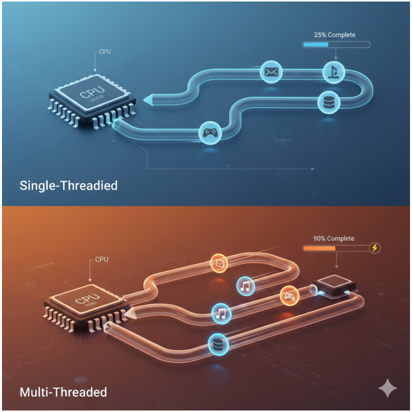


Image Source: Gemini

CSL 301

OPERATING SYSTEMS

Lecture 16

Concurrency Intro

Threads

Instructor

Dr. Dhiman Saha

Recap: The Process

- The OS gives us the abstraction of a **process**.

Recap: The Process

- The OS gives us the abstraction of a **process**.
- A process has two key components:
  - **Virtual CPUs**: The illusion that the program is running on its own CPU.
  - **Virtual Memory**: The illusion that the program has its own private address space.

Recap: The Process

- The OS gives us the abstraction of a **process**.
- A process has two key components:
  - **Virtual CPUs**: The illusion that the program is running on its own CPU.
  - **Virtual Memory**: The illusion that the program has its own private address space.
- This "classic" view of a process has a **single point of execution**.
  - One Program Counter (PC).
  - One set of instructions being executed.

A New Abstraction: The Thread

- A **multi-threaded** program has *more than one* point of execution.
- Think of it as multiple PCs, each fetching and executing instructions.
- It's like having multiple processes, with one key difference...

A New Abstraction: The Thread

- A **multi-threaded** program has *more than one* point of execution.
- Think of it as multiple PCs, each fetching and executing instructions.
- It's like having multiple processes, with one key difference...

The Key Difference

All threads within a single process **share the same address space**. They can all access the same data.

Thread vs. Process State

What's shared?

- Address Space (Code, Heap)
- Page Table
- File Descriptors

What's private?

- Program Counter (PC)
- Registers
- **Stack**

Thread vs. Process State

What's shared?

- Address Space (Code, Heap)
- Page Table
- File Descriptors

What's private?

- Program Counter (PC)
- Registers
- **Stack**

- Switching between threads of the same process is much cheaper than switching between processes. Why?

Thread vs. Process State

What's shared?

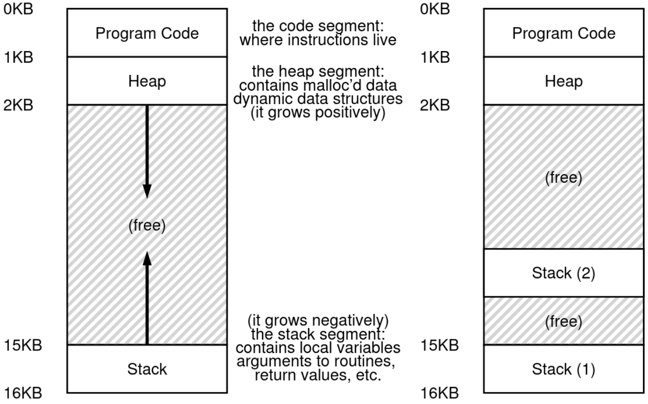
- Address Space (Code, Heap)
- Page Table
- File Descriptors

What's private?

- Program Counter (PC)
- Registers
- **Stack**

- Switching between threads of the same process is much cheaper than switching between processes. Why?
- *No need to switch the address space!*

Visualizing Address Spaces - Single Vs Multi-Threaded



- In a multi-threaded process, there is one stack **per thread**.
- This means local variables and function call arguments are private to each thread (this is called *thread-local storage*).

Reason 1: Parallelism

- Imagine a program performing an operation on a very large array.
- On a system with multiple CPUs, you can speed this up by dividing the work.

Reason 1: Parallelism

- Imagine a program performing an operation on a very large array.
- On a system with multiple CPUs, you can speed this up by dividing the work.
- **Parallelization**: The task of transforming a single-threaded program into one that does work on multiple CPUs.
- Using one thread per CPU is a natural way to make programs run faster on modern hardware.

Reason 2: Overlapping I/O

- Many programs perform slow I/O operations (e.g., reading from a disk, waiting for a network request).
- In a single-threaded program, the entire process **blocks** and can't do any other work.

Reason 2: Overlapping I/O

- Many programs perform slow I/O operations (e.g., reading from a disk, waiting for a network request).
- In a single-threaded program, the entire process **blocks** and can't do any other work.
- With threads, you can avoid getting stuck!
  - While one thread is blocked waiting for I/O...
  - ...the OS scheduler can switch to another thread, which is ready to run and do useful work.

Reason 2: Overlapping I/O

- Many programs perform slow I/O operations (e.g., reading from a disk, waiting for a network request).
- In a single-threaded program, the entire process **blocks** and can't do any other work.
- With threads, you can avoid getting stuck!
  - While one thread is blocked waiting for I/O...
  - ...the OS scheduler can switch to another thread, which is ready to run and do useful work.
- This is essential for modern servers (web servers, databases) that handle many concurrent requests.

Example: Creating Threads (t0.c)

```c
#include <stdio.h>
#include <pthread.h>

void *mythread(void *arg) {
    printf("%s\n", (char *) arg);
    return NULL;
}

int main(int argc, char *argv[]) {
    pthread_t p1, p2;
    printf("main: begin\n");
    pthread_create(&p1, NULL, mythread, "A");
    pthread_create(&p2, NULL, mythread, "B");
    // join waits for the threads to finish
    pthread_join(p1, NULL);
    pthread_join(p2, NULL);
    printf("main: end\n");
    return 0;
}
```

Understanding Thread Execution

- The main thread creates two new threads, T1 and T2.
- It then calls `pthread_join()` twice, waiting for each thread to complete.
- What will the output be?

Understanding Thread Execution

- The main thread creates two new threads, T1 and T2.
- It then calls `pthread_join()` twice, waiting for each thread to complete.
- What will the output be?

Trace 1

```
main: begin
A
B
main: end
```

Trace 2

```
main: begin
B
A
main: end
```

Trace 3

```
main: begin
main: end
A
B
```

Understanding Thread Execution

- The main thread creates two new threads, T1 and T2.
- It then calls `pthread_join()` twice, waiting for each thread to complete.
- What will the output be?

Trace 1

```
main: begin
A
B
main: end
```

Trace 2

```
main: begin
B
A
main: end
```

Trace 3

```
main: begin
main: end
A
B
```

The Point

The order of execution is non-deterministic! It depends on the OS scheduler. Any of these outputs (and more) are possible.

What if Threads Interact?

- The previous example was simple: the threads didn't interact.
- Things get much more complicated when threads access **shared data**.
- Let's look at an example where two threads try to update a shared counter.

Example: A Shared Counter (t1.c) I

```c
#include <stdio.h>
#include <pthread.h>

static volatile int counter = 0;

void *mythread(void *arg) {
    printf("%s: begin\n", (char *) arg);
    int i;
    for (i = 0; i < 1e7; i++) {
        counter = counter + 1;
    }
    printf("%s: done\n", (char *) arg);
    return NULL;
}
```

Example: A Shared Counter (t1.c) II

```c
int main(int argc, char *argv[]) {
    pthread_t p1, p2;
    pthread_create(&p1, NULL, mythread, "A");
    pthread_create(&p2, NULL, mythread, "B");
    pthread_join(p1, NULL);
    pthread_join(p2, NULL);
    printf("main: done with both (counter = %d)\n", counter);
    return 0;
}
```

What Should The Result Be?

- Two threads, each incrementing the counter 10,000,000 times.
- Expected final value: **20,000,000**.

What Should The Result Be?

- Two threads, each incrementing the counter 10,000,000 times.
- Expected final value: **20,000,000**.
- But when we run it...

```
prompt> ./main
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 19345221)

prompt> ./main
main: done with both (counter = 19221041)
```

Problem!

Not only is the result wrong, it's **different** every time! Why?

The "Atomic" Illusion

- The C statement `counter = counter + 1;` seems simple.
- But to the CPU, it's not a single, **atomic** operation.
- It's actually a sequence of instructions.

The "Atomic" Illusion

- The C statement `counter = counter + 1;` seems simple.
- But to the CPU, it's not a single, **atomic** operation.
- It's actually a sequence of instructions.

x86 Assembly for 'counter++'

```
mov 0x8049a1c, %eax      ; Load counter's value
add $0x1, %eax           ; Add 1 to register
mov %eax, 0x8049a1c      ; Store new value
```

The "Atomic" Illusion

- The C statement `counter = counter + 1;` seems simple.
- But to the CPU, it's not a single, **atomic** operation.
- It's actually a sequence of instructions.

x86 Assembly for ‘counter++‘

```
mov 0x8049a1c, %eax      ; Load counter's value
add $0x1, %eax           ; Add 1 to register
mov %eax, 0x8049a1c      ; Store new value
```

The Core Problem

A context switch can happen **between any** of these instructions!

Anatomy of a Race Condition

Let's say `counter` is 50.

- **Thread 1** executes `mov`, loading 50 into its private register (eax=50).

Anatomy of a Race Condition

Let's say `counter` is 50.

- **Thread 1** executes `mov`, loading 50 into its private register (eax=50).
- **Thread 1** executes `add`, incrementing its register (eax=51).

The Result

An increment has been lost! We performed two increments, but the value only increased by one.

Anatomy of a Race Condition

Let's say `counter` is 50.

- **Thread 1** executes `mov`, loading 50 into its private register (eax=50).
- **Thread 1** executes `add`, incrementing its register (eax=51).
- **TIMER INTERRUPT!** The OS context switches to Thread 2.

The Result

An increment has been lost! We performed two increments, but the value only increased by one.

Anatomy of a Race Condition

Let's say `counter` is 50.

- **Thread 1** executes `mov`, loading 50 into its private register (eax=50).
- **Thread 1** executes `add`, incrementing its register (eax=51).
- **TIMER INTERRUPT!** The OS context switches to Thread 2.
- **Thread 2** executes `mov`, loading the *original* value 50 from memory into its private register (eax=50).

The Result

An increment has been lost! We performed two increments, but the value only increased by one.

Anatomy of a Race Condition

Let's say `counter` is 50.

- **Thread 1** executes `mov`, loading 50 into its private register (eax=50).
- **Thread 1** executes `add`, incrementing its register (eax=51).
- **TIMER INTERRUPT!** The OS context switches to Thread 2.
- **Thread 2** executes `mov`, loading the *original* value 50 from memory into its private register (eax=50).
- **Thread 2** executes `add` (eax=51) and then `mov`, storing 51 back to memory. `counter` is now 51.

The Result

An increment has been lost! We performed two increments, but the value only increased by one.

Anatomy of a Race Condition

Let's say `counter` is 50.

- **Thread 1** executes `mov`, loading 50 into its private register (eax=50).
- **Thread 1** executes `add`, incrementing its register (eax=51).
- **TIMER INTERRUPT!** The OS context switches to Thread 2.
- **Thread 2** executes `mov`, loading the *original* value 50 from memory into its private register (eax=50).
- **Thread 2** executes `add` (eax=51) and then `mov`, storing 51 back to memory. `counter` is now 51.
- **CONTEXT SWITCH!** The OS switches back to Thread 1.

The Result

An increment has been lost! We performed two increments, but the value only increased by one.

Anatomy of a Race Condition

Let's say `counter` is 50.

- **Thread 1** executes `mov`, loading 50 into its private register (eax=50).
- **Thread 1** executes `add`, incrementing its register (eax=51).
- **TIMER INTERRUPT!** The OS context switches to Thread 2.
- **Thread 2** executes `mov`, loading the *original* value 50 from memory into its private register (eax=50).
- **Thread 2** executes `add` (eax=51) and then `mov`, storing 51 back to memory. `counter` is now 51.
- **CONTEXT SWITCH!** The OS switches back to Thread 1.
- **Thread 1** resumes. It executes its final instruction, `mov`, storing its register value (51) back to memory. `counter` is still 51.

The Result

An increment has been lost! We performed two increments, but the value only increased by one.

Key Concurrency Terms

Race Condition The result of a program depends on the timing or interleaving of operations. The situation we just saw.

Key Concurrency Terms

Race Condition The result of a program depends on the timing or interleaving of operations. The situation we just saw.

Critical Section A piece of code that accesses a shared resource (like our counter) and must not be concurrently executed by more than one thread.

Key Concurrency Terms

Race Condition The result of a program depends on the timing or interleaving of operations. The situation we just saw.

Critical Section A piece of code that accesses a shared resource (like our counter) and must not be concurrently executed by more than one thread.

Mutual Exclusion The property we want to enforce. It guarantees that if one thread is executing in a critical section, other threads will be prevented from entering it.

The Wish for Atomicity

- If we had a single hardware instruction that did the load, add, and store all at once, there would be no problem.

```
memory-add 0x8049a1c, $0x1
```

- This is an **atomic** operation: it runs "as a unit", cannot be interrupted, and appears to happen instantaneously.

The Wish for Atomicity

- If we had a single hardware instruction that did the load, add, and store all at once, there would be no problem.

```
memory-add 0x8049a1c, $0x1
```

- This is an **atomic** operation: it runs "as a unit", cannot be interrupted, and appears to happen instantaneously.
- But hardware can't provide atomic instructions for every complex operation we might want (e.g., "atomically update B-Tree").

The Real Solution: Synchronization

- Instead of asking for complex atomic instructions...
- We ask the hardware for a few simple, useful atomic instructions.
- We then use these, with help from the OS, to build higher-level **synchronization primitives**.
- Examples: Locks (Mutexes), Semaphores, Condition Variables.

The Real Solution: Synchronization

- Instead of asking for complex atomic instructions...
- We ask the hardware for a few simple, useful atomic instructions.
- We then use these, with help from the OS, to build higher-level **synchronization primitives**.
- Examples: Locks (Mutexes), Semaphores, Condition Variables.
- By using these primitives, we can protect our critical sections and ensure mutual exclusion.

The Goal

To build multi-threaded code that is correct, reliable, and produces deterministic results despite the challenging nature of concurrency.