1.2. Why use concurrency?

There are two main reasons to use concurrency in an application: separation of concerns and performance. In fact, I'd go so far as to say that they're pretty much the *only* reasons to use concurrency; anything else boils down to one or the other (or maybe even both) when you look hard enough (well, except for reasons like "because I want to").

1.2.1. Using concurrency for separation of concerns

Separation of concerns is almost always a good idea when writing software; by grouping related bits of code together and keeping unrelated bits of code apart, you can make your programs easier to understand and test, and thus less likely to contain bugs. You can use concurrency to separate distinct areas of functionality, even when the operations in these distinct areas need to happen at the same time; without the explicit use of concurrency you either have to write a task-switching framework or actively make calls to unrelated areas of code during an operation.

Consider a processing-intensive application with a user interface, such as a DVD player application for a desktop computer. Such an application fundamentally has two sets of responsibilities: not only does it have to read the data from the disk, decode the images and sound, and send them to the graphics and sound hardware in a timely fashion so the DVD plays without glitches, but it must also take input from the user, such as when the user clicks Pause or Return To Menu, or even Quit. In a single thread, the application has to check for user input at regular intervals during the playback, thus conflating the DVD playback code with the user interface code. By using multithreading to separate these concerns, the user interface code and DVD playback code no longer have to be so closely intertwined; one thread can handle the user interface and another the DVD playback. There will have to be interaction between them, such as when the user clicks Pause, but now these interactions are directly related to the task at hand.

This gives the illusion of responsiveness, because the user interface thread can typically respond immediately to a user request, even if the response is simply to display a busy cursor or Please Wait message while the request is conveyed to the thread doing the work. Similarly, separate threads are often used to run tasks that must run continuously in the background, such as monitoring the filesystem for changes in a desktop search application. Using threads in this way generally makes the logic in each thread much simpler, because the interactions between them can be limited to clearly identifiable points, rather than having to intersperse the logic of the different tasks.

In this case, the number of threads is independent of the number of CPU cores available, because the division into threads is based on the conceptual design rather than an attempt to increase throughput.

1.2.2. Using concurrency for performance

Multiprocessor systems have existed for decades, but until recently they were mostly found only in supercomputers, mainframes, and large server systems. But chip manufacturers have increasingly been favoring multicore designs with 2, 4, 16, or more processors on a single chip over better performance with a single core. Consequently, multicore desktop computers, and even multicore

embedded devices, are now increasingly prevalent. The increased computing power of these machines comes not from running a single task faster but from running multiple tasks in parallel. In the past, programmers have been able to sit back and watch their programs get faster with each new generation of processors, without any effort on their part. But now, as Herb Sutter put it, "The free lunch is over."[1] *If software is to take advantage of this increased computing power, it must be designed to run multiple tasks concurrently*. Programmers must therefore take heed, and those who have hitherto ignored concurrency must now look to add it to their toolbox.

[1] "The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software," Herb Sutter, *Dr. Dobb's Journal*, 30(3), March 2005. http://www.gotw.ca/publications/concurrency-ddj.htm.

There are two ways to use concurrency for performance. The first, and most obvious, is to divide a single task into parts and run each in parallel, thus reducing the total runtime. This is *task parallelism*. Although this sounds straightforward, it can be quite a complex process, because there may be many dependencies between the various parts. The divisions may be either in terms of processing—one thread performs one part of the algorithm while another thread performs a different part—or in terms of data—each thread performs the same operation on different parts of the data. This latter approach is called *data parallelism*.

Algorithms that are readily susceptible to such parallelism are frequently called *embarrassingly parallel*. Despite the implications that you might be embarrassed to have code so easy to parallelize, this is a good thing: other terms I've encountered for such algorithms are *naturally parallel* and *conveniently concurrent*. Embarrassingly parallel algorithms have good scalability properties—as the number of available hardware threads goes up, the parallelism in the algorithm can be increased to match. Such an algorithm is the perfect embodiment of the adage, "Many hands make light work." For those parts of the algorithm that aren't embarrassingly parallel, you might be able to divide the algorithm into a fixed (and therefore not scalable) number of parallel tasks. Techniques for dividing tasks between threads are covered in chapter 8.

The second way to use concurrency for performance is to use the available parallelism to solve bigger problems; rather than processing one file at a time, process 2 or 10 or 20, as appropriate. Although this is really just an application of *data parallelism*, by performing the same operation on multiple sets of data concurrently, there's a different focus. It still takes the same amount of time to process one chunk of data, but now more data can be processed in the same amount of time. Obviously, there are limits to this approach too, and this won't be beneficial in all cases, but the increase in throughput that comes from such an approach can actually make new things possible—increased resolution in video processing, for example, if different areas of the picture can be processed in parallel.

1.2.3. When not to use concurrency

It's just as important to know *when not* to use concurrency as it is to know *when* to use it. Fundamentally, the only reason not to use concurrency is when the benefit is not worth the cost. Code using concurrency is harder to understand in many cases, so there's a direct intellectual cost to writing and maintaining multithreaded code, and the additional complexity can also lead to more bugs. Unless the potential performance gain is large enough or separation of concerns clear enough to justify the additional development time required to get it right and the additional costs associated with maintaining multithreaded code, don't use concurrency.

Also, the performance gain might not be as large as expected; there's an inherent overhead associated with launching a thread, because the OS has to allocate the associated kernel resources and stack space and then add the new thread to the scheduler, all of which takes time. If the task

being run on the thread is completed quickly, the actual time taken by the task may be dwarfed by the overhead of launching the thread, possibly making the overall performance of the application worse than if the task had been executed directly by the spawning thread.

Furthermore, threads are a limited resource. If you have too many threads running at once, this consumes OS resources and may make the system as a whole run slower. Not only that, but using too many threads can exhaust the available memory or address space for a process, because each thread requires a separate stack space. This is particularly a problem for 32-bit processes with a flat architecture where there's a 4 GB limit in the available address space: if each thread has a 1 MB stack (as is typical on many systems), then the address space would be all used up with 4096 threads, without allowing for any space for code or static data or heap data. Although 64-bit (or larger) systems don't have this direct address-space limit, they still have finite resources: if you run too many threads, this will eventually cause problems. Though thread pools (see chapter 9) can be used to limit the number of threads, these are not a silver bullet, and they do have their own issues.

If the server side of a client/server application launches a separate thread for each connection, this works fine for a small number of connections, but can quickly exhaust system resources by launching too many threads if the same technique is used for a high-demand server that has to handle many connections. In this scenario, careful use of thread pools can provide optimal performance (see chapter 9).

Finally, the more threads you have running, the more context switching the operating system has to do. Each context switch takes time that could be spent doing useful work, so at some point adding an extra thread will actually *reduce* the overall application performance rather than increase it. For this reason, if you're trying to achieve the best possible performance of the system, it's necessary to adjust the number of threads running to take account of the available hardware concurrency (or lack of it).

Use of concurrency for performance is just like any other optimization strategy: it has potential to greatly improve the performance of your application, but it can also complicate the code, making it harder to understand and more prone to bugs. Therefore it's only worth doing for those performance-critical parts of the application where there's the potential for measurable gain. Of course, if the potential for performance gains is only secondary to clarity of design or separation of concerns, it may still be worth using a multithreaded design.

Assuming that you've decided you *do* want to use concurrency in your application, whether for performance, separation of concerns, or because it's "multithreading Monday," what does that mean for C++ programmers?