Chapter 1

Special Topic 1 — Parallelism & Concurrency

This Lecture felt pretty clear, didn't add too much to it, the video is also very interesting, I would recommend watching it (if you have time).

1.1 Introduction

Understanding the fundamental concepts of parallelism and concurrency through practical examples. Modern computing systems must efficiently utilize multiple resources to achieve optimal performance. This chapter explores two fundamental concepts that enable efficient resource utilization: **parallelism** and **concurrency**. We begin with an intuitive analogy to illustrate these concepts before diving into their technical implementations.

1.1.1 The Cooking Analogy

Consider preparing chicken fettuccine alfredo using different approaches to understand execution strategies.
To understand the difference between sequential, concurrent, and parallel execution, consider preparing a meal with the following tasks:

1. Boil water, cook pasta until al dente (15 minutes)
2. Dice chicken and mushrooms (5 minutes)
3. Fry chicken and mushrooms (10 minutes)
4. Make the sauce (10 minutes)
5. Put it all together (5 minutes)

Sequential Execution

Performing tasks one after another using a single resource.
In sequential execution, one person performs all tasks in order:

- Total time: 45 minutes
- Each task must complete before the next begins
- Simple but inefficient use of time and resources

Concurrent Execution

Overlapping tasks in time using intelligent scheduling.
A single person can optimize the process by overlapping tasks:

1. Start boiling water (2 minutes)
2. While water heats, dice chicken and mushrooms (5 minutes)
3. Add pasta to boiling water (13 minutes remaining)
4. While pasta cooks, fry chicken and mushrooms (10 minutes)
5. Make the sauce (10 minutes)
6. Put it all together (5 minutes)

Total time: 27 minutes — a 40% improvement through intelligent task scheduling.

Concurrent and Parallel Execution

Combining task overlap with multiple resources working simultaneously.
Two people working together can achieve even better performance:

- Tasks are both overlapped in time AND performed simultaneously by different people
- Chicken and mushrooms diced by two people in parallel (2.5 minutes each)
- Other optimizations through coordination
- Total time: 19.5 minutes — a 57% improvement over sequential execution

1.2 Fundamental Definitions

Precise definitions of concurrency and parallelism in computing contexts.

1.2.1 Concurrency

Concurrency occurs when multiple tasks execute in an overlapping time period (but not necessarily simultaneously) such that at least one task begins before another task finishes.

Computing Implementation. A single CPU achieves concurrency by rapidly switching between threads (time slicing). The CPU switches between tasks so quickly that it creates the illusion of simultaneous execution while allowing application execution to overlap with blocking system calls (disk I/O or network I/O).

1.2.2 Parallelism

Parallelism focuses on performing multiple tasks or several parts of a single task simultaneously by using multiple processing units.

Computing Implementation. Multiple CPUs execute different tasks at the same time, truly parallelizing the process across multiple processing units.

1.3 Motivation for Concurrent and Parallel Computing

Understanding why these concepts are essential for modern computing systems.

1.3.1 Performance Benefits

From the user perspective, concurrency and parallelism provide:

- **Increased throughput** — more operations per second
- **Reduced latency** — tasks complete faster

From the system perspective, they enable:

- **Efficient hardware resource management**
- **Utilization of multiple CPU cores**
- **Hiding the cost of I/O operations**

These concepts are fundamental to both applications and operating systems. Consider how a web browser and an IDE can run simultaneously on your computer — this is concurrency in action.

1.4 Concurrency Mechanisms

Exploring different approaches to achieving concurrent execution.

1.4.1 Process-Level vs Thread-Level Concurrency

Different mechanisms exist for achieving concurrency between processes versus within a single process.

Process Limitations

While processes provide strong isolation guarantees, they are often too heavyweight for fine-grained concurrency:

- Processes don't share memory, requiring explicit communication
- Creating new processes has high overhead
- Context switching between processes is expensive due to memory and resource isolation requirements
- Limited scalability for applications requiring many concurrent tasks

This leads to the question: *Could we have a lighter-weight abstraction?*

1.5 The Thread Abstraction

Understanding threads as the fundamental unit of concurrent execution.

1.5.1 Thread Definition

A **thread** is the smallest unit of execution that can be scheduled by the operating system. At any moment, a CPU core executes exactly one thread. Processes can contain multiple threads that share certain resources while maintaining separate execution contexts.

1.5.2 Thread Characteristics

Within a single process, multiple threads share:

- **Address space** (heap, data, and text segments)
- **File descriptors**
- **Process-level resources**

Each thread maintains its own:

- **Stack** (local variables and function call history)
- **Program counter** (current instruction)
- **Register state**

Each thread is:

- **Independently scheduled** by the operating system
- **Capable of issuing system calls**

1.5.3 Memory Layout Comparison

Visualizing the difference between single-threaded and multi-threaded address spaces.

Single-Threaded Process

In a single-threaded process, the address space contains one stack:

Stack
Free Space
Heap
Data
Text

Multi-Threaded Process

In a multi-threaded process, each thread has its own stack while sharing other segments:

Stack 2
Free Space
Stack 1
Free Space
Heap (Shared)
Data (Shared)
Text (Shared)

Observations.

- Each thread has its own stack segment for local variables
- Threads share the code (text segment)
- Threads share the heap for dynamic memory allocation
- Threads share global data

1.6 Thread Management

Programming interfaces for creating and managing threads.

1.6.1 POSIX Thread API

The POSIX threading library (pthread) provides standard functions for thread management:

Thread Creation

```
pthread_create(&thread_id, &attributes, start_routine, arguments)
```

This function creates a new thread that begins execution by calling `start_routine(arguments)`.

Thread Synchronization

```
pthread_join(&thread_id, &return_value)
```

This function blocks the calling thread until the specified thread completes execution.

1.6.2 First Multithreaded Program

A practical example demonstrating thread creation and the challenges of shared memory.
Consider a program that increments a shared counter using two threads:

```c
#include <stdio.h>
#include <pthread.h>
#define NUM_ITER 100000
int counter = 0;  // Global variable shared among all threads

void *incr(void *arg) {
   printf("%s starts\n", (char *)arg);
   for (int i = 0; i < NUM_ITER; i++)
      counter = counter + 1;  // Critical: shared memory access
   return NULL;
}

int main(int argc, char *argv[]) {
   pthread_t t1, t2;

   // Create two threads
   pthread_create(&t1, NULL, incr, "T1");
   pthread_create(&t2, NULL, incr, "T2");

   // Wait for both threads to complete
   pthread_join(t1, NULL);
   pthread_join(t2, NULL);

   printf("Counter: %d (expected: %d)\n", counter, NUM_ITER*2);
   return 0;
}
```

Expected vs Actual Results. This program should increment the counter 200,000 times (100,000 per thread), but **each execution produces different, incorrect results**. This non-deterministic behavior reveals fundamental challenges in concurrent programming.

1.7 Concurrency Challenges

Understanding race conditions and their causes.

1.7.1 The Race Condition Problem

Analyzing why simple operations become complex in concurrent environments.
The statement `counter = counter + 1` appears atomic at the source code level, but the processor actually executes multiple instructions:

```
mov    counter, %eax    # Load counter value into register
add    $1, %eax         # Increment register value
mov    %eax, counter    # Store register value back to memory
```

1.7.2 Thread Interleaving Analysis

Step-by-step examination of how thread scheduling affects program correctness.
Consider the following execution sequence where both threads attempt to increment the counter from an initial value of 50:

Thread 1	**Thread 2**	**%eax**	**counter**
`mov counter, %eax`		50	50
`add $1, %eax`		51	50
CONTEXT SWITCH		51	50
	`mov counter, %eax`	50	50
	`add $1, %eax`	51	50
	`mov %eax, counter`	51	51
CONTEXT SWITCH		51	51
`mov %eax, counter`		51	51

Analysis. Despite two increment operations, the counter increases by only 1 instead of 2. Thread 1's work is lost because Thread 2 overwrote the counter value before Thread 1 completed its read-modify-write sequence.

1.7.3 Race Conditions and Data Races

Formal definitions of concurrent programming hazards.

- **Race condition**: The program's correctness depends on the relative timing or ordering of events
- **Data race**: One thread accesses a mutable variable while another thread writes to it without proper synchronization
- These conditions lead to non-deterministic behavior that varies between program executions
- Multi-core systems exacerbate these problems through true parallel execution

1.8 Synchronization Mechanisms

Techniques for coordinating thread execution and ensuring program correctness.

1.8.1 Atomic Operations

The fundamental requirement for correct concurrent programming.
To fix the counter increment problem, we need the three-instruction sequence to execute **atomically** — appearing as a single, indivisible operation from the perspective of all threads.

1.8.2 Critical Sections

Identifying and protecting shared resource access.
A **critical section** is a code block that accesses shared resources and must not be executed concurrently by multiple threads.
To ensure program correctness, we must enforce:

- **Atomicity**: Execute critical sections as uninterruptible blocks
- **Mutual exclusion**: Only one thread can execute a critical section at any time

1.8.3 Locks

The primary mechanism for implementing mutual exclusion.
Locks (also called mutexes) ensure atomicity through mutual exclusion:

- All threads competing for a critical section share a lock
- Only one thread can acquire the lock at a time (**lock holder**)
- Other threads must wait until the lock is released (**lock waiters**)

Lock Usage Pattern

```
lock_t mutex;           // Shared lock variable
...
lock(&mutex);           // Acquire exclusive access
counter = counter + 1;  // Critical section
unlock(&mutex);         // Release access
```

1.9 Lock Implementations

Different approaches to implementing mutual exclusion with their respective trade-offs.

1.9.1 Interrupt-Based Locks

Using hardware interrupt control for mutual exclusion.

Basic Approach

Disable interrupts during critical section execution:

```
void lock(lock_t *l) {
    disable_interrupts();
}

void unlock(lock_t *l) {
    enable_interrupts();
}
```

Mechanism

- Prevents hardware and timer interrupts
- Stops the scheduler from switching threads
- Code executes atomically between interrupt disable/enable

Evaluation

Advantages:

- Simple implementation
- Effective for low-complexity code

Disadvantages:

- Requires privileged operations (OS-level access)
- No support for multiple independent locks
- Only works on single-processor systems
- Vulnerable to abuse (programs can monopolize resources)
- Can cause hardware interrupt loss

1.9.2 Software-Only Solutions

Attempting to implement locks using only memory operations.

Naive Approach

Use a shared boolean variable to coordinate access:

```
bool lock_available = true;

void lock(bool *l) {
    while (!(*l));      // Wait until lock is available
    *l = false;         // Claim the lock
}

void unlock(bool *l) {
    *l = true;          // Release the lock
}
```

Critical Flaw

This approach suffers from the same race condition it attempts to solve:

1. Thread 1 checks lock status (available)
2. Thread 2 checks lock status (available) — *context switch*
3. Thread 1 claims lock
4. Thread 2 claims lock — *context switch*
5. Both threads enter critical section simultaneously

Conclusion. Pure software solutions cannot solve the mutual exclusion problem without hardware support.

1.9.3 Hardware-Supported Atomic Instructions

Leveraging processor capabilities for correct lock implementation.

Test-and-Set Instruction

Modern processors provide atomic read-modify-write instructions. The **test-and-set** instruction atomically:

1. Reads the current value from a memory location
2. Writes a new value to that location
3. Returns the original value

```
int test_and_set(int *ptr, int new_value) {
   int old_value = *ptr;     // Read current value
   *ptr = new_value;         // Write new value
   return old_value;         // Return original value
}
```

Atomicity Guarantee. The processor guarantees that these three operations execute as a single, uninterruptible instruction across all CPU cores.

Spinlock Implementation

Using test-and-set to implement a correct lock:

```
bool lock_taken = false;

void lock(bool *l) {
    while (test_and_set(l, true)) {
        // Spin until we successfully acquire the lock
        // (test_and_set returns false, meaning lock was available)
    }
}

void unlock(bool *l) {
    *l = false;
}
```

Spinlock Evaluation

Advantages:

- Correctly implements mutual exclusion
- Works on multiprocessor systems
- No kernel involvement required

Disadvantages:

- **Busy waiting**: Lock waiters consume CPU cycles continuously
- Poor performance when lock contention is high
- Can lead to priority inversion problems

1.9.4 Blocking Locks (Mutexes)

Eliminating busy waiting through operating system cooperation.

The Busy-Waiting Problem

Spinlocks waste CPU cycles when threads wait for locks. Instead of spinning, waiting threads should **yield the CPU** to other threads.

Mutex Approach

Mutexes (mutual exclusion locks) address busy waiting:

- Threads attempting to acquire a held lock go to sleep
- The operating system maintains a queue of waiting threads
- When a lock is released, the OS wakes up one or more waiting threads
- Threads only consume CPU time when they can make progress

POSIX Mutex Implementation

```c
#include <pthread.h>

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void critical_section(int *counter) {
    pthread_mutex_lock(&mutex);      // Block if lock is held
    *counter += 1;                   // Critical section
    pthread_mutex_unlock(&mutex);    // Wake up waiting threads
}
```

Mutex Semantics.

- Threads block (sleep) when the lock is unavailable
- Integrates with other synchronization primitives
- Provides better overall system performance than spinlocks

1.10 Complex Synchronization: The Dining Philosophers Problem

A classic problem illustrating the subtleties of concurrent programming.
Even with proper locking mechanisms, designing correct concurrent programs remains challenging. The dining philosophers problem demonstrates common pitfalls.

1.10.1 Problem Description

Five philosophers sit around a circular table with five forks. Each philosopher alternates between thinking and eating. To eat, a philosopher needs both adjacent forks (left and right). The challenge is to design a protocol that allows all philosophers to eat without deadlock or starvation.

1.10.2 Basic Program Structure

```
void philosopher(int id) {
    while (true) {
        think(id);
        take_forks(id);      // Acquire both forks
        eat(id);
        put_forks(id);       // Release both forks
    }
}

int main() {
    pthread_t philosophers[5];
    for (int i = 0; i < 5; i++) {
        pthread_create(&philosophers[i], NULL, philosopher, i);
    }
    // Join all threads...
}
```

1.10.3 Attempt 1: Unprotected Shared State

```
bool fork_available[5] = {true, true, true, true, true};

void take_forks(int id) {
    int left = id;
    int right = (id + 1) % 5;

    while (!fork_available[left] || !fork_available[right]) {
        // Wait until both forks are available
    }
    fork_available[left] = false;   // Take left fork
    fork_available[right] = false;  // Take right fork
}
```

Problem: Data Race. Multiple philosophers can "take" the same fork simultaneously, violating the mutual exclusion requirement.

1.10.4 Attempt 2: Individual Fork Locks

```
pthread_mutex_t fork_locks[5] = {PTHREAD_MUTEX_INITIALIZER};

void take_forks(int id) {
    int left = id;
    int right = (id + 1) % 5;

    pthread_mutex_lock(&fork_locks[left]);   // Take left fork
    pthread_mutex_lock(&fork_locks[right]);  // Take right fork
}

void put_forks(int id) {
    int left = id;
    int right = (id + 1) % 5;

    pthread_mutex_unlock(&fork_locks[left]);
    pthread_mutex_unlock(&fork_locks[right]);
}
```

Problem: Deadlock. If all philosophers simultaneously pick up their left fork, they will all wait indefinitely for their right fork, creating a circular dependency.

1.10.5 Attempt 3: Breaking Circular Dependencies

```
void take_forks(int id) {
    int left = id;
    int right = (id + 1) % 5;

    if (id == 4) {  // Last philosopher uses different order
        pthread_mutex_lock(&fork_locks[right]);  // Right first
        pthread_mutex_lock(&fork_locks[left]);   // Then left
    } else {
        pthread_mutex_lock(&fork_locks[left]);   // Left first
        pthread_mutex_lock(&fork_locks[right]);  // Then right
    }
}
```

Solution Analysis. By having one philosopher acquire forks in reverse order, we break the circular dependency that causes deadlock. This ensures that not all philosophers can be blocked simultaneously.

1.10.6 The Need for Higher-Level Abstractions

Building correct concurrent programs from low-level primitives is challenging and error-prone. The complexity motivates the development of higher-level abstractions that hide synchronization details from programmers.

Parallel Programming Frameworks

- **OpenMP**: Compiler directives for shared-memory parallelism
- **CUDA**: GPU programming model for massively parallel computation
- **MapReduce/Spark**: Distributed computing frameworks that handle parallelism and fault tolerance
- **Transactional Memory**: Database-style transactions for memory operations

These frameworks allow programmers to focus on algorithmic concerns rather than low-level synchronization mechanisms, leading to more productive and less error-prone parallel programming.

Design Philosophy. The evolution from manual thread and lock management toward declarative parallel programming models reflects a broader trend in computer science: *raising the level of abstraction to hide complexity and reduce the opportunity for errors.*

Good Luck for the exam!