Threads

C++ for Developers

lafftale
for developers

Content

- What is a thread?
- Threads in C++
- Challenges
- Solutions

What is a thread?

C++ for Developers

CPU

- The brain of the computer
- Responsible for executing machine instructions

A CPU can consist of several **Cores** that executes instructions.

What is an instruction?

Every CPU architecture have a set of machine instructions that the CPU can execute. These instructions can be:

- Data transfer
- Arithmetic / Logic
- Control flow
- System calls

The instructions syntax and machine code differs. This is why you need to compile your C++ programs differently depending on what environment they run in.

ARM vs x86 ASM

```
    .section .text
    .global _start

_start:
    mov     r0, #1
    ldr     r1, =msg
    ldr     r2, =len
    mov     r7, #4
    svc     #0

    @ exit(0)
    mov     r0, #0
    mov     r7, #1
    svc     #0

    .section .data
msg:
    .ascii  "Hello world!"
len = . - msg
```

```
section        .text
global         _start
_start:
    mov edx, len
    mov ecx, msg
    mov ebx, 1
    mov eax, 4
    int 0x80
    mov eax, 1
    int 0x80

section        .data
    msg        db "Hello world!", 0xa
    len        equ $ -msg
```

These not only differ in syntax, but on a binary level.

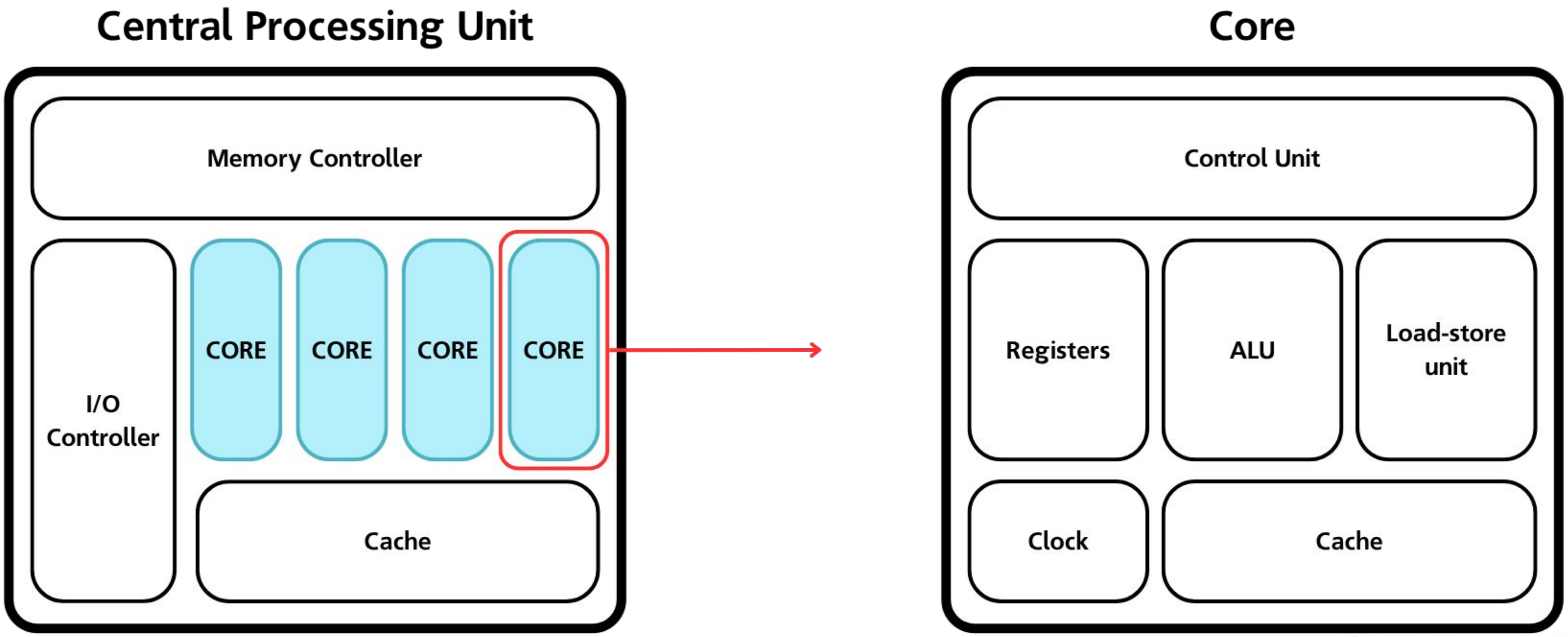
Example

Intel primarily uses the x86 architecture for their CPU's

Apple mainly uses the ARM architecture for their CPU's

This is why you need to choose which architecture you are using sometimes when downloading programs.

CPU architecture



lafftale for developers

Execution of instructions

Each component in the core can only execute one instruction at the time.

Pipeline
By pipelining, the goal is to make sure every component is executing an instruction each clock cycle.

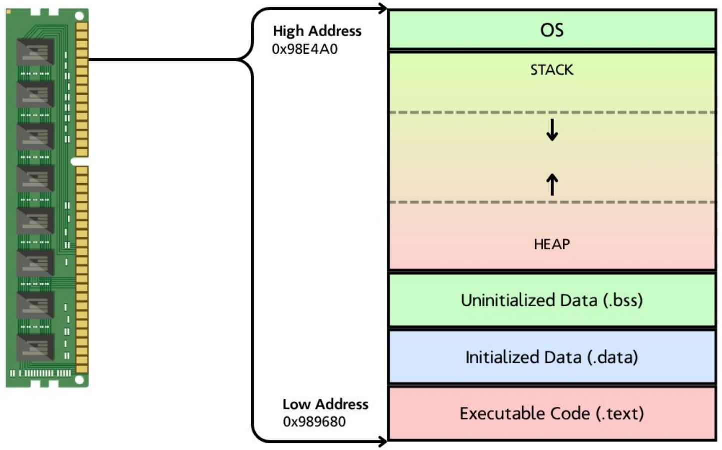
Branch prediction
Good *branch predictions* can help the CPU perform instructions more efficiently by not filling the pipeline with irrelevant *predicted* instructions.

Pipeline example

Clock Cycle	Fetch	Decode	Execute
1	mov edx, len	-	-
2	mov ecx, msg	mov edx, len	-
3	mov ebx, 1	mov ecx, msg	mov edx len

Process

A process is a program in execution. You might remember this picture?



This is a part of what you call a process!

Process

Besides the memory space of our program - a process also includes:

- Resources (sockets, files, environment variables)
- At least one *thread* to execute instructions

Now… What is a thread?

A thread is a subset of a process. The thread is the smallest sequence of instructions the scheduler can handle.

Every process runs at least one (*main)* thread, but can run several additional threads concurrently.

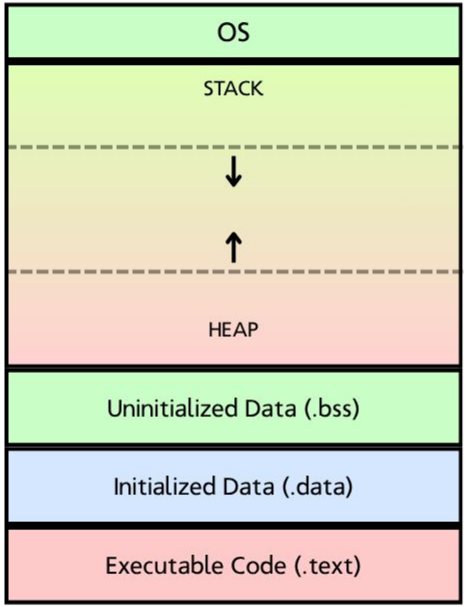
You can think of a thread as the stream of instructions from a process that the scheduler ship for execution.

Difference between process and thread

A process is an independent container that holds a program’s code, data and system resources.

A thread is the smallest unit of execution within a process. It carries the program counter, stack, and registers needed to execute code.

Process



Thread

Stack Space:
Data specific to the thread.

Register Set:
Hold temporary data and intermediate results for the thread's execution.

Program Counter:
Tracks the current instruction being executed by the thread.

CPU

Executes the instructions from the thread

Multithreading

A process can have several threads running at the same time. All threads belonging to the same process share code section, data section and OS resources.

This means that one thread can affect another.

Scheduler

A scheduler is part of an operating system responsible for selecting which thread should run on the CPU at a given time.

Preemptive
The process can be interrupted, in which case its context is saved and the scheduler chooses the next ready thread to be executed.

Non-preemptive
The thread being executed can’t be interrupted until itself blocks, finishes or yields control.

Parallelism

With a CPU that houses several cores, it is possible to achieve multiple threads executing *truly* at the same time.

This is called *Parallelism*.

Time slicing

Time slicing is an operating system scheduling technique that creates the illusion of *parallelism*.

The scheduler gives each thread a small time slice to run before switching to the next one.

By switching rapidly between threads, it appears as though they are running in parallel, even though only one executes at a time.

What will be used?

It depends on the hardware and the operating system.

- If the CPU has only one core → the OS uses time slicing to create the illusion of parallel execution.

- If the CPU has multiple cores → the OS can use true parallelism and time slicing.

On modern systems, the operating system's scheduler decides how threads are distributed across cores and when to switch between them.

Multithreading in C++

C++ for Developers

Threads in C++

Threads in C++ are defined in <thread>

To start a thread you can use the following syntax:

```
std::thread thread_one(print_symbol, 'H', times);
```

std::thread thread_name(function_name, variable1, variable2 …);

Threads in C++

When we create a thread object, we do two things:

- We instantiate a thread object
- Create a thread that runs our function

When first created - the std::thread *thread_name* owns the thread.

Object ownership

When a thread object run out of scope, we lose track of the thread - and have no way of accessing it again. Therefore, C++ have a safeguard which forces us to explicitly inform what to do:

We have two choices:

- Wait for the thread to finish, only then can the program finish.
- Detach the thread from the std::thread object

If we haven't told what to do, the program will call std::terminate()

thread_name.join()

This will pause execution in the current thread, until the *thread_name.join()* has finished its execution. When joining a thread we can expect the following:

- The thread has finished executing and is no longer relevant for the OS.
- The thread object is now not joinable - meaning it is no longer a thread in executing.
- The thread object is now safe to delete (or run out of scope).

thread_name.detach()

This will separate the actual thread in execution from the thread object in our code. We can expect the following:

- We lose control over the thread.
- The thread will keep going after the main thread is finished.
- The OS will clean up after the thread finish executing.

A detached thread is often handled as “fire and forget”.

detach() vs join()

The .join() is absolutely most commonly used over all. But .detach() finds it place sometimes.

If you have functionality that has zero risk of multithreading problems (like deadlock) and doesn’t apply any waits - it could be detached as well.

For example: Sending a notification.

Let’s look at some code!

Clafftale for developers

Challenges when multithreading

C++ for Developers

Clafftale for developers

What are some challenges that we face in multithreading?

- Race condition
- Deadlocks
- Complexity in debugging
- Performance overhead

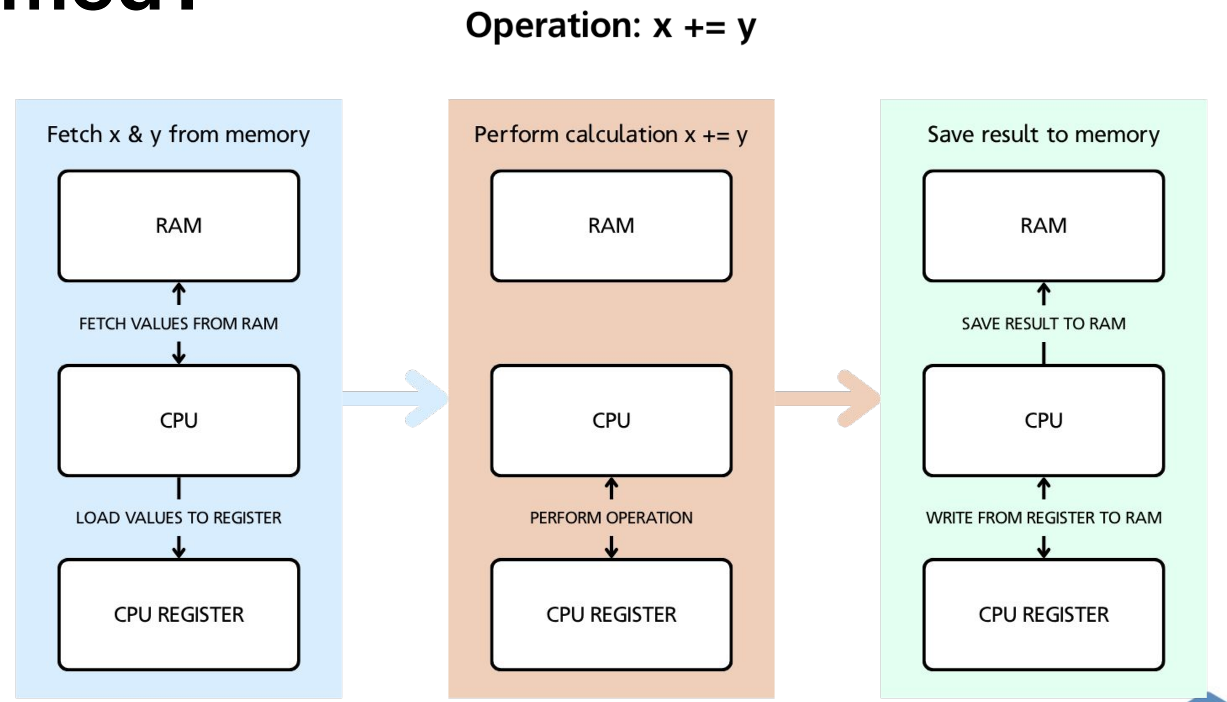
Race conditions

This occurs when two or more threads are trying to modify the same variable.

If it's not properly synchronized, you will get unexpected results.

Why is that?

How are operations performed?



Deadlocks

Deadlocks occurs when two threads are waiting for each other indefinitely.

This can happen when:

- Thread 1 is waiting for Thread 2 to free Resource A.
- Thread 2 is waiting for Thread 1 to free Resource A.

Both are waiting to be notified, but since none of them are executing - neither of them will notify the other.

Complexity

Debugging multi-threaded software can be very complex.

- Thread execution is non-deterministic, meaning your program can execute differently every time.
- Because of this, bugs may only appear sometimes - making it harder to debug.

Performance overhead

Too many threads executing at the same time can cause the overall program to run slower, because all threads take longer to execute.

Solutions

C++ for Developers

lafftale
for developers

Solutions

- Synchronization
- Mutex
- Semaphores
- Atomic
- Condition Variables
- Thread planning

Synchronization

Synchronization means coordinating the actions of multiple threads so that shared data is accessed in a safe and predictable way.

Mutex (std::mutex)

A lock or mutex (short for mutual exclusion) enforces concurrency control by ensuring that only *one thread at a time* can access or modify a shared resource or section of code.

- .lock() - takes the resource. No other thread can access the mutex area.
- .unlock() - release the resource. Another thread can now access the mutex area.

lafftale for developers

Mutex analogy: The Locked Room

The synchronized block of code can be thought of as a locked room, which only has one key to unlock it.

When a thread .lock() - it takes the key and enter the room (the block of code). While the thread is in there with the key, no other thread can enter.

When the thread is done .unlock() - it leaves the key and another thread can take it.

Semaphores (std::counting_semaphore<N>)

This is a less strict type of lock, that can allow several threads to use the same resource. This is a software *integer* which decides how many threads that can access the resource at the same time.

std::counting_semaphore<4> sem(4);

- sem.acquire() | takes a resources - counter--;
- sem.release() | release the resource - counter++;

This would allow up to 4 resources to use the resource.

Atomic (std::atomic)

Atomic operations are operations that are performed completely or not at all. They cannot be interrupted by other threads.

This ensures that only one thread can perform that operation on a particular variable at a time.

```
std::atomic<int> counter = 0;
counter++;
```

A thread can’t read or write to counter while it is being used by another thread

Condition variables (std::condition_variable)

With these, we can create signaling between threads waiting for certain conditions to be met.

In this case, we acquire the lock - check if the condition is true or false.

REMEMBER - Always start single-threaded.

First, write and test a correct single-threaded version of the program.

Then identify performance bottlenecks and multithread only the hot spots.

lafftale
for developers