CPSC 4240/5240: Parallel Programming Techniques

Lecture 8: Concurrency and Locks

Lecturer: Quanquan C. Liu

Spring 2026

Some slides from Steve Tarzia, EECS-343, Northwestern

Midterm 1 on **February 10th, during class**

- Before Homework 2 is due! Plan your time well
- **In-class midterm**
- Allowed **1 sheet front and back** of notes
- Multiple-choice
- Will cover content from lectures and homeworks
- No constraints on the contents of the questions: can be knowledge recall, problem solving similar to what we've been doing in class, other types of questions content…etc.
- All the knowledge you need to solve the problems will come from lecture and homeworks
- Last year's Midterm 1 on Canvas (covers slightly more material)

Challenges in Concurrency

- **Race Conditions**: Multiple threads accessing/modifying shared data without proper synchronization.
- **Deadlocks**: Two or more threads wait forever for each other's locks.
- **Livelocks**: Threads continually react to each other without progressing.
- **Starvation**: Some threads never get enough CPU or resources to make progress.
- **Data Inconsistency**: Interleaving of operations leads to incorrect or partial updates.

How to Solve These Challenges

- **Synchronization Techniques**
  - **Mutexes (Mutual Exclusion Locks)**:
    - Ensures only one thread accesses critical section at a time.
  - **Semaphores**:
    - Can allow multiple threads up to a limit to access a shared resource.
  - **Spinlocks**:
    - Busy waiting locks suitable for short lock durations.
  - **Monitors / Condition Variables**:
    - High-level constructs for safe access and signaling between threads.
  - **Atomic Operations**:
    - Hardware-level support for indivisible updates to shared data.

Concurrency in Practice

- **Programming Languages**: C/C++: Pthreads, std::thread, std::mutex, and atomic library.
- Java: java.util.concurrent package (Executors, Locks, etc.).
- Python: threading, multiprocessing (noting GIL limitations).
- .NET (C#): Task, async/await, and Parallel classes.
- **Frameworks**: OpenMP: Compiler directives for shared-memory concurrency.
- MPI: Message-passing for distributed memory systems.

Concurrency Design Patterns

- **Producer-Consumer**: One or more producers generate data, consumers process it.
  - Typically uses a blocking queue or buffer.
- **Pipeline**: Data flows through multiple stages, each in its own thread.
- **Fork-Join**: Tasks split (fork) into sub-tasks and then (join) to merge results.
- **Map-Reduce**: Map phase transforms data; reduce phase aggregates results (common in big data).

Debugging and Testing Concurrent Programs

- **Logging**: Detailed timestamps and thread identifiers.
- **Tools**: Race condition detectors (e.g., ThreadSanitizer).
  - Performance profilers (to detect hotspots and contention).
- **Testing Strategies**: Stress testing and randomization.
  - Model checking or formal verification (for safety-critical systems).

Concurrency Best Practices

- **Minimize Shared State**: Use immutable objects or local variables whenever possible.
- **Favor High-Level Abstractions**: Let frameworks handle lower-level synchronization details (e.g. OpenDP).
- **Plan for Scalability**: Make sure concurrency solutions scale as cores/processors increase.
- **Design for Fault Tolerance**: Graceful recovery if part of the system fails or times out.

Concurrency can create tricky problems

```
#include <iostream>
#include <thread>
#include <atomic>

int counter = 0;
static const int LOOPS = 1e7;

void mythread(const char* arg) {
    std::cout << arg << ": begin\n";
    for (int i = 0; i < LOOPS; i++) {
        counter++; // Unsafe increment
    }
    std::cout << arg << ": done\n";
}
```

```
int main() {
    std::cout << "main: begin (counter = " << counter << ")\n";

    // Create two threads
    std::thread t1(mythread, "A");
    std::thread t2(mythread, "B");

    // Wait for threads to finish
    t1.join();
    t2.join();

    std::cout << "main: done with both (counter = " << counter
              << ", goal was " << 2 * LOOPS << ")\n";

    return 0;
}
```

- Start two threads, each of which increments a shared global **counter** variable 10^7 times.

Concurrency can create tricky problems

```
#include <iostream>
#include <thread>
#include <atomic>

int counter = 0;
static const int LOOPS = 1e7;

void mythread(const char* arg) {
    std::cout << arg << ": begin\n";
    for (int i = 0; i < LOOPS; i++) {
        counter++; // Unsafe increment
    }
    std::cout << arg << ": done\n";
}
```

```
int main() {
    std::cout << "main: begin (counter = " << counter << ")\n";

    // Create two threads
    std::thread t1(mythread, "A");
    std::thread t2(mythread, "B");

    // Wait for threads to finish
    t1.join();
    t2.join();

    std::cout << "main: done with both (counter = " << counter
              << ", goal was " << 2 * LOOPS << ")\n";

    return 0;
}
```

Output: main: done with both (counter = 10286823, goal was 20000000)

- Start two threads, each of which increments a shared global **counter** variable 10^7 times.

What's the problem?

- You have to understand the low-level behavior to find the problem.
  - In short, the "counter++" operation is not ***atomic***.

Output: main: done with both (counter = 10286823, goal was 20000000)

Incrementing a number in ASSEMBLY

- "counter++" has to:
  1. Copy from the memory location of the counter variable to a register
  2. Increment the register's value
  3. Copy from the register back to main memory
- Assuming that "counter" is in memory location 0x8049a1c:

```
mov 0x8049a1c, %eax
add $0x1, %eax
mov %eax, 0x8049a1c
```

- The scheduler can interrupt the thread before or after the "add"
  - This would cause both threads to ***read the same value***, increment it to the same value, and thus they would **repeat work**.

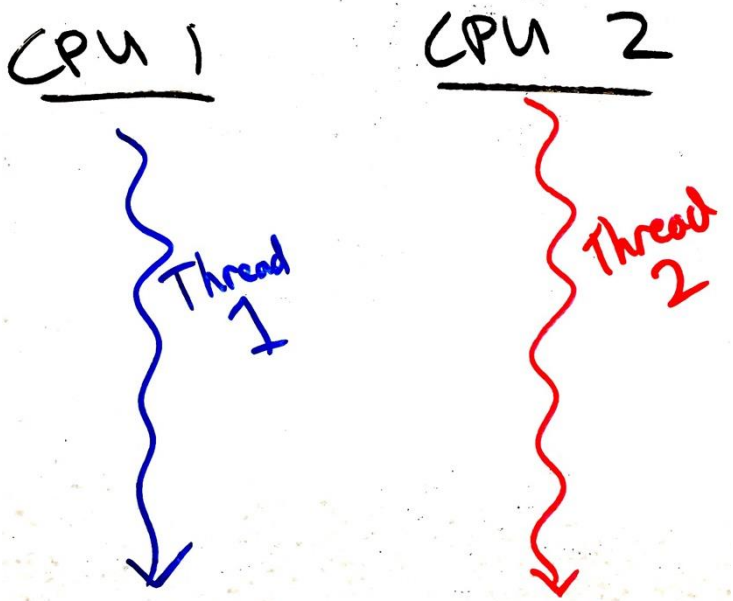
The increment failure in detail: 50 + 1 + 1 = 51!

OS	Thread 1	Thread 2	(after instruction) PC	%eax	counter
	before critical section		100	0	50
	mov 0x8049a1c, %eax		105	**50**	50
	add $0x1, %eax		108	**51**	50
interrupt					
save T1's state					
restore T2's state			100	0	50
		mov 0x8049a1c, %eax	105	**50**	50
		add $0x1, %eax	108	**51**	50
		mov %eax, 0x8049a1c	113	51	**51**
interrupt					
save T2's state					
restore T1's state			108	51	~~50~~
	mov %eax, 0x8049a1c		113	51	**51**

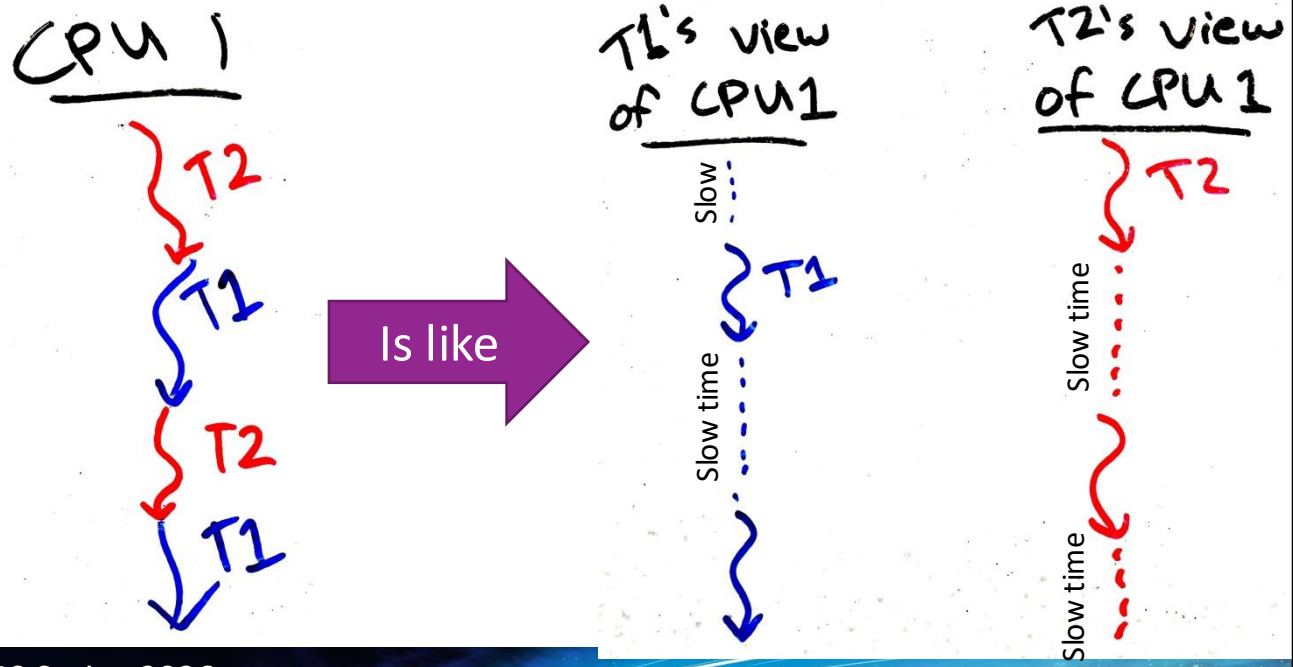
The process scheduler creates concurrency

- Even if only one CPU is present, threads operate “concurrently” because they are taking turns using the CPU.
- Each process thinks it has its own CPU that is sometimes very, very slow…

Obvious concurrency on two CPU cores:



Simulated concurrency on one CPU core:



Assume the scheduler is up to no good!

- Remember that processes have no control over the scheduler.
- So, to protect against concurrency bugs, we must assume that the scheduler can interrupt us at any time and schedule any other process.
- In other words, assume that the scheduler is ***adversarial***, and will do the worst possible scheduling.
- To prevent weird and rare concurrency bugs, your code should work correctly even when faced with an evil scheduler.

Concurrency Terminology

- ***Race condition***:
  - Two or more things are happening at the same time,
  - it's not clear which will finish first, and
  - the result will be different depending on which finishes first.
- ***Indeterminate***: Output can be different each time (not ***deterministic***).
- ***Critical section***:
  - Code that accesses a shared resource and must not be executed concurrently.
  - In other words, code that would lead to a race condition.
  - Sometimes called a ***transaction***, especially in database systems.
  - We must execute critical sections ***atomically***, meaning that it cannot be *partially* executed. Atomic means it cannot be divided, or is executed "all or none."
- ***Mutual exclusion primitives*** are used to protect critical sections.
- ***Locks*** are the simplest kind of mutual exclusion primitive.

Critical sections

- Critical sections often involve modification of multiple related data
  - While the modifications are happening there is some inconsistency
  - The inconsistency is eventually resolved before leaving the critical section
- For example:
  - Inserting an element in the middle of a linked list
    - Two pointers must change. List is broken if just one is changed.
  - Swapping two values.
- Don’t have to worry about critical sections if:
  - Operation is just one assembly instruction (CPU executes these atomically), or
  - Program is single-threaded or the particular data is not shared among threads

Critical Sections Example

```
#include <iostream>
#include <thread>
#include <mutex>

/* Shared data */
int counter = 0;
std::mutex m;

/* Thread function: increments the shared counter */
void increment() {
    for (int i = 0; i < 1000; ++i) {
        std::lock_guard<std::mutex> lock(m);
    // --- CRITICAL SECTION ---
    ++counter;
    // --- CRITICAL SECTION ---
}}
```

```
int main() {
    // Create two threads that run 'increment'
    std::thread t1(increment);
    std::thread t2(increment);

    // Wait for both threads to finish
    t1.join();
    t2.join();

    // Print the final counter value
    std::cout << "Final counter: " << counter << std::endl;

    return 0;
}
```

Critical Sections Example

```
#include <iostream>
#include <thread>
#include <mutex>

/* Shared data */
int counter = 0;
std::mutex m;

/* Thread function: increments the shared counter */
void increment() {
 for (int i = 0; i < 1000; ++i) {
       std::lock_guard<std::mutex> lock(m);
   // --- CRITICAL SECTION ---
   ++counter;
   // --- CRITICAL SECTION ---
}}
```

Mutex: m. Prevents multiple threads from executing the critical section simultaneously.

```
int main() {
   // Create two threads that run 'increment'
   std::thread t1(increment);
   std::thread t2(increment);

   // Wait for both threads to finish
   t1.join();
   t2.join();

   // Print the final counter value
   std::cout << "Final counter: " << counter << std::endl;

   return 0;
}
```

Critical Sections Example

```
#include <iostream>
#include <thread>
#include <mutex>

/* Shared data */
int counter = 0;
std::mutex m;

/* Thread function: increments the shared counter */
void increment() {
 for (int i = 0; i < 1000; ++i) {
       std::lock_guard<std::mutex> lock(m);
   // --- CRITICAL SECTION ---
   ++counter;
   // --- CRITICAL SECTION ---
}}
```

Mutex: m. Prevents multiple threads from executing the critical section simultaneously.

```
int main() {
   // Create two threads that run 'increment'
   std::thread t1(increment);
   std::thread t2(increment);

   // Wait for both threads to finish
   t1.join();
   t2.join();

   // Print the final counter value
   std::cout << "Final counter: " << counter << std::endl;

   return 0;
}
```

Critical Sections Example

```
#include <iostream>
#include <thread>
#include <mutex>

/* Shared data */
int counter = 0;
std::mutex m;

/* Thread function: increments the shared counter */
void increment() {
 for (int i = 0; i < 1000; ++i) {
   std::lock_guard<std::mutex> lock(m);
   // --- CRITICAL SECTION ---
   ++counter;
   // --- CRITICAL SECTION ---
}}
```

std::lock_guard<std::mutex> automatically **locks the mutex upon construction** and unlocks it when the lock guard goes out of scope, ensuring mutual exclusion on counter.

```
int main() {
   // Create two threads that run 'increment'
   std::thread t1(increment);
   std::thread t2(increment);

   // Wait for both threads to finish
   t1.join();
   t2.join();

   // Print the final counter value
   std::cout << "Final counter: " << counter << std::endl;

   return 0;
}
```

Locking in a Single CPU

- On a single CPU, OS typically switches between different threads by using **interrupts**
- **No interrupts = no preemption** of tasks
  - Must wait for current code to finish running
- **Ensures atomicity and mutual exclusion**
- **Kernel-level code**: OS systems disable interrupts (briefly) when performing important system processes (process queues, scheduler structures, critical kernel data)

Disadvantages of Locking in a Single CPU

- **Loss of Responsiveness**:
  - CPU **cannot respond to *any*** interrupts—including potentially important ones like hardware interrupts or high-priority timer event
- **Limited Scalability**:
  - On **multi-core** architectures, disabling interrupts on one core does not automatically prevent another core from running
- **Increased Complexity**:
  - If interrupts remain disabled for too long, timing deadlines may be missed and the system might seem “frozen”

Disadvantages of Locking in a Single CPU

- **Loss o**
  - CPU
    pote
    prio
- **Limite**
  - On
    doe
- **Increa**
  - If in
    may be missed and the system might seem "frozen"

Gives too much power to **user processes**!

With interrupts disabled, **kernel has no ability to preempt user processes**!

Disadvantages of Locking in a Single CPU

- **Loss o**
  - CPU
    pote
    prio
- **Limite**
  - On
    doe
- **Increa**
  - If in
    may

Gives too much power to **user processes**!

With interrupts disabled, **kernel has no ability to preempt user processes**!

Not a good locking strategy!

Two different metaphors: Lock

- **Lock**
  - A lock is something that's designed to block access.
  - Our virtual lock works as follows:
    - Anyone can **lock** or **unlock** (there is no "key").
    - Trying to lock an already-locked lock will cause you to wait until it's unlocked.
  - The "lock" is a confusing metaphor sometimes.

Two different metaphors: Token

- Token
  - Holding the token gives you permission to do something.
  - There is only one token.
  - Thus, you:
    1. Try to **acquire** the token (“lock”). You have to wait your turn if someone else is holding it.
    2. When done, **release** the token/lock.
  - The token represents exclusive access to a shared resource or a critical section.

Types of Locks

Here are some types of locks that we can use in our program

std::mutex: Your Simplest Lock

```
#include <mutex>
#include <thread>
#include <iostream>
#include <vector>

static std::mutex mtx;
static int sharedCounter = 0;

void worker(int threadId) {
    for (int i = 0; i < 1000; ++i) {
        std::lock_guard<std::mutex> lock(mtx);
        ++sharedCounter;
    }
}
```

```
int main() {
    std::vector<std::thread> threads;
    threads.reserve(4);

    for (int i = 0; i < 4; ++i) {
        threads.emplace_back(worker, i);
    }

    for (auto &t : threads) {
        t.join();
    }

    std::cout << sharedCounter << std::endl;
    return 0;
}
```

Advantages of mutex

- **Simplicity and Familiarity**
  - Conceptually straightforward: lock the mutex before accessing shared data; unlock it when done.
- **Portability**
  - Mutexes are part of high-level threading libraries
- **Well-Understood Semantics**
  - Clear specifications regarding locking and unlocking behavior, so it's easier to reason about program correctness
- **Fairness and Priority Inheritance** (Implementation-Dependent)
  - Many mutex implementations (especially in real-time operating systems) provide fairness or priority inheritance mechanisms

Disadvantages of mutex

- **Potential for Deadlocks**
  - If multiple locks are acquired out of order or forget to unlock, you can create a deadlock
- **Possible Performance Overhead**
  - Entering and leaving the kernel (for blocking, scheduling, etc.) adds overhead.
  - If a lock is highly contended, threads will frequently block and unblock
- **Blocking**
  - When one thread holds the lock, all other threads that need it must go to sleep (block)
- **Coarse-Grained Locking**
  - Mutexes encourage a “big lock” approach (one global lock for large portions of code). This can serialize more than needed, limiting parallelism.

Other Locking Strategies

- Using **a lock flag**
- Each thread **first calls lock()**, then **performs their computation**, then calls **unlock**
- What can go wrong here?
- Two or more threads **simultaneously call lock** and copy value of locked into registers

This **tests** and **sets** the variable locked to perform computations

But they are not atomic!

```
// Globa
int locked = 0;

// Naive lock function void lock() {
    while (locked); // While busy, wait
    locked = 1;
}

// Naive unlock function
void unlock() {
   locked = 0;
}
```

CPU hardware support for concurrency

- Atomic ***test-and-set*** instruction (e.g. x86)
  - Called “atomic exchange”
  - `lock; xchg [reg], [mem]` on x86
  - Atomically swaps contents of `[reg]` with value at memory location `[mem]`
- Why is it atomic?
  - On x86, `lock` tells the processor to lock the cache line that contains the target memory address; hardware mechanism
- Used in **spinlocks**

Spinlock with test-and-set

```
#include <atomic>
#include <thread>
#include <iostream>
#include <vector>

class Spinlock {
private:
    std::atomic_flag locked = ATOMIC_FLAG_INIT;

public:
    void lock() {
        while (locked.test_and_set(std::memory_order_acquire)) {}
    }

    void unlock() {
        locked.clear(std::memory_order_release);
    }
};
```

```
static Spinlock spinlock;
static int sharedCounter = 0;

void worker(int threadId) {
    for (int i = 0; i < 1000; ++i) {
        spinlock.lock();
        ++sharedCounter;
        spinlock.unlock();
    }
}

int main() {
    std::vector<std::thread> threads;
    for (int i = 0; i < 4; ++i) {
        threads.emplace_back(worker, i);
    }

    for (auto& t : threads) {
        t.join();
    }

    std::cout << sharedCounter <<    std::endl;
    return 0;
}
```

Spinlock with test-and-set

What are some disadvantages of spinlocks? (i.e. why is it called a spinlock?)

```
#include <atomic>
#include <thread>
#include <iostream>
#include <vector>

class Spinlock {
private:
    std::atomic_flag locked = ATOMIC_FLAG_INIT;

public:
    void lock() {
        while (locked.test_and_set(std::memory_order_acquire)) {}
    }

    void unlock() {
        locked.clear(std::memory_order_release);
    }
};
```

```
static Spinlock spinlock;
static int sharedCounter = 0;

void worker(int threadId) {
    for (int i = 0; i < 1000; ++i) {
        spinlock.lock();
        ++sharedCounter;
        spinlock.unlock();
    }
}

int main() {
    std::vector<std::thread> threads;
    for (int i = 0; i < 4; ++i) {
        threads.emplace_back(worker, i);
    }

    for (auto& t : threads) {
        t.join();
    }

    std::cout << sharedCounter <<    std::endl;
    return 0;
}
```

Spinlock with test-and-set

```
#include <atomic>
#include <thread>
#include <iostream>
#include <vector>

class Spinlock {
private:
    std::atomic_flag lo

public:
    void lock() {
        while (locked.test_and_set(std::memory_order_acquire)) {}
    }

    void unlock() {
        locked.clear(std::memory_order_release);
    }
};
```

Takes up CPU cycles! (spins…) Good for very short critical sections without (a lot of) contention

```
static Spinlock spinlock;
static int sharedCounter = 0;

void worker(int threadId) {
    1000; ++i) {
    ();
    unter;
    ock();

    ::thread> threads;
    for (int i = 0; i < 4; ++i) {
        threads.emplace_back(worker, i);
    }

    for (auto& t : threads) {
        t.join();
    }

    std::cout << sharedCounter <<    std::endl;
    return 0;
}
```

Spinlock with test-and-set

```
#include <atomic>
#include <thread>
#include <iostream>
#include <vector>

class Spinlock {
private:
    std::atomic_flag lo

public:
    void lock() {
        while (locked.te
    }

    void unlock() {
        locked.clear(std::memory_order_release);
    }
};
```

```
static Spinlock spinlock;
static int sharedCounter = 0;

void worker(int threadId) {
                                  00; ++i) {
                                  er;
                                  ();
                                  hread> threads;
                                  ++i) {
                                  e_back(worker, i);
                                  ads) {
    }

    std::cout << sharedCounter <<    std::endl;
    return 0;
}
```

Takes up CPU cycles! (spins…) Good for very short critical sections without (a lot of) contention

Doesn’t put thread to sleep

Generally use this infrequently