Multithreaded Programming and Parallel Programming in C/C++

In many applications today, software needs to make decisions quickly. And the best way to do that is through parallel programming in C/C++ and multithreading (multithreaded programming).

Here we explain what is parallel programming, multithreading (multithreaded programming), concurrent vs parallel, and how to avoid parallel programming C/C++ defects.

What Is Parallel Programming?

> *Parallel programming is the process of using a set of resources to solve a problem in less time by dividing the work.*

Using parallel programming in C is important to increase the performance of the software.

Concurrent vs Parallel: How Does Parallel Programming Differ From Multithreaded Programming?

Parallel programming is a broad concept. It can describe many types of processes running on the same machine or on different machines.

Multithreading specifically refers to the concurrent execution of more than one sequential set (thread) of instructions.

> *Multithreaded programming is programming multiple, concurrent execution threads. These threads could run on a single processor. Or there could be multiple threads running on multiple processor cores.*

Concurrent vs Parallel: Multithreaded Programming on a Single Processor

Multithreading on a single processor gives the illusion of running in parallel. In reality, the processor is switching by using a scheduling algorithm. Or, it's switching based on a combination of external inputs (interrupts) and how the threads have been prioritized.

Concurrent vs Parallel: Multithreaded Programming on Multiple Processors

Multithreading on multiple processor cores is truly parallel. Individual microprocessors work together to achieve the result more efficiently. There are multiple parallel, concurrent tasks happening at once.

Why Is Multithreaded Programming Important?

Multithreading is important to development teams today. And it will remain important as technology evolves.

Here's why:

Processors Are at Maximum Clock Speed

Processors have reached maximum clock speed. The only way to get more out of CPUs is with parallelism.

Multithreading allows a single processor to spawn multiple, concurrent threads. Each thread runs its own sequence of instructions. They all access the same shared memory space and communicate with each other if necessary. The threads can be carefully managed to optimize performance.

Parallelism Is Important For AI

As we reach the limits of what can be done on a single processor, more tasks are run on multiple processor cores. This is particularly important for AI.

One example of this is autonomous driving. In a traditional car, humans are relied upon to make quick decisions. And the average reaction time for humans is 0.25 seconds.

So, within autonomous vehicles, AI needs to make these decisions very quickly — in tenths of a second.

Using multithreading in C and parallel programming in C is the best way to ensure these decisions are made in a required timeframe.

C/C++ Languages Now Include Multithreading Libraries

Moving from single-threaded programs to multithreaded increases complexity. Programming languages, such as C and C++, have evolved to make it easier to use multiple threads and handle this complexity. Both C and C++ now include threading libraries.

Modern C++, in particular, has gone a long way to make parallel programming easier. C++11 included a standard threading library. C++17 added parallel algorithms — and parallel implementations of many standard algorithms.

Additional support for parallelism is expected in future versions of C++.

What Are Common Multithreaded Programming Issues?

There are many benefits to multithreading in C. But there are also concurrency issues that can arise. And these errors can compromise your program — and lead to security risks.

Using multiple threads helps you get more out of a single processor. But then these threads need to sync their work in a shared memory. This can be difficult to get right — and even more difficult to do without concurrency issues.

Traditional testing and debugging methods are unlikely to identify these potential issues. You might run a test or a debugger once — and see no errors. But when you run it again, there's a bug. In reality, you could keep testing and testing — and still not find the issue.

Here are two common types of multithreading issues that can be difficult to find with testing and debugging alone.

Race Conditions (Including Data Race)

Race conditions occur when a program's behavior depends on the sequence or timing of uncontrollable events.

A data race is a type of race condition. A data race occurs when two or more threads access shared data and attempt to modify it at the same time — without proper synchronization.

This type of error can lead to crashes or memory corruption.

Deadlock

Deadlock occurs when multiple threads are blocked while competing for resources. One thread is stuck waiting for a second thread, which is stuck waiting for the first.

This type of error can cause programs to get stuck.