation & Execution (on Linux/macOS):

```
gcc -o my_thread_app my_thread_app.c -lpthread # -lpthread links the pthreads library
./my_thread_app
```

Output will show interleaved messages from main and the new thread.
Possible Output:

```
Main thread: Creating a new thread.
Hello from the new thread! Arg received: Hello from main thread!
Main thread: Waiting for new thread to complete.
New thread exiting.
Main thread: New thread has finished. Exiting.
```

Multithreading Models: Mapping User Threads to Kernel Threads

Why do we need models?

- Operating systems differ in how they provide thread support.
- The model dictates how user-level threads (what the programmer writes) are mapped to kernel-level threads (what the OS schedules).

Multithreading Models: Mapping User Threads to Kernel Threads

Why do we need models?

- Operating systems differ in how they provide thread support.
- The model dictates how user-level threads (what the programmer writes) are mapped to kernel-level threads (what the OS schedules).

Three Primary Models:

1. **Many-to-One (M:1):** Many user threads map to a single kernel thread.
2. **One-to-One (1:1):** Each user thread maps to a unique kernel thread.
3. **Many-to-Many (M:N):** Many user threads map to a smaller or equal number of kernel threads.

Multithreading Models: Mapping User Threads to Kernel Threads

Why do we need models?

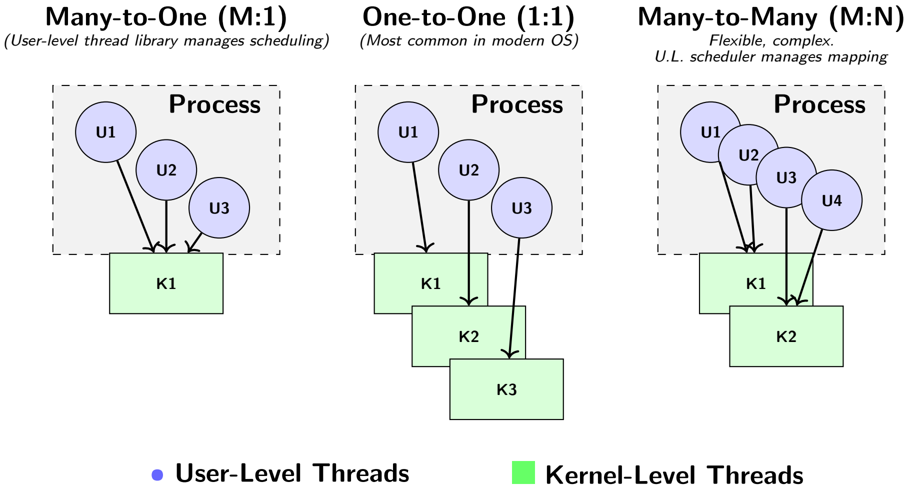
- Operating systems differ in how they provide thread support.
- The model dictates how user-level threads (what the programmer writes) are mapped to kernel-level threads (what the OS schedules).

Three Primary Models:

1. **Many-to-One (M:1):** Many user threads map to a single kernel thread.
2. **One-to-One (1:1):** Each user thread maps to a unique kernel thread.
3. **Many-to-Many (M:N):** Many user threads map to a smaller or equal number of kernel threads.

Thread Library's Role: The thread library (e.g., pthreads, Java Virtual Machine's thread system) is responsible for managing this mapping, creating/destroying user threads, and potentially managing them if they are user-level.

Thread Mapping Models (Illustration)



Many-to-Many Model (M:N): Hybrid Flexibility I

Concept: A flexible model where many user-level threads are multiplexed onto a smaller or equal number of kernel-level threads.

Key Characteristics:

- Combines the best features of M:1 and 1:1 models.
- The operating system creates a number of kernel threads (less than or equal to the number of user threads).
- The user-level thread library maps user threads to these available kernel threads dynamically.
- When a user thread makes a blocking system call, the user-level thread library can switch another user thread to run on the **same** kernel thread, preventing the entire process from blocking.

Many-to-Many Model (M:N): Hybrid Flexibility II

Pros & Cons:

- **Pros:**
  - True parallelism on multi-core systems (via multiple kernel threads).
  - Efficient handling of blocking system calls (one user thread blocks, others can run).
  - User-level thread management (fast context switching).
- **Cons:**
  - Significantly more complex to implement both in the OS kernel and the user-level thread library.
  - Requires complex coordination between the kernel and the user-level library.

Many-to-Many Model (M:N): Hybrid Flexibility III

Usage:

- Historically used in some older Unix systems (e.g., Solaris prior to version 9).
- Less common as a primary model in modern general-purpose OSes (like Linux, Windows) which predominantly use 1:1.
- However, concepts are found in highly concurrent runtimes and virtual machines (e.g., Go's goroutines, Java's virtual threads/fibers often implement a similar user-to-kernel thread mapping).

Discussion Prompt: Choosing the Right Threading Model

Consider the following application types: Which threading model(M:1, 1:1, M:N) would be most suitable, and why?
Justify your choice based on performance (context switching, parallelism), OS visibility, and overhead.

- **Scenario 1: A Scientific Computation Application**
  - Needs to perform 1000 highly parallel, CPU-bound computations simultaneously.
  - Each computation is independent and does not involve I/O.
- **Scenario 2: A Real-time Audio Processing Engine**
  - Requires extremely low latency and predictable response times.
  - Involves frequent I/O (reading audio data) and CPU processing.
- **Scenario 3: A High-Throughput Web Server**
  - Handles thousands of concurrent client requests.
  - Each request involves a mix of CPU processing and blocking I/O (e.g., reading from disk, network communication).

Key Takeaways I

- **Threads** are lightweight units of execution within a process, sharing resources but having independent execution contexts. They offer efficient concurrency.
- **Multithreading** provides significant benefits: improved responsiveness, efficient resource sharing, reduced overhead, and true parallelism on multi-core systems.
- **User-level threads** are managed by a library in user space (fast, but limited parallelism and blocking issues).
- **Kernel-level threads** are managed and scheduled directly by the OS kernel (true parallelism, but higher overhead).
- **Multithreading Models** define how user threads map to kernel threads:
    - **Many-to-One (M:1):** User-managed, no true parallelism, entire process blocks on I/O.
    - **One-to-One (1:1):** Kernel-managed, true parallelism, widely used in modern OSes.

Key Takeaways II

- **Many-to-Many (M:N):** Hybrid approach, flexible but complex.

- Careful **synchronization** is essential when using threads to prevent race conditions.

Reflection Prompt: Debugging Threads

If multiple threads within a single process are accessing and modifying a shared global variable without any synchronization, what kind of problem might arise? How would this manifest (e.g., crashes, incorrect results)? Why is this particularly challenging to debug?

Next Week Preview: Thread Synchronization

Ensuring Data Consistency and Order in Concurrent Programs

- **The Critical-Section Problem:** Understanding race conditions and mutual exclusion.
- **Hardware-based Solutions:** Test-and-Set, Compare-and-Swap.
- **Software-based Solutions:** Peterson's Solution.
- **Synchronization Tools:**
  - **Mutex Locks:** Basic mutual exclusion.
  - **Semaphores:** More generalized signaling mechanisms.
  - **Condition Variables:** For threads to wait on specific conditions.
- **Classic Synchronization Problems:** Bounded-Buffer, Readers-Writers, Dining-Philosophers.

Next Week Preview: Thread Synchronization

Ensuring Data Consistency and Order in Concurrent Programs

- **The Critical-Section Problem:** Understanding race conditions and mutual exclusion.
- **Hardware-based Solutions:** Test-and-Set, Compare-and-Swap.
- **Software-based Solutions:** Peterson's Solution.
- **Synchronization Tools:**
  - **Mutex Locks:** Basic mutual exclusion.
  - **Semaphores:** More generalized signaling mechanisms.
  - **Condition Variables:** For threads to wait on specific conditions.
- **Classic Synchronization Problems:** Bounded-Buffer, Readers-Writers, Dining-Philosophers.

Prep Tip for Next Session

Think about everyday scenarios where concurrent access to a shared resource could lead to problems (e.g., multiple people trying to update a shared calendar, multiple cashiers accessing the same inventory). This will help you understand the need for synchronization.

Outline

Quick Quiz: Threads and Models

Test Your Conceptual Understanding:

1. **Scenario:** A web browser tab freezes because a complex JavaScript calculation is running. Other tabs in the same browser process continue to work. What threading model is this browser likely using for its tabs? Justify your answer.
2. **True/False:** Creating 1000 user-level threads (M:1 model) will generally consume significantly more kernel memory than creating 1000 kernel-level threads (1:1 model). Justify your answer.
3. **Definition:** What is the primary advantage of a Many-to-Many (M:N) threading model over a Many-to-One (M:1) model, particularly on a multi-core processor?
4. **Problematic Aspect:** If multiple threads within a process need to increment a shared counter variable, why is direct unsynchronized access problematic, and what type of problem does it lead to?

Quick Quiz: Threads and Models

Test Your Conceptual Understanding:

1. **Scenario:** A web browser tab freezes because a complex JavaScript calculation is running. Other tabs in the same browser process continue to work. What threading model is this browser likely using for its tabs? Justify your answer.
2. **True/False:** Creating 1000 user-level threads (M:1 model) will generally consume significantly more kernel memory than creating 1000 kernel-level threads (1:1 model). Justify your answer.
3. **Definition:** What is the primary advantage of a Many-to-Many (M:N) threading model over a Many-to-One (M:1) model, particularly on a multi-core processor?
4. **Problematic Aspect:** If multiple threads within a process need to increment a shared counter variable, why is direct unsynchronized access problematic, and what type of problem does it lead to?

Think & Discuss

Formulate your answers before reviewing the solutions or discussing with peers.

Exercise: Applying Threading Concepts

Part 1: Thread Context Analysis
Consider a multi-threaded process. List the components of a process's context that are **shared** among its threads, and the components that are **unique** to each thread.
Part 2: Model Selection Scenario
You are developing a high-performance scientific simulation application that requires massive parallelism. The application generates millions of independent data points, each requiring a separate computation that is CPU-bound and has no I/O dependencies.
Task:

1. If you could choose any threading model, which one (M:1, 1:1, M:N) would be theoretically *most ideal* for this application's performance on a 64-core machine? Explain your choice in terms of parallelism and overhead.
2. If the only available threading library on your target embedded system supports an M:1 model, what practical limitations would you face in trying to achieve high performance for this application?
3. Now, imagine the simulation involves occasional heavy disk I/O (e.g., reading large datasets for some calculations). How would this change your preferred threading model, and why?

Reminder

Relate your answers back to the fundamental characteristics and trade-offs of processes and threads.

Appendix: Week 8 Advanced Topics to Explore I

Beyond Core Concepts: Deeper Dives into Threading

I. Threading Mechanisms & Libraries

- **Native Thread APIs:** Explore specific thread APIs beyond POSIX (e.g., Windows Thread API, Java Threads, C++ ‘std::thread‘).
- **Thread Pools:** How applications manage a pool of reusable threads to reduce creation/destruction overhead for short-lived tasks.
- **Fibers/Coroutines:** Even lighter-weight concurrency units than user-level threads, managed entirely by the application, offering cooperative multitasking.
- **Language-level Concurrency Primitives:** How modern languages (Go Goroutines, Rust ‘async‘/‘await‘, Python ‘asyncio‘) provide built-in support for concurrency.

Appendix: Week 8 Advanced Topics to Explore II

II. Concurrency vs. Parallelism & Performance

- **Amdahl's Law:** Understanding the theoretical speedup limits of parallelizing a task.
- **Cache Coherency & False Sharing:** How multi-core systems manage shared data in caches and potential performance pitfalls.
- **Memory Models:** How different architectures and languages guarantee (or don't guarantee) visibility of memory writes between threads (e.g., C++ memory model, Java memory model).
- **CPU Affinity (Revisited):** How threads can be "pinned" to specific CPU cores for performance reasons.

Appendix: Week 8 Advanced Topics to Explore III

III. Threading in Specific Contexts

- **Green Threads (Historical Context):** Understanding their role in early Java implementations and why they largely moved to native threads.
- **Process vs. Thread Pools in Web Servers:** When to use multiprocess vs. multithreaded models for handling requests in web server architectures.
- **Debugging Multithreaded Applications:** Specific challenges (race conditions, deadlocks, non-deterministic bugs) and tools (debuggers, thread sanitizers).