Concurrency in C++

Felix Zhou

May 24, 2019

Contents

1 Introduction **3**
1.1 What is Concurrency? . . . 3
1.2 What is Multithreading? . . . 3
1.3 Reasons for Multithreading . . . 3
1.3.1 Performance . . . 3
1.3.2 Seperation of Concerns . . . 3

2 Thread Management **4**

3 Sharing Data **5**
3.1 Race Conditions . . . 5
3.1.1 Mutexes . . . 5
3.1.2 Interfaces . . . 5
3.2 Deadlocks . . . 5
3.2.1 Fixed-Order Locking . . . 5
3.2.2 Backoff . . . 5
3.2.3 Lock Hierarchy . . . 5
3.3 Alternate Data Protection Schemes . . . 6
3.3.1 Protecting Data Once . . . 6

4 Synchronizing Threads **7**
4.1 Waiting for an Event . . . 7
4.2 Waiting for One-Off Events . . . 7
4.3 Packaging Tasks . . . 7
4.4 Waiting with a Time Limit . . . 7

List of Figures

List of Tables

1 Introduction

1.1 What is Concurrency?

Definition 1.1.1 (Concurrency)
A single system performing multiple independent activities in parallel rather than sequentially.

There are many ways this can benefit us such as running multiple processes, possibly even over different machines, to achieve a single computational task. We will focus on *multithreading*, which concerns more what we can do on a single machine.

1.2 What is Multithreading?

Definition 1.2.1 (Multithreading)
Multithreading is a type of execution model that allows multiple threads to exist within the context of a process such that they execute independently but share their process resources.

We can think of this as having two workers working on a single site (multithreading) versus having two workers working on two different sites (multiprocessing). Both can achieve performance gains but we will focus on the first one.

1.3 Reasons for Multithreading

Definition 1.3.1 (Embarrassingly/Naturally/Conveniently Parallel/Concurrent)
Algorithms such that increasing the number of hardware threads would lead to improved parallelism in of the algorithm.

1.3.1 Performance

This is the main selling point of multithreading. If you are familiar with machine learning, or any sort of computational linear algebra, you would know that matrix multiplication can be performed in parallel to achieve speed ups. Another (not completely) relevant example is Google Chrome, which by default starts a new **process** for every tab you open, making it very resource intensive but blazing fast.

1.3.2 Seperation of Concerns

This is the lesser known but arguably equally relevant reason to include multithreading in your tool-kit. Consider any sort of input-logic-display type of program. While waiting for user input is a blocking task, we could implement the logic so that it does not have to be blocked by the input! This would require some form of communication between threads, which we will address in future sections.

2 Thread Management

Let A be an array of size n and f a function returning nothing that we wish to apply to each element of the array seperately. It might be a good idea to parallellize this!

This begs the question of how many threads should we employ so that we can gain the most performance increase.

Naively, we may choose to use one thread per element of the array but doing so shows ignorance of the overhead of thread allocation. We would also be ignoring the overhead of our processors managing the threads.

A more ideal method is to decide the number of hardware threads at run-time, divide A into blocks of size B, and allocate $\left\lfloor \frac{n}{B} \right\rfloor - 1$ threads. Note the -1 is due to the fact that the caller thread should be able to do work as well.

3 Sharing Data

3.1 Race Conditions

Definition 3.1.1 ((Problematic) Race Condition)
When there is an outcome which depends on the relative ordering of execution of operations on two or more threads leading to broken invariants.

3.1.1 Mutexes

One way of avoiding race conditions is to group intermediate sequences which have broken invariants together and only allow the thread doing the modification to the data structure to see these steps.
Essentially, we are marking the code which modifies a structure as *mutually exclusive*, hence the term mutex.

3.1.2 Interfaces

It may be the case that in order to prevent race conditions when adapting an interface, we must group different functions together.

3.2 Deadlocks

Note that with the introduction of mutexes, we introduce a new problem. If threads 1 and 2 acquire locks A, B respectively, then attempt to acquire B, A, this would lead to a deadlock.

Definition 3.2.1 (Deadlock)
When two threads hold one lock each and in addition attempts to acquire the locks of each other by waiting.

This can normaly be avoided by always locking A before B, but if the locks each protect an instance of the same type of data, this could lead to complications.

3.2.1 Fixed-Order Locking

If there is a way of putting instances of data in a total order such as indice in a linked list, choosing a fixed order in which we may lock instances is a sure way to prevent deadlocks.

3.2.2 Backoff

It may be that choosing a fixed order is not feasible or even possible. In this case, we can apply some sort of heuristic to try all the possible ways to lock instances of data until we lock everything.

3.2.3 Lock Hierarchy

This is a special case of fixed-order locking which checks for violations at run-time. We divide the program into layers and enforce the invariant that threads holding a lock from a layer with lower hierarchy is unable to acquire the lock with higher hierarchy.

3.3 Alternate Data Protection Schemes

3.3.1 Protecting Data Once

In the case of, or similar to, lazy initialization of data, where we wish to perform a procedure once and protect it, it might be an overkill to use a mutex. We can employ some sort of once flag so that the initilization will only happen once.

Our work in preventing deadlocks can be extended for any sort of issue where a cycle occurs in computation.

4 Synchronizing Threads

If there was no communication between our threads, we might as well just run them as different processes.

4.1 Waiting for an Event

We can loop until a flag has been set to indicate that we may continue processing on this thread, on the other hand, we can subscribe to a condition variable and have that variable notify a thread to wake up and continue execution.

4.2 Waiting for One-Off Events

We essentially wish to have some sort of a getter and setter between threads, with the setter only being able to set the value of data to be trasferred to another thread and the getter being able to retrieve the data, possibly multiple times. The getter can also then block until the setter has its value set.

4.3 Packaging Tasks

The idea of getters and setter between threads can be abstracted into packaged tasks. We can specify a callable and wrap it with a container which also contains a setter. The container can then "give out" getters. All this sets up the possibility of invoking our container with the appropriate parameters, which then triggers the setter, leading to any blocking getters being notified.

4.4 Waiting with a Time Limit