An Introduction to Programming with C# Threads

Andrew D. Birrell

This paper provides an introduction to writing concurrent programs with "threads". A threads facility allows you to write programs with multiple simultaneous points of execution, synchronizing through shared memory. The paper describes the basic thread and synchronization primitives, then for each primitive provides a tutorial on how to use it. The tutorial sections provide advice on the best ways to use the primitives, give warnings about what can go wrong and offer hints about how to avoid these pitfalls. The paper is aimed at experienced programmers who want to acquire practical expertise in writing concurrent programs. The programming language used is C#, but most of the tutorial applies equally well to other languages with thread support, such as Java.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Programming; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Concurrent programming structures*; D.4.1 [**Operating Systems**]: Process Management

General Terms: Design, Languages, Performance

Additional Key Words and Phrases: Threads, Concurrency, Multi-processing, Synchronization

---

CONTENTS

---

1. INTRODUCTION

Almost every modern operating system or programming environment provides support for concurrent programming. The most popular mechanism for this is some provision for allowing multiple lightweight "threads" within a single address space, used from within a single program.

Programming with threads introduces new difficulties even for experienced programmers. Concurrent programming has techniques and pitfalls that do not occur in sequential programming. Many of the techniques are obvious, but some are obvious only with hindsight. Some of the pitfalls are comfortable (for example, deadlock is a pleasant sort of bug—your program stops with all the evidence intact), but some take the form of insidious performance penalties.

The purpose of this paper is to give you an introduction to the programming techniques that work well with threads, and to warn you about techniques or interactions that work out badly. It should provide the experienced sequential programmer with enough hints to be able to build a substantial multi-threaded program that works—correctly, efficiently, and with a minimum of surprises.

This paper is a revision of one that I originally published in 1989 [2]. Over the years that paper has been used extensively in teaching students how to program with threads. But a lot has changed in 14 years, both in language design and in computer hardware design. I hope this revision, while presenting essentially the same ideas as the earlier paper, will make them more accessible and more useful to a contemporary audience.

A "thread" is a straightforward concept: a single sequential flow of control. In a high-level language you normally program a thread using procedure calls or method calls, where the calls follow the traditional stack discipline. Within a single thread, there is at any instant a single point of execution. The programmer need learn nothing new to use a single thread.

Having "multiple threads" in a program means that at any instant the program has multiple points of execution, one in each of its threads. The programmer can mostly view the threads as executing simultaneously, as if the computer were endowed with as many processors as there are threads. The programmer is required to decide when and where to create multiple threads, or to accept such decisions made for him by implementers of existing library packages or runtime systems. Additionally, the programmer must occasionally be aware that the computer might not in fact execute all his threads simultaneously.

Having the threads execute within a "single address space" means that the computer's addressing hardware is configured so as to permit the threads to read and write the same memory locations. In a traditional high-level language, this usually corresponds to the fact that the off-stack (global) variables are shared among all the threads of the program. In an object-oriented language such as C# or Java, the static variables of a class are shared among all the threads, as are the instance variables of any objects that the threads share.* Each thread executes on a separate call stack with its own separate local variables. The programmer is

---

* There is a mechanism in C# (and in Java) for making static fields thread-specific and not shared, but I'm going to ignore that feature in this paper.