- Critical sections, which can be optimized with spin count tuning.
- Mutexes, which can be dramatically slower than the alternatives, especially with multiple processors. A semaphore throttle can be marginally useful in some cases with a large number of threads.

Appendix C gives additional results, using other systems and different parameter values.

Parallelism Revisited

Chapter 7 discussed some basic program characteristics that allow application threads to run in parallel, potentially improving program performance, especially on multiprocessor systems.

Parallelism has become an important topic because it is the key to improving application performance, since processor clock rates no longer increase regularly as they have in the past. Most systems today and in the foreseeable future, whether laptops or servers, will have clock rates in the 2–3GHz range. Instead, chip makers are marketing multicore chips with 2, 4, or more processors on a single chip. In turn, system vendors are installing multiple multicore chips in their systems so that systems with 4, 8, 16, or more total processors are common.[2]

Therefore, if you want to increase your application's performance, you will need to exploit its inherent parallelism using threads, NT6 thread pools, and, possibly, parallelism frameworks (which this section surveys). We've seen several simple examples, but these examples give only a partial view of what can be achieved:

- In most cases, such as `wcMT` and `grepMT`, the parallel operations are straightforward. There is one thread per file.
- `sortMT` is more complex, as it divides a single data structure, an array, into multiple parts, sorts them, and merges the results. This is a simplistic use of the divide and conquer strategy.
- `cciMT` (Chapter 14) divides the file into multiple segments and works on them independently.

In each case, the maximum potential speedup, compared to a single-threaded implementation or running on a single processor, is easy to determine. For example,

[2] Nearly all desktop systems, most laptops, and many notebooks sold today have multiple processors and cost less than single-processor systems sold a few years ago. The "Additional Reading" section suggests references for parallelism and the appropriate technology trends.

the speedup for `wcMT` and `grepMT` is limited by the minimum of the number of processors and the number of files.

A Better Foundation and Extending Parallel Program Techniques

The previous parallelism discussion was very informal and intuitive, and the implementation, such as `wcMT`, are all boss/worker models where the workers are long-lived, executing for essentially the entire duration of the application execution.

While a formal discussion is out of scope for this book, it's important to be aware that parallelism has been studied extensively, and there is a solid theoretical foundation along with definitions for important concepts. There are also analytical methods to determine algorithmic complexity and parallelism. Interested readers will find several good treatments; for example, see Chapter 27 of Cormen, Leiserson, Rivest, and Stein, *Introduction to Algorithms,* Third Edition.

Furthermore, parallel programming techniques are far more extensive than the boss/worker examples used here. For example:

- Parallel sort-merge can be made far more effective than `sortMT`'s rudimentary divide and conquer design.
- Recursion and divide and conquer techniques are important for exploiting finer-grained program parallelism where parallel tasks can be short-lived relative to total program duration.
- Computational tasks that are amenable to parallel programming include, but are hardly limited to, matrix multiplication, fast Fourier transformations, and searching. Usually, a new thread is created for every recursive function call.
- Games and simulations are other application types that can often be decomposed into parallel components.

In short, parallelism is increasingly important to program performance. Our examples have suggested the possibilities, but there are many more that are beyond this book's scope.

Parallel Programming Alternatives

Once you have identified the parallel components in your program, the next issue is to determine how to implement the parallelism. There are several alternatives; we've been using the first two, which Windows supports well.

- The most direct approach is to "do it yourself" (DIY), as in some of the examples. This requires direct thread management and synchronization. DIY is manageable and effective for smaller programs or programs with a simple parallel structure. However, DIY can become complex and error prone, but not impossible, when implementing recursion.
- Thread pools, both legacy and NT6 thread pools, enable advanced kernel scheduling and resource allocation methods that can enhance performance. Be aware, however, that the NT6 thread pool API is limited to that kernel; this should become less of an issue in coming years.
- Use a parallelism framework, which extends the programming language to help you express program parallelism. See the next section.

Parallelism Frameworks

Several popular "parallelism frameworks" offer alternatives to DIY or thread pools. Framework properties include:

- Programming language extensions that express program parallelism. The two most common extensions are to express *loop parallelism* (that is, every loop iteration can execute concurrently) and *fork-join parallelism* (that is, a function call can run independently from the calling program, which eventually must wait for the called function to complete). The language extensions may take the form of compiler directives or actual extensions that require compiler front ends.
- The supported languages almost always include C and C++, often C# and Java, and Fortran for scientific and engineering applications.
- There is run-time support for efficient scheduling, locking, and other tasks.
- There is support for result "reduction" where the results from parallel tasks are combined (for instance, word counts from individual files are summed), and there is care to minimize locking.
- Parallel code can be serialized for debugging and produces the same results as the parallel code.
- The frameworks frequently contain tools for race detection and identifying and measuring parallelism during program execution.
- The frameworks are sometimes open source and portable to UNIX and Linux.

Popular parallelism frameworks include:[3]

- OpenMP is open source, portable, and scalable. Numerous compilers and development environments, including Visual C++, support OpenMP.
- Intel Thread Building Blocks (TBB) is a C++ template library. The commercial release is available on Windows as well as many Linux and UNIX systems.
- Intel's Cilk++ supports C and C++. Cilk++ is available on Windows and Linux and provides a simple language extension with three key words along with a run-time library for "work stealing" scheduling. Cilk++ also provides flexible reduction templates.
- .NET Framework 4's Task Parallel Library (TPL), not yet available, will simplify creating applications with parallelism and concurrency.

Do Not Forget the Challenges

As stated previously, this book takes the point of view that multithreaded programming is straightforward, beneficial, and even enjoyable. Nonetheless, there are numerous pitfalls, and we've provided numerous guidelines to help produce reliable multithreaded programs; there are even more guidelines in the next chapter. Nonetheless, do not overlook the challenges when developing multithreaded applications or converting legacy systems. These challenges are daunting even when using a parallelism framework. Here are some of the notable challenges that you can expect and that have been barriers to successful implementations:

- Identifying the independent subtasks is not always straightforward. This is especially true for legacy applications that may have been developed with no thought toward parallelism and threading.
- Too many subtasks can degrade performance; you may need to combine smaller subtasks.
- Too much locking can degrade performance, and too little can cause race conditions.
- Global variables, which are common in large, single-threaded applications, can cause races if independent subtasks modify global variables. For example, a global variable might contain the sum or other combination of results from separate loop iterations; if the iterations run in parallel, you will need to find

[3] Wikipedia covers all of these, and a Web search will yield extensive additional information. This list is not complete, and it will take time for one or more frameworks to become dominant.

a way to combine ("reduce") the independent results to produce a single result without causing a data race.

- There can be subtle performance issues due to the memory cache architecture and the combination of multiple multicore chips.

Processor Affinity

The preceding discussion has assumed that all processors of a multiprocessor system are available to all threads, with the kernel making scheduling decisions and allocating processors to threads. This approach is simple, natural, consistent with multiprocessors, and almost always the best approach. It is possible, however, to assign threads to specific processors by setting processor affinity. Processor affinity can be used in several situations.

- You can dedicate a processor to a small set of one or more threads and exclude other threads from that processor. This assumes, however, that you control all the running applications, and even then, Windows can schedule its own threads on the processor.
- You can assign a collection of threads to a processor pair sharing L2 cache (see Figure 8–2) to minimize the delay caused by memory barriers.
- You may wish to test a processor. Such diagnostic testing, however, is out of scope for this book.
- Worker threads that contend for a single resource can be allocated to a single processor.

You may wish to skip this section, considering the specialized nature of the topic.

System, Process, and Thread Affinity Masks

Each process has its own process affinity mask, which is a bit vector. There is also a system affinity mask.

- The system mask indicates the processors configured on this system.
- The process mask indicates the processors that can be used by the process's threads. By default, its value is the same as the system mask.
- Each individual thread has a thread affinity mask, which must be a subset of the process affinity mask. Initially, a thread's affinity mask is the same as the process mask.

- Affinity masks are pointers (either 32 or 64 bits). Win32 supports up to 32 processors. Consult MSDN if you need to deal with more than 64 processors on Win64.

There are functions to get and set the masks, although you can only read (get) the system mask and can only set thread masks. The set functions use thread and process handles, so one process or thread can set the affinity mask for another, assuming access rights, or for itself. Setting a mask has no effect on a thread that might already be running on a processor that is masked out; only future scheduling is affected.

A single function, `GetProcessAffinityMask`, reads both the system and process affinity masks. On a single-processor system, the two mask values will be `1`.

```
BOOL GetProcessAffinityMask (
   HANDLE hProcess,
   LPDWORD lpProcessAffinityMask,
   LPDWORD lpSystemAffinityMask)
```

The process affinity mask, which will be inherited by any child process, is set with `SetProcessAffinityMask`.

```
BOOL SetProcessAffinityMask (
   HANDLE hProcess,
   DWORD_PTR dwProcessAffinityMask)
```

The new mask must be a *subset* of the values obtained from `GetProcessAffinityMask`. It does not, however, need to be a proper subset. Such a limitation would not make sense because you would not be able to restore a system mask to a previous value. The new value affects all the threads belonging to this process.

Thread masks are set with a similar function.

```
DWORD_PTR SetThreadAffinityMask (
   HANDLE hThread,
   DWORD dwThreadAffinityMask)
```

These functions are not designed consistently. `SetThreadAffinityMask` returns a `DWORD` with the previous affinity mask; `0` indicates an error. `SetThreadAffinityMask`, however, returns a `BOOL` and does not return the previous value.

`SetThreadIdealProcessor` is a variation of `SetThreadAffinityMask`. You specify the preferred ("ideal") processor number (not a mask), and the scheduler will assign that processor to the thread if possible, but it will use a different processor if the preferred processor is not available. The return value gives the previous preferred processor number, if any.

Finding the Number of Processors

The system affinity mask does indicate the number of processors on the system; all that is necessary is to count the number of bits that are set. It is easier, however, to call `GetSystemInfo`, which returns a `SYSTEM_INFO` structure whose fields include the number of processors and the active processor mask, which is the same as the system mask. A simple program and project, `version`, in the *Examples* file, displays this information along with the Windows version. See Exercise 6–12 for `version` output on the system used for the run screenshots in this chapter.

Performance Guidelines and Pitfalls

Multiple threads can provide significant programming advantages, including simpler programming models and performance improvement. However, there are several performance pitfalls that can have drastic and unexpected negative performance impact, and the impact is not always consistent on different computers, even when they are running the same Windows version. Some simple guidelines, summarizing the experience in this chapter, will help you to avoid these pitfalls. Some of these guidelines are adapted from Butenhof's *Programming with POSIX Pthreads,* as are many of the designing, debugging, and testing hints in the next chapter.

In all cases, of course, it's essential to maintain program correctness. For example, while the `statsXX` programs, as written, can run without locking, that is not the case in general. Likewise, when you make a critical code section as small as possible, be sure not to move critical code out of the code section. Thus, if the critical code section adds an element to a search tree, all the code required for the insertion operation must be in the critical code section.

- Beware of conjecture and theoretical arguments about performance, which often sound convincing but can be wrong in practice. Test the conjecture with a simple prototype, such as `TimedMutualExclusion`, or with alternative implementations of your application.

- Test application performance on as wide a variety of systems as are available to you. It is helpful to run with different memory configurations, processor types, Windows versions, and number of processors. *An application may perform very well on one system and then have extremely poor performance on a similar one;* see the discussion after Program 9–1.
- Locking is expensive; use it only as required. Hold (own) a lock, regardless of the type, only as long as required and no longer (see earlier comment). As an additional example, consider the message structures used in `simplePC` (Program 8–1). The critical code section incudes everything that modifies the message structure, and nothing else, and the invariant holds everywhere outside the critical code section.
- Use distinct locks for distinct resources so that locking is as granular as possible. In particular, avoid global locks.
- High lock contention hinders good performance. The greater the frequency of thread locking and unlocking, and the larger the number of threads, the greater the performance impact. Performance degradation can be drastic and is not just linear in the number of threads. Note, however, that this guideline involves a trade-off with fine-grained locking, which can increase locking frequency.
- CSs provide an efficient, lightweight locking mechanism. When using CSs in a performance-critical multiprocessor application, tune performance with the CS spin counts. SRW locks are even more efficient but do not have adjustable spin counts.
- Semaphores can reduce the number of active contending threads without forcing you to change your programming model.
- Multiprocessors can cause severe, often unexpected, performance impacts in cases where you might expect improved performance. This is especially true when using mutexes. Reducing contention and using thread affinity are techniques to maintain good performance.
- Investigate using commercially available profiling and performance analysis tools, which can help clarify the behavior of the threads in your program and locate time-consuming code segments.

Summary

Synchronization can impact program performance on both single-processor and multiprocessor systems; in some cases, the impact can be severe. Careful program design and selection of the appropriate synchronization objects can help assure good performance. This chapter discussed a number of useful techniques and

guidelines and illustrated performance issues with a simple test program that captures the essential characteristics of many real programming situations.

Looking Ahead

Chapter 10 shows how to use Windows synchronization in more general ways, particularly for message passing and correct event usage. It also discusses several programming models, or patterns, that help ensure correctness and maintainability, as well as good performance. Chapter 10 creates several compound synchronization objects that are useful for solving a number of important problems. Subsequent chapters use threads and synchronization as required for applications, such as servers. There are also a few more basic threading topics; for example, Chapter 12 illustrates and discusses thread safety and reentrancy in DLLs.

Additional Reading

Chapter 10 provides information sources that apply to this chapter as well. Duffy's *Concurrent Programming on Windows*, in addition to covering the synchronization API, also gives insight into the internal implementation and performance implications and compares the Windows features with features available in .NET. In particular, see Chapter 6 for synchronization and Chapter 7 for thread pools.

Chapter 27 of Cormen, Leiserson, Rivest, and Stein, *Introduction to Algorithms, Third Edition,* is invaluable for understanding parallelism and effective parallel algorithm design.

The Wikipedia "Multi-core" entry gives a good introduction to the commercial and technical incentives as well as long-term trends for multicore systems.

Exercises

9–1. Experiment with the `statsXX` variations on your own system and on as many different systems (both hardware and Windows versions) as are available to you. Do you obtain similar results as those reported in this chapter and in Appendix C?

9–2. Use `TimedMutualExclusionSC`, included in the *Examples* file, to experiment with `CRITICAL_SECTION` spin counts to see whether adjusting the count can improve and tune multiprocessor performance when you have a large number of threads. Results will vary from system to system, and I have found approximately optimal points ranging from 2,000 to 10,000. How do the best results compare with exclusive-mode SRW locks?

9–3. Experiment with the `statsXX` variations by modifying the delay time in the worker function. For example, increasing the delay should increase the total elapsed time for all variations, but the relative impact of the locking model could be less.

9–4. Use `TimedMutualExclusion` to experiment with delay and sleep point counts.

9–5. `TimedMutualExclusion` also uses a semaphore throttle to limit the number of running threads. Experiment with the count on both single-processor and multiprocessor systems. If an NT4 system is available, compare the results with NT5 and NT6.

9–6. Do the seven `statsXX` variations all operate correctly, ignoring performance, on multiprocessor systems? Experiment with a large number of worker threads. Run on a multiprocessor Windows 2003 or 2008 server. Can you reproduce the "false-sharing" performance problem described earlier?

9–7. Enhance Program 9–2 (`statsSRW_VTP`) to display the thread number as suggested after the program listing.

9–8. What is the effect of using an NT6 thread pool with a CS or mutex? Suggestion: Modify `statsSRW_VTP`. What is the effect of using a semaphore throttle with a CS (modify `statsMX_ST`)?

9–9. Rewrite `statsSRW_VTP` to use `TrySubmitThreadpoolCallback`. Compare the results and ease of programming with `statsSRW_VTP`.

9–10. Use processor affinity as a possible performance-enhancement technique by modifying this chapter's programs.

9–11. Run 9–1b compared SRWs with CSs, and SRWs were always considerably faster. Modify the `statsXX` programs so that they each use a pair of locks (be careful to avoid deadlocks!), each guarding a separate variable. Are CSs more competitive in this situation?

9–12. The `statsXX` programs have an additional command line parameter, not shown in the listings, that controls the delay time in the worker threads. Repeat the comparisons in Runs 9–1a and 9–1b with larger and smaller delays (changing the amount of contention). What is the effect?

CHAPTER

10 Advanced Thread Synchronization

The preceding chapter described Windows performance issues and how to deal with them in realistic situations. Chapter 8 described several simple problems that require synchronization. This chapter solves additional practical but more complex synchronization problems, relying on the ideas introduced in Chapters 8 and 9.

The first step is to combine two or more synchronization objects and data to create compound objects. The most useful combination is the "condition variable model" involving a mutex and one or more events. The condition variable model is essential in numerous practical situations and prevents many serious program race condition defects that occur when programmers do not use Windows synchronization objects, especially events, properly. Events are complex, and their behavior varies depending on the choices illustrated in Table 8–1, so they should be used according to well-understood models. In fact, even the condition variable model described here has limitations that we'll describe.

NT6 (Windows Vista, Windows Server 2008, and Windows 7) added condition variables to the Windows API, which is a significant advance that will be easy to understand after we cover the condition variable model. Programmers, however, will not be able to use condition variables if they need to support NT5 (Windows XP and Server 2003), which will probably be a requirement for many years after publication. NT6 condition variables are essential for a totally correct implementation that overcomes all the event limitations.

Subsequent sections show how to use asynchronous procedure calls (APCs) so that individual, cooperating threads can be controlled and canceled in an orderly manner.

Additional performance issues are discussed as appropriate.

The Condition Variable Model and Safety Properties

Threaded programs are much easier to develop, understand, and maintain if we use well-understood and familiar techniques and models. Chapter 7 discussed this and introduced the boss/worker and work crew models to establish a useful framework for understanding many threaded programs. The critical code region concept is essential when using mutexes, and it's also useful to describe the invariants of your data structure. Finally, even defects have models, as we saw with the deadlock example. *Note:* Microsoft has its own distinct set of models, such as the apartment model and free threading. These terms are most often used with COM.

Using Events and Mutexes Together

The next step is to describe how to use mutexes and events together, generalizing Program 8–2, where we had the following situation, which will occur over and over again. *Note:* This discussion applies to `CRITICAL_SECTION`s and SRW locks as well as to mutexes.

- The mutex and event are both associated with the message block or other data structure.
- The mutex defines the critical code section for accessing the data structure.
- The event signals that there is a new message or some other significant change to the data structure.
- Generalizing, the mutex ensures the object's invariants (or safety properties), and the event signals that the object has changed state (e.g., a message has been added or removed from a message buffer), possibly being put into a known state (e.g., there is at least one message in the message buffer).
- One thread (the producer in Program 8–2) locks the data structure, changes the object's state by creating a new message, and signals the event associated with the fact that there is a new message.
- At least one other thread (the consumer in this example) waits on the event for the object to reach the desired state. The wait must occur outside the critical code region so that the producer can access the object.
- A consumer thread can also lock the mutex, test the object's state (e.g., is there a new message in the buffer?), and avoid the event wait if the object is already in the desired state.

This general situation, where one thread changes a state variable and other threads wait for the change, occurs in numerous situations. The example here