Bitter, Rick et al "Multithreading in LabVIEW"
*LabVIEW Advanced Programming Techniques*
Boca Raton: CRC Press LLC, 2001
This chapter discusses using multithreading to improve LabVIEW applications’ performance. Multithreading is an advanced programming topic, and its use requires the programmer to possess a fundamental understanding of this technology. LabVIEW provides two significant advantages to the programmer when working with multitasking and multithreading. The first advantage is the complete abstraction of the threads themselves. LabVIEW programmers never create, destroy, or synchronize threads. The second advantage is the dataflow model used by LabVIEW. This model provides a distinct advantage over its textual language counterparts because it simplifies a programmer’s perception of multitasking. The fundamental concept of multitasking can be difficult to grasp with text-based languages.

Multithreading adds a new dimension to software engineering. Applications can perform multiple tasks somewhat simultaneously. A good example of an application that has added multithreading is Microsoft Word for Windows 95 (Version 7.0). Word for Windows 95 uses multithreading to perform spell-checking and grammar validation. The threads added to perform this task allow the application to perform these tasks while the user is typing. The previous version, Word 6.0 for Windows 3.1, cannot do this because it runs only one task at a time; a user would have to stop typing and select Check Spelling. The first six sections of this chapter provide the basic knowledge of multithreading. This discussion focuses on definitions, multitasking mechanics, multithreading specific problems, and information on various thread-capable operating systems.

A brief section on multithreading myths is presented. The impact of multithreading on applications is misunderstood by a number of programmers. Section 9.6 explains precisely what the benefits of multithreading are. Many readers will be surprised to learn that multithreading does little to increase the speed of an application. Multi-threading does provide the illusion that sections of an application run faster.

The last three sections of this chapter are devoted to the effective use of multithreading in LabVIEW. A strategy to estimate the maximum number of useful threads will be presented. The focal point of this chapter is using subroutine VIs to maximize application performance. The use of threads adds a new dimension of benefits to both subroutine VIs and DLLs.
9.1 MULTITHREADING TERMINOLOGY

The following terminology will be used throughout this chapter. Programmers who require additional information on any of these topics should consult the chapter bibliography.

9.1.1 Win32

Win32 is an Application Programming Interface (API) that is used by Microsoft’s 32-bit operating systems: Windows 95, Windows 98, and Windows NT. All three operating systems look very similar at the user interface. The common API for programming also makes the three operating systems look comparable to the programmer. Win32 replaces the Win16 API used in Windows 3.1.

Windows NT is designed to perform differently than Windows 95 and Windows 98. The primary focus of Windows 95 is to be as backward compatible as possible with Windows 3.1. Windows NT is designed to be as stable as possible. The differences in behavior of Windows NT and Windows 95/98 are usually not obvious to most users. For example, many users do not know that Windows 95/98 will stop preemptive multithreading for certain Windows 3.1 applications while Windows NT will not.

9.1.2 UNIX

UNIX is an operating system that was conceived by AT&T Bell Labs. Like Win32, UNIX is a standard, but the Open Systems Foundation maintains the UNIX standard. Unlike Win32, UNIX is supported by a number of vendors who write and maintain operating systems to this standard. The most popular are Sun Microsystems’ Solaris, IBM’s AIX, Hewlett Packard’s HP-UX, and LINUX.

Threads are specified in UNIX with the Portable Operating System Interface (POSIX) threads standard (pthread). At a low-level programming interface, pthreads are different than Win32 threads. This does not impact the LabVIEW programmer. Fundamentally, pthreads and Win32 threads operate in the same fashion. Their application-programming interfaces (API) are different, but, conceptually, threads are threads.

9.1.3 Multitasking

Multitasking simply means to coordinate multiple tasks, which is a familiar concept to the LabVIEW programmer. As a dataflow-based language, LabVIEW has always supported multitasking. Operating systems, such as Windows 3.1 and MacOS, multitask operations such as multiple applications. The act of coordinating multiple tasks should not be confused with multithreading; multithreading is fundamentally different, and this section will help to explain why.

The best example of multitasking is you. At work, you have a priority list of tasks that need to get accomplished. You will work on tasks one at a time. You may only accomplish some parts of a task, but not all before working briefly on another task. This is simple multitasking; you work on one thing for a while and then work on
another topic. The process of switching between tasks is not multithreading, it is simply multitasking. The act of deciding how long to work on a task before working on a different one is your scheduling algorithm. This is the principle that Windows 3.1 and MacOS are built on. The applications in Windows 3.1 and MacOS decide how long they will run before surrendering the CPU to another process. Figure 9.1 demonstrates time utilization of a CPU when cooperative multitasking is used.

A simple demonstration of multitasking in LabVIEW is independent While loops. It is important for the reader to clearly understand that multitasking has always been available, and multithreading does not add or subtract from LabVIEW’s ability to multitask operations. One of the key questions this chapter hopes to answer for the LabVIEW programmer is when multithreading will be of benefit.

9.1.3.1 Preemptive Multithreading

Taking the previous example of multitasking, you at work, we will explain the fundamental concept of multithreading. Imagine that your hands were capable of independently working. Your right hand could be typing a memo while the left dialed a phone number. Once the left hand completed dialing the number, it began to solder components on a circuit board. If you were capable of talking on the phone, typing a memo, and soldering components at the same time, you would be multithreading. Your body is effectively a process, and your hands and mouth are threads of execution. They belong to the same process, but are functioning completely independent of each other.

This is fundamentally what multithreading is doing in a computer program. Each thread has a task it works on regardless of what the rest of the program is doing. This is a difficult concept for many programmers to grasp. When programming with text-based languages such as C/C++, programmers associate lines of code as operating sequentially. The concept that is difficult to grasp is that threads behave as their own little program running inside a larger one. LabVIEW’s graphical code allows programmers to visualize execution paths much easier.

Preemptive multithreading uses CPU hardware and an operating system capable of supporting threads of execution. Preemption occurs at the hardware level; a hardware timer interrupt occurs and the CPU takes a thread of execution off the CPU and brings in another one. A lot of interesting things happen with CPUs that support multithreading operating systems. First, each program has a memory map.
This memory map is unique to each application. The memory maps are translated into real memory addresses by hardware in the CPU. When preemption occurs, the timer in the CPU informs the CPU to change maps to the operating system’s management. The operating system will then determine which thread is next to run and inform the CPU to load that thread’s memory map. Operating systems that support LabVIEW and multithreading are Win32, Solaris, HP-UX, and Concurrent PowerMax. Windows 3.1 and MacOS do not support multithreading for applications. The act of scheduling threads and processes will be explained in Section 9.2. Figure 9.2 shows the timelines for multiple processes.

9.1.4 Kernel Objects

Kernel objects are small blocks of memory, often C structures, that are owned by the operating system. They are created at the request of programs and are used to protect sections of memory. Later in this chapter the need for protection will be clearly explained, but a short definition of this term is provided now.

9.1.5 Thread

Before we begin describing a thread, a few terms used for programs must be quickly defined. A program that has one thread of execution, which is a single-threaded program, must have a call stack. The call stack retains items like variables and what the next instruction is to be executed. C programmers will quickly associate variables placed on the stack as local variables. In some operating systems, the program will also have a copy of CPU registers. CPU registers are special memory locations inside the CPU. The advantage of using CPU registers is that the CPU can use these memory locations significantly faster than standard memory locations. The disadvantage is that there are relatively few memory locations in the registers. Each thread has its own call stack and copy of CPU registers.

Effectively, a thread of execution is a miniature program running in a large, shared memory space. Threads are the smallest units that may be scheduled time for execution on the CPU and possess a call stack and set of CPU registers. The call stack is a First-In-First-Out (FIFO) stack that is used to contain things like function calls and temporary variables. A thread is only aware of its own call stack. The registers are loaded into the CPU when the thread starts its execution cycle, and pulled out and loaded back into memory when the thread completes its execution time.
9.1.6 Process

The exact definition of a process depends on the operating system, but a basic definition includes a block of memory and a thread of execution. When a process is started by the operating system, it is assigned a region of memory to operate in and has a list of instructions to execute. The list of instructions to begin processing is the “thread of execution.” All applications begin with a single thread and are allowed to create additional threads during execution.

The process’s memory does not correlate directly to physical memory. For example, a Win32 process is defined as four gigabytes of linear address space and at least one thread of execution. The average computer has significantly less memory. Notice that there is no mention made of things like conventional memory, extended memory, and high memory. This is referred to as a memory map, or protected memory. The operating system is responsible for mapping the addresses the program has into the physical memory. The concept behind protected memory is that a process cannot access memory of other processes because it has no idea where other memory is. The operating system and CPU switch memory in and out of physical memory and hard disk space. Memory mapped to the hard disk is referred to as “virtual memory” in Windows and “swap space” in UNIX.

A process also has security information. This information identifies what the process has authorization to do in the system. Security is used in Windows NT and UNIX, but is completely ignored in Windows 95 and 98.

9.1.7 Application

An application (or program) is a collection of processes. With concepts like Distributed Computing Environments (DCE) and Distributed Component Object Model (DCOM), applications are not required to execute in one process, or even on one computer. With the lower cost of computers and faster network speeds, distributed computing is becoming feasible and desirable in many applications. Applications that use processes on multiple machines are “distributed applications.”

As an example of a distributed application, consider LabVIEW. With ActiveX support and the new VI Server functionality, VIs can control other VIs that are not resident and executing on the same computer. Component Object Model (COM) and Distributed Component Objects (DCOM) were discussed in Chapter 7 on ActiveX.

9.1.8 Priority

Every process and thread has a priority associated with it. The priority of the process or thread determines how important it is when compared to other processes or threads. Priorities are numbers, and the higher the number, the more important the process. Process priorities are relative to other processes while thread priorities are relative only to other threads in the same process. LabVIEW programmers have access to priority levels used by LabVIEW. Configuring LabVIEW’s thread usage will be discussed in Section 9.7.4.
9.1.8.1 How Operating Systems Determine which Threads

Both Win32 and POSIX have 32-integer values that are used to identify the priority of a process. The implementation of priority and scheduling is different between Win32 and pthreads. Additional information on both specifications appears in Sections 9.3 and 9.4.

9.1.9 Security

Windows NT and UNIX systems have security attributes that need to be verified by the operating system. Threads may only operate within the security restrictions the system places on them. When using Distributed COM objects, and in some cases ActiveX containers, security permissions can become an issue. The default security attributes associated with an application are the level of permissions the user who started the application has. System security can limit access to files, hardware devices, and network components.

9.1.10 Thread Safe

Programmers often misunderstand the term “thread safe.” The concept of thread-safe code implies that data access is atomic. “Atomic” means that the CPU will execute the entire instruction, regardless of what external events occur, such as interrupts. Assembly-level instructions require more than one clock cycle to execute, but are atomic. When writing higher-level code, such as LabVIEW VIs or C/C++ code, the concept of executing blocks of code in an atomic fashion is critical when multiple threads of execution are involved. Threads of execution are often required to have access to shared data and variables. Atomic access allows for threads to have complete access to data the next time they are scheduled.

The problem with preemption is that a thread is removed from the CPU after completion of its current machine instruction. Commands in C can be comprised of dozens of machine-level instructions. It is important to make sure data access is started and completed without interference from other threads. Threads are not informed by the operating system that they were preemptively removed from the CPU. Threads cannot know shared data was altered when it was removed from the CPU. It is also not possible to determine preemption occurred with code; it is a hardware operation that is hidden from threads. Several Kernel objects can be used to guarantee that data is not altered by another thread of execution.

Both UNIX Pthreads and Win32 threads support semaphores and mutexes. A Mutual Exclusion (mutex) object is a Kernel object that allows one thread to take possession of a data item. When a thread requires access to a data item that is protected by a mutex, it requests ownership from the operating system. If no other threads currently own the mutex, ownership is granted. When preemption occurs, the owning thread is still shifted off the CPU, but when another thread requests ownership of the mutex it will be blocked. A thread that takes possession of a mutex is required to release the mutex. The operating system will never force a thread to relinquish resources it has taken possession of. It is impossible for the operating
system to determine if data protected by the mutex is in a transient state and would cause problems if another thread were given control of the data.

A semaphore is similar to a mutex, but ownership is permitted by a specified number of threads. An analogy of a semaphore is a crowded nightclub. If capacity of the club is limited to 500 people, and 600 people want to enter the club, 100 are forced to wait outside. The doorman is the semaphore, and restricts access to the first 500 people. When a person exits, another individual from outside is allowed to enter. A semaphore works the same way.

Mutexes and semaphores must be used in DLLs and code libraries if they are to be considered thread-safe. LabVIEW can be configured to call DLLs from the user interface subsystem, its primary thread, if it is unclear that the DLL is thread safe. A programmer should never assume that code is thread safe; this can lead to very difficult issues to resolve.

### 9.2 THREAD MECHANICS

All activities threads perform are documented in an operating system’s specification. The actual behavior of the threads is dependent on a vendor’s implementation. In the case of Windows, there is only one vendor, Microsoft. On the other hand, UNIX has a number of vendors who implement the standard in slightly different ways. Providing detailed information on operating system-specific details for all the UNIX flavors is beyond the scope of this book.

Regardless of operating system, all threads have a few things in common. First, threads must be given time on the CPU to execute. The operating system scheduler determines which threads get time on the CPU. Second, all threads have a concept of state; the state of a thread determines its eligibility to be given time on the CPU.

#### 9.2.1 THREAD STATES

A thread can be in one of three states: Active, Blocked, or Suspended. Active threads will be arranged according to their priority and allocated time on the CPU. An active thread may become blocked during its execution. In the event an executing thread becomes blocked, it will be moved into the inactive queue, which will be explained shortly.

Threads that are blocked are currently waiting on a resource from the operating system (a Kernel object or message). For example, when a thread tries to access the hard drive of the system, there will be a delay on the order of 10 ms for the hard drive to respond. The operating system blocks this thread because it is now waiting for a resource and would otherwise waste time on the CPU. When the hard drive triggers an interrupt, it informs the operating system it is ready. The operating system will signal the blocked thread and the thread will be put back into a run queue.

Suspended threads are “sleeping.” For example, in C, using the Sleep statement will suspend the thread that executed the command. The operating system effectively treats blocked and suspended threads in the same fashion; they are not allowed time on the CPU. Both suspended and blocked threads are allowed to resume execution when they have the ability to, when they are signaled.
9.2.2 Scheduling Threads

The operation of a scheduling algorithm is not public knowledge for most operating systems, or at least a vendor's implementation of the scheduling algorithm. The basic operation of scheduling algorithms is detailed in this section.

The operating system will maintain two lists of threads. The first list contains the active threads and the second list contains blocked and suspended threads. The active thread list is time-ordered and weighted by the priority of the thread in the system. The highest priority thread will be allowed to run on the CPU for a specified amount of time, and will then be switched off the CPU. This is referred to as a “Round Robin” scheduling policy.

When there are multiple CPUs available to the system, the scheduler will determine which threads get to run on which CPU. Symmetric Multiprocessing (SMP) used in Windows NT allows for threads of the same process to run on different CPUs. This is not always the case. A dual-CPU machine may have threads of different processes running on the pair of CPUs, which is determined by the scheduling algorithm. Some UNIX implementations only allow a process’s threads on a single CPU.

The blocked/suspended queue is not time-ordered. It is impossible for the operating system to know when a signal will be generated to unblock a thread. The scheduling algorithm polls this list to determine if any threads have become available to execute. When a thread becomes unblocked and has higher priority than the currently running thread, it will be granted control of the CPU.

9.2.3 Context Switching

The process of changing which thread is executing on the CPU is called “context switching.” Context switching between threads that are in the same process is relatively fast. This is referred to as a “thread context switch.” The CPU needs to offload the current thread’s instruction pointer and its copy of the CPU registers into memory. The CPU will then load the next thread’s information and begin executing the next thread. Since threads in the same process execute in the same protected memory, there is no need to remap physical memory into the memory map used by the process.

When context switching occurs between threads of different processes, it is called a “process context switch.” There is a lot more work that is required in a process context switch. In addition to swapping out instruction pointers and CPU registers, the memory mapping must also be changed for the new thread.

As mentioned in Section 9.2.2, when a thread becomes signaled (eligible to run) and has a higher priority than the currently running thread, it will be given control of the CPU. This is an involuntary context switch. This is a potential problem for LabVIEW programmers. Section 9.5 will discuss multithreading problems such as starvation and priority inversion that can be caused by poorly-designed configurations of LabVIEW’s thread pools. Configuring the threads that LabVIEW uses is discussed in Section 9.7.4.
9.3 WIN32 MULTITHREADING

The Win32 model expands the capabilities of the Win16 model. Threading and security are two of the features that were added to Windows. Windows NT and Windows 95 operate differently when threads are considered. The API for the two operating systems is the same. Windows 95 and 98 ignore several attributes given to a thread. For example, Windows 95 and 98 ignore security attributes. Windows 95 and 98 are designed for home usage so thread/process security is not used in these operating systems. Issues involving security and permissions are usually not a problem in LabVIEW applications. Some issues may surface when using the VI server or DCOM objects.

Windows NT will always operate using preemptive multithreading. The primary design goal of Windows NT was stability. Windows 95 and 98 were designed with backward compatibility in mind. Windows 95 and 98 will run as cooperative multithreaded environments, similar to Windows 3.1. This is a consideration for LabVIEW programmers to remember when they anticipate end users running legacy applications such as Microsoft Office Version 4. Older versions of Microsoft Word (Version 6.0) may be given cooperative multitasking support so they feel at home in a Windows 3.1 environment. Legacy DOS applications may behave the same way. Either way, when Windows 95 drops into a cooperative multitasking mode it will not inform the user that this decision was made. Performance degradation may be seen in Windows 95 environments when legacy applications are run. If users raise issues related to performance of LabVIEW applications on Windows 95 systems, ask if any other applications are being used concurrently with LabVIEW.

Another difference between Windows 95 and NT is hardware accessibility. Windows 95 allows applications to directly access hardware. This allows drivers to have faster access to devices such as GPIB cards and DAQ boards. Windows NT’s model for stability forbids direct hardware access. Drivers must be written specifically to act as operating system components. Many DAQ programmers have observed that hardware access is slower under Windows NT.

9.4 PTHREADS

Many topics on UNIX multithreading are well beyond the scope of this book. The chapter bibliography lists a number of sources that contain additional information. One of the difficulties in writing about UNIX is the number of vendors writing UNIX. The POSIX standard is intended to provide a uniform list of design requirements for vendors to support. This does not translate directly to uniform behavior of UNIX operating systems. Vendors write an operating system that conforms to their interpretation of the specification.

Priority and scheduling are different for Pthreads. Pthreads have defined scheduling policies, round robin; first-in, first-out; and other. The FIFO policy lets a thread execute until it completes its execution or becomes blocked. This policy is multitasking by any other name, because there is no preemption involved. The round-robin policy is preemptive multithreading. Each thread is allowed to execute for a maximum amount of time, a unit referred to as a “quantum.” The time of a quantum
is defined by the vendor’s implementation. The “other” policy has no formal definition in the POSIX standard. This is an option left up to individual vendors. Pthreads expand on a concept used in UNIX called “forking.” A UNIX process may duplicate itself using a fork command. Many UNIX daemons such as Telnet use forking. Forking is not available to the Win32 programmer. A process that generates the fork is called the Parent process, while the process that is created as a result of the fork command is referred to as the Child process. The Child process is used to handle a specific task, and the Parent process typically does nothing but wait for another job request to arrive. This type of multitasking has been used for years in UNIX systems.

As an example of forking, consider a Telnet daemon. When a connection request is received, the process executes a fork. The Telnet daemon is replicated by the system and two copies of the daemon are now running in memory; the Parent process and Child process. The Parent continues to listen on the well-known Telnet port. The Child process takes over for the current Telnet connection and executes as an independent process. If another user requests another Telnet session, a second Child process will be spawned by another fork. The Parent process, the original Telnet daemon, will continue to listen to the well-known Telnet port. The two Child processes will handle the individual Telnet sessions that users requested.

Forking has both advantages and disadvantages. The major disadvantage to forking is that Parent and Child processes are independent processes, and both Parent and Child have their own protected memory space. The advantage to having independent memory spaces is robustness; if the Child process crashes, the Parent process will continue to execute. The disadvantage is that since Parent and Child are independent processes, the operating system must perform process context switches, and this requires additional overhead.

9.5 MULTITHREADING PROBLEMS

This section will outline problems that multithreading can cause. This is an important section and will be referenced in Multithreading Myths. Some of these problems do not occur in LabVIEW, but are included for completeness. Many of these problems can be difficult to diagnose because they occur at the operating system level. The OS does not report to applications that cause these problems. Some issues, such as race conditions and deadlock, cannot be observed by the operating system. The operating system will not be able to determine if this is normal operation or a problem for the code.

It is important to understand that any multitasking system can suffer from these problems, including LabVIEW. This section is intended to discuss these problems relative to LabVIEW’s multithreading abilities. As a programmer, you can create race conditions, priority inversion, starvation, and deadlock in LabVIEW’s VIs when working with VI priorities. We will identify which problems you can cause by poor configuration of LabVIEW’s thread counts.
9.5.1 **Race Conditions**

Many LabVIEW programmers are familiar with the concept of a “Race Condition.” Multithreading in general is susceptible to this problem. Fortunately, when writing LabVIEW code, a programmer will not create a race condition with LabVIEW’s threads. A race condition in multithreaded code happens when one thread requires data that should have been modified by a previous thread. Additional information on LabVIEW race conditions can be found in the LabVIEW documentation or training course materials. LabVIEW execution systems are not susceptible to this problem. The dedicated folks at National Instruments built the threading model used by LabVIEW’s execution engine to properly synchronize and protect data. LabVIEW programmers cannot cause thread based race conditions; however, there is still plenty of room for LabVIEW programmers to create race conditions in their own code.

Thread-based race conditions can be extremely hazardous. If pointer variables are involved, crashes and exception errors are fairly likely. This type of problem can be extremely difficult to diagnose because it is never made clear to the programmer the order in which the threads were executing. Scheduling algorithms make no guarantees when threads get to operate.

9.5.2 **Priority Inversion**

A problem occurs when two threads of different priority require a resource. If the lower-priority thread acquires the resource, the higher-priority process is locked out from using the resource until it is released. Effectively, the higher-priority process has its priority reduced because it must now wait for the lower-priority process to finish. This type of blocking is referred to as *Priority Inversion*. The resource could be in the application, or the resource could be external to the application, such as accessing a shared file on the hard drive. Internal resources include things like variables. External resources include accessing the hard drive, waiting on ActiveX components, and waiting on Kernel-level operating system components such as mutexes and semaphores.

Priority inversion will degrade an application’s performance because high-priority threads do not execute as such. However, the program will still execute properly. Inversion is a problem that can be caused by a LabVIEW programmer who errantly alters the priority level of LabVIEW’s thread pools. Priority levels of “Normal” should be used, and this will prevent priority inversion problems. When threads have no active work to do, they will be blocked.

Most LabVIEW applications should not require modification of priority levels of threads. Errant priority levels can cause a number of problems for a LabVIEW application. An example of when a programmer would consider adjusting the priority level of a subsystem is a high-speed data acquisition program. If the DAQ subsystem required a majority of the CPU time, then a programmer may need to raise priority levels for the DAQ subsystem. If a common queue were used to store data brought in from the DAQ card, priority inversion would occur. VIs performing numerical processing or data display that are resident in the user subsystem will execute with
lower priority than the DAQ threads. This becomes a problem when the user interface threads have access to the queue and spend a lot of time waiting for permission to execute. The DAQ threads trying to put data into the queue become blocked until lower-priority interface threads complete. The lesson to learn here is important: when priority inversion occurs, the high-priority threads end up suffering. This type of problem can seriously impact application performance and accuracy. If the DAQ card’s buffer overflows because the DAQ subsystem was blocked, application accuracy would become questionable.

9.5.3 Starvation

Starvation is essentially the opposite of priority inversion. If access to a resource that two threads need is for very short periods of time, then the higher-priority thread will almost always acquire it before the lower priority thread gets the opportunity to do so. This happens because a higher-priority thread receives more execution time. Statistically, it will get the resource far more often than the lower-priority thread.

Like, priority inversion, “Starvation” is resolved differently by Windows NT and 95. Again, like priority inversion, a LabVIEW programmer can cause this multithreading problem. We will discuss prevention of both priority inversion and starvation issues later in this chapter.

If Windows will actively seek to resolve starvation and priority inversion, then why bother to prevent them from happening? The reason you do not want to cause either problem is that both reduce the efficiency of an application. It is poor programming practice to design an application that requires the operating system to resolve its deficiencies.

If there is a valid reason for a thread to execute at a higher priority than others, then the program design should make sure that its execution time is not limited by priority inversion. Lower-priority threads can suffer from starvation. This is highly undesirable because execution time of the higher-priority threads will become limited while Windows allows the lower threads to catch up. A balanced design in thread priorities is required. In multithreaded code, it is often best just to leave all threads at the same priority.

9.5.4 Deadlocking

Deadlock is the most difficult multithreading problem that is encountered. Deadlock can only occur when two or more threads are using several resources. For example, there are two threads, A and B. There are also two resources, C and D. If both threads need to take ownership of both resources to accomplish their task, deadlock occurs when each thread has one resource and is waiting for the other. Thread A acquires resource C, and thread B acquires resource D. Both threads are blocked until they acquire the other resource.

The bad news as far as deadlocking is concerned: Windows has no mechanism to resolve this type of problem. The operating system will never force a thread to release its resources. Fortunately, deadlocking is highly unlikely to be caused by a LabVIEW programmer. This is a thread-level problem and would be caused by the execution engine of LabVIEW. Once again, the dedicated folks at National Instruments
have thoroughly tested LabVIEW’s engine to verify that this problem does not exist. This eliminates the possibility that you, the programmer, can cause this problem.

9.5.5 Operating System Solutions

Operating systems try to compensate for starvation and priority inversion. Requiring the operating system to compensate is poor programming practice, but here is how UNIX and Win32 try to resolve them.

Priority inversion is resolved differently by Windows 95 and NT. Windows NT will add a random number to the priority of every thread when it orders the active queue. This obviously is not a complete solution, but a well-structured program does not require the operating system to address thread problems.

Windows 95 will actually increase the priority of the entire process. Increasing the priority of the process gives the entire application more of the CPU time. This is not as effective a solution as the one used by NT. The reason for this type of handling in Windows 95 is for backward compatibility. Win16 programs are only aware of one thread of execution. Elevating the entire process’s priority makes Win16 applications feel at home in a Windows 3.1 environment.

9.6 Multithreading Myths

This section discusses some common myths about multithreading. We have heard many of these myths from other LabVIEW programmers. Multithreading is one of the exciting new additions to LabVIEW 5.0; unfortunately, it is also one of the most dangerous. The following myths can lead to performance degradation for either a LabVIEW application or the entire operating system. The single biggest myth surrounding multithreading is that it makes applications run faster. This is entirely false, and the case is outlined below.

9.6.1 The More Threads, the Merrier

The first myth that needs to be addressed is “more threads, better performance.” This is just not in line with reality; application speed is not a function of the number of running threads. When writing code in languages like C++, this is an extremely dangerous position to take. Having a thread of execution for every action is more likely to slow an application down than speed it up. If many of the threads are kept suspended or blocked, then the program is more likely to use memory inefficiently. Either way, with languages like C++ the programmer has a lot of room to cause significant problems.

LabVIEW abstracts the threading model from the programmer. This threading model is a double-edged sword. In cases like the number of threads available, LabVIEW will not always use every thread it has available, and the programmer will just waste memory. When applications are running on smaller computers, such as laptops where memory might be scarce, this could be a problem.

The rule of thumb for how many threads to have is rather vague: do not use too many. It is often better to use fewer threads. A large number of threads will introduce
a lot of overhead for the operating system to track, and performance degradation will eventually set in. If your computer has one CPU, then no matter what threading model you use, only one thread will run at a time. We’ll go into customizing the LabVIEW threading model later in this chapter. We will give guidelines as to how many threads and executions systems programs might want to have.

9.6.2 Always Makes My Program Run Faster

This myth is not as dangerous as “the more threads the merrier,” but still needs to be brought back to reality. The basic problem here is that threads make sections of a program appear to run at the same time. This illusion is not executing the program faster. When the user interface is running in its own thread of execution, the application’s GUI will respond more fluidly. Other tasks will have to stop running when the user interface is updating. When high performance is required, a nicely updated GUI will degrade the speed that other subsystems operate.

The only true way to have a program run faster with multiple threads is to have more than one CPU in the computer. This is not the case for most computers, and, therefore, threads will not always boost application performance. In the world of Windows, this requires Windows NT. Windows 95 and 98 do not support Symmetric Multiprocessing, and will only make use of one CPU in the system.

Later in this chapter, we will show how performance gains can be made without making significant changes to the thread configuration. As a rule of thumb, threads are always a difficult business. Whenever possible, try to tweak performance without working with threads. We will explain the mechanics of multithreading in the next sections, and it will become clear to the reader that multithreading is not a “silver bullet” for slow applications.

9.6.3 Makes Applications More Robust

There is nothing mentioned in any specification regarding threads that lends stability to an application. If anything, writing multithreaded code from a low level is far more difficult than writing single-threaded code. This is not a concern for the LabVIEW programmer because the dedicated folks at National Instruments wrote the low-level details for us. Writing thread-safe code requires detailed design and an intimate understanding of the data structures of a program. The presence of multiple threads does not add to the stability of an application. When a single thread in a multithreaded application throws an exception, or encounters a severe error, it will crash the entire process. In distributed application, it could potentially tear down the entire application. The only way to ensure the stability of a multithreaded application is a significant amount of testing. The folks at National Instruments have done this testing, so there is no need to spend hours doing so to verify your threading models.

9.6.4 Conclusion on Myths

Multithreading gives new abilities to LabVIEW and other programming languages, but there is always a price to be paid for gains. It is important to clearly understand what multithreading does and does not provide. Performance gains will not be
realized with threads running on a single-processor system. Large numbers of threads will slow an application down because of the overhead the system must support for the threads to run. Applications’ graphical interfaces will respond more fluidly because the graphics subsystems’ threads will get periodic time to execute. Intensive computational routines will not block other sections of code from executing.

9.7 MULTITHREADED LABVIEW

Fundamentally, the dataflow operation of LabVIEW is not impacted by multithreading. The real differences are abstracted from the programmer. In this section we discuss the architecture of multithreaded LabVIEW, the main run queue, and how to configure the thread systems. Understanding how threads interact with the LabVIEW subsystems will allow a programmer to use threading effectively. Threads in one subsystem may access VIs you thought were operating in another subsystem. The topic is somewhat abstract, and this section intends to clarify the interactions of VIs and threads. Once the reader understands the subsystem architecture and main run queue, an introduction to thread configuration will be presented.

9.7.1 EXECUTION SUBSYSTEMS

The various activities that LabVIEW performs are handled by six subsystems. The LabVIEW subsystems are User, Standard, I/O, DAQ, Other 1, and Other 2. The original design of LabVIEW 5.0 used these subsystems to perform tasks related to the system’s name. This rigid partitioning did not make it into the release version of LabVIEW 5.0, and tasks can run in any LabVIEW subsystem. Figure 9.3 depicts LabVIEW broken up into its constituent subsystems.

Each subsystem has a pool of threads and task queue associated with it. LabVIEW also maintains a main run queue. The run queue stores a priority-sorted list of tasks that are assigned to the threads in the subsystem. A LabVIEW subsystem has an “array” of threads and priorities. The maximum number of threads that can be created for a subsystem is 40; this is the maximum of 8 threads per priority and 5 priority levels. Section 9.7.4 discusses thread configuration for subsystems. Figure 9.4 shows a subsystem and its array of threads.

FIGURE 9.3 (Courtesy National Instruments)
The User subsystem is the only subsystem that is required for LabVIEW to run, because all other subsystems are optional to running LabVIEW. Configuring thread counts and priorities for subsystems is covered in 9.7.4. The User subsystem maintains the user interface, compiles VIs, and holds the primary thread of execution for LabVIEW. When a DLL or code fragment with questionable thread safety is run, the User subsystem is where it should always be called. Single threaded LabVIEW 5.0 still has one thread of execution, and it rides on this subsystem.

The Standard subsystem was intended to be the default subsystem for LabVIEW executable code. If a programmer is interested in keeping dedicated execution time to the user interface, assign the main level VIs to this subsystem. This will guarantee that the User subsystem threads have plenty of time to keep the display updated. Like the User subsystem, the Standard subsystem can be configured to run with an array of threads.

The DAQ subsystem was originally intended to run data acquisition-specific tasks. It is currently available to any VI. The I/O subsystem was intended for VXI, GPIB, Serial, and IP communications (TCP and UDP). This design is interesting when the programmer considers using the VISA communication suite, VISA is a communications subsystem, and a dedicated thread pool is certainly a good idea. Remember that having dedicated threads guarantees some amount of execution time. On single CPU systems there is still a “borrow from Peter to pay Paul” issue, but communications is fundamental to many LabVIEW applications and justification for guaranteed execution time is sometimes appropriate. The priority levels of a subsystem’s threads relative to the other subsystem threads would serve as a rough gauge of the amount of execution time available. Other 1 and Other 2 were intended for user-specified subsystems. Again, these two subsystems can be used for any purpose desired.

Most applications do not need to assign VIs to specific subsystems. A simple litmus test to decide if a VI should be dedicated to a subsystem is to write a one-paragraph description of why the VI should only be executed in a specific subsystem.
This is a simple Description Of Logic (DOL) statement that should be included in any application design. Description of Logic was discussed in Chapter 4. If the DOL cannot describe what the benefits of the assignment are, then the assignment is not justified. Valid reasons for dedicating a set of VIs to a specific subsystem include the need to guarantee some amount of execution time. If a programmer decides to modularize the execution of an application, then assignment of a VI to a subsystem can be done in VI Setup as shown in Figure 9.5.

When a new VI is created, its default priority is normal and the system is “same as caller.” If all VIs in a call chain are listed as run under “same as caller,” then any of the subsystems could potentially call the VI. The default thread configuration for LabVIEW is to have one thread per subsystem with normal priority. The subsystem that calls a VI will be highly variable; during execution it is impossible to determine which threads were selected by the scheduler to execute. When a VI is assigned to a particular subsystem, only threads belonging to the specified subsystem will execute it.

Now that a simple definition of the LabVIEW subsystems has been presented, let’s consider threads. Subsystem threads execute in a round-robin list and are scheduled by the operating system. When a VI is listed to execute in a specific subsystem, only threads assigned to that subsystem can execute it. As an example, a VI, test_other1.vi is only permitted to be executed by the Other 1 subsystem. When test2_other2.vi is executed and told to call test_other1.vi, the thread that is executing test2_other2.vi will block. The data flow at the call is blocked until a thread from the Other 1 subsystem is available to execute it. This is up to the scheduler of the operating system, and also one of the points where errant priorities can cause priority inversion or starvation. LabVIEW cannot directly switch to a thread in Other 1 to execute the thread. Only the operating system can decide which thread gets to execute.
when. Other 1 threads will not be considered special to the operating system, and they must wait in the thread queue until it is their turn to execute.

When assigning VIs to particular subsystems, use caution when assigning thread priorities. If VIs are called by two subsystems, there should not be large differences in the priorities of the threads belonging to these subsystems. An example is if the subsystem Other 1 has 2 threads at “Above Normal” priority and Other 2 has a “Background” priority thread. If a VI called in subsystem Other 1 calls a subVI in Other 2, not only does the thread in Other 1 block, it has to wait for a background priority thread in Other 2 to get scheduled time to execute! This is a simple example of priority inversion that can be caused by a LabVIEW programmer. If Other 1 and Other 2 were both executing with “Normal” priority, the scheduling algorithm would have mixed the scheduling of the threads and no priority inversion would occur. The thread in Other 1 would have needed to wait on the thread context switches, but those delays are relatively minor.

Another threading problem that can be caused by errant priority settings is starvation. Consider the following case: VIs other1.vi, other2.vi, and io.vi. The Other 1 VI is listed with “time-critical” priority level, Other 2 VI is listed with “background” priority, and io.vi is listed “same as caller.” Same as caller allows a VI to execute in any subsystem with the same priority thread that executed the calling VI. Both other1.vi and other2.vi need to call io.vi. Since other1.vi is running in a subsystem with a time-critical priority thread, it is going to get a significant amount of execution time compared with other2.vi. Access to io.vi will be granted to other1.vi far more often than other2.vi. Other2.vi will become starved because it does not get enough access to io.vi. Scheduling algorithms are notoriously unforgiving when they schedule threads. If a thread is available to execute, it will get put in the active list based entirely on its priority. If a thread with low priority is always available, it will still make the active list, but will always be near the bottom.

### 9.7.2 The Run Queue

LabVIEW maintains several run queues consisting of a main run queue and a run queue for each subsystem. A run queue is simply a priority-ordered list of tasks that are executed. When a VI is executed, the LabVIEW execution engine determines which elements in the block diagram have the needed inputs to be executed. The engine then orders inputs by priority into the run queues. The run queue is not strictly a First In First Out (FIFO) stack. VIs have priorities associated with them (the default priority is “normal”). After execution of each element, the run queue is updated to reflect elements (subVIs or built-in LabVIEW functions, such as addition, subtraction, or string concatenation) that still need to be executed. It is possible for VIs to take precedence over other VIs because they have higher priority. Wildly changing VI priorities will likely result in performance issues with LabVIEW. One key point to understand is that VI priorities are in no way associated with thread priorities. The thread that pulls it off the run queue will execute a VI with high priority. If the thread has background priority, a slow thread will execute the high-importance VI. The execution engine will not take a task away from one thread and reassign it to a thread with a more suitable thread priority.
To help illustrate the use of run queues, consider the In Box on your desk. With LabVIEW 4.1 and earlier, there was a single in box, and each time a VI was able to run it would be put into the in box and you would be able to grab the task and perform it. LabVIEW 5.0 and later has multiple run queues which equates to one in box for each of your hands. As tasks become available they get put into an appropriate in box and the hand that corresponds to that in box can grab the task. Another comparison for Windows programmers is the message pump. Windows 3.1 had a single message loop. Each time an event occurred, such as a mouse click, the event would be put into the system-wide message loop. All applications shared the same message loop, and numerous problems were caused because some applications would not return from the message loop and would lock up windows. Windows 95 and later has message loops for each application. Every application can continue to run regardless of what other applications are doing. We have the same benefit in LabVIEW now. VIs assigned to different subsystems can now operate with their own threads and their own run queues.

A thread will go to the run queue associated with its subsystem and pull the top task off the list. It will then execute this task. Other threads in the subsystem will go to the run queue and take tasks. Again, this is not a FIFO stack — the highest-priority VI will be handed to a thread. This leaves a lot of room for both performance tweaking and performance degradation. Priorities other than “normal” should be the exception, and not status quo.

When a VI is configured to run only in a particular subsystem, it will be put onto the run queue of that particular subsystem, and then the VI must wait for a thread belonging to this system to be assigned to execute it. This can cause performance degradation when thread priorities are different between subsystems. Section 9.8.2 discusses thread configurations in multisubsystem LabVIEW applications.

Figure 9.6 shows the run queues that are developed when a LabVIEW code diagram is run. When VIs are scheduled to run in the “same as caller subsystem,” a thread belonging to the subsystem will end up executing the VI. A subtle point is that if there are multiple threads belonging to the subsystem, there are no guarantees which thread will execute which VI.
9.7.3 DLLs in Multithreaded LabVIEW

Special care must be taken when working with DLLs in multithreaded LabVIEW. DLLs can potentially be called from several different threads in LabVIEW. If the DLL has not been written to handle access by multiple threads, it will likely cause problems during execution. Recall thread safe in Section 9.1.10. If mutexes, semaphores, or critical sections are not explicitly designed into the DLL, then it is not guaranteed to be thread safe.

Threading problems are not always obvious. Several million calls may need to be made to the DLL before a problem surfaces. This makes troubleshooting thread problems extremely difficult. Bizarre program operation can be extremely difficult to troubleshoot, especially when the code can execute for days at a time without failure. When working with DLLs and crashes occur only occasionally, suspect a thread problem.

When writing C/C++ code to be called by LabVIEW, you need know if it will possibly be called by multiple threads of execution. If so, then you need to include appropriate protection for the code. It is fairly simple to provide complete coverage for small functions. The Include file that is needed in Visual C++ 5.0 and 6.0 is process.h. This file contains the definitions for Critical Sections, Mutexes, and Semaphores. This example is fairly simple and will use Critical Sections for data protection. A Critical Section is used to prevent multiple threads from running a defined block of code. Internal data items are protected because their access is within these defined blocks of code. Critical Sections are the easiest thread protection mechanism available to the Windows programmer, and their use should be considered first.

```c
#include <process.h>

//Sample code fragment for CriticalSections to be used by a //LabVIEW function.
CRITICAL_SECTION Protect_Foo
Void Initialize_Protection(void)
{  
    INITIALIZE_CRITICAL_SECTION(&Protect_Foo);
}

Void Destroy_Protection(void)
{  
    DELETE_CRITICAL_SECTION(&Protect_Foo);
}

int foo (int test)
{  
    int special_Value;
    ENTER_CRITICAL_SECTION(&Protect_Foo);  //Block other threads from accessing
    Special_Value = Use_Values_That_Need_Protection(void);
    LEAVE_CRITICAL_SECTION(&Protect_Foo);  //Let other threads access Special Value, I’m finished.
    Retun special_Value;
}
```

©2001 CRC Press LLC
The fragment above does not do a lot of useful work as far as most programmers are concerned, but it does illustrate how easy thread protection can be added. When working with Critical Sections, they must be initialized prior to use. The INITIALIZE_CRITICAL_SECTION must be called. The argument to this function is a reference to the Critical Section being initialized. Compile errors will result if the Critical Section itself is passed. The Critical Section must also be destroyed when it will no longer be used. Initialization and destruction should be done at the beginning and end of the application, not during normal execution.

Using a Critical Section requires that you call the Enter and Leave functions. The functions are going to make the assumption that the Critical Section that is being passed was previously initialized. It is important to know that once a thread has entered a Critical Section, no other threads can access this block until the first thread calls the Leave function.

If the functions being protected include time-consuming tasks, then perhaps the location of the Critical Section boundaries should be moved to areas that access things like data members. Local variables do not require thread protection. Local variables exist on the call stack of the thread and each thread has a private call stack, which makes local variables completely invisible to other threads.

C++ programmers must also remember that LabVIEW uses C naming conventions. The keyword `extern C` must be used when object methods are being exposed to LabVIEW. LabVIEW does not guarantee that C++ DLLs will work with LabVIEW. In the event you encounter severe problems getting C++ DLLs to operate with LabVIEW, you may have hit a problem that cannot be resolved.

A few additional notes on DLLs being called from LabVIEW before we complete this section. Windows 3.1 DLLs must be 16-bit DLLs. Considering that Windows 3.1 is a 16-bit programming interface, this rule seems to make sense. LabVIEW for Windows NT, 95, and 98 can only call DLLs compiled as 32-bit applications. If 16-bit DLLs need to be called, then you must write a “wrapper DLL” that is compiled as a 32-bit DLL, which then calls your legacy 16-bit DLL. This involves a process called “thunking.” There is a performance hit that occurs when calling 16-bit DLLs. High-performance applications may need old DLLs to be rebuilt.

LabVIEW 5.0 uses color-coding to identify DLLs that are executed by the user interface thread from DLLs that are listed as “reentrant.” If a DLL call is shown in an orange icon, this identifies a DLL call that will be made from the User Interface subsystem. If the standard off-yellow color is shown, it will be considered reentrant by LabVIEW and will allow multiple threads to call the DLL. Library call functions default to User Interface subsystem calls. If a DLL was written to be thread safe, then changing this option to reentrant will help improve performance. When a User Interface Only DLL call is made, execution of the DLL will wait until the user interface thread is available to execute the call. If the DLL has time-consuming operations to perform, the user interface’s performance will degrade.

When working with DLLs of questionable thread safety, always call them from the user interface. When it is known that threading protection has been built into a DLL, make the library call reentrant. This will allow multiple threads to call the DLL and not cause performance limitations. If you are stuck with a DLL that is not known to be thread safe, be careful when calling the DLL from a loop. The number
of thread context switches will be increased, and performance degradation may set in. We did get a tip from Steve Rogers, one of the LabVIEW development team members, on how to minimize the number of context switches for DLLs. This tip works when you are repeatedly calling a DLL from a loop that is not assigned to the user subsystem. Wrap the DLL call in a VI that is assigned to the user subsystem. The thread context switches have been moved to the VI call and not the DLL call. Effectively, this means that the entire loop will execute in the user subsystem, and the number of needed context switches will drop dramatically.

Execution of a DLL still blocks LabVIEW’s multitasking. The thread that begins executing the DLL will not perform other operations until the DLL call has completed. Unlike LabVIEW 4.0 and earlier, other threads in the subsystem will continue to perform tasks.

### 9.7.4 Customizing the Thread Configuration

The number of threads in each LabVIEW execution subsystem is specified in the labview.ini file. This information should not be routinely altered for development workstations. The default configuration will work just fine for most applications. The default configuration for a multithreaded platform is one thread of execution per subsystem of normal priority.

In situations where a LabVIEW application is running on a dedicated machine, tweaking the thread configuration can be considered. National Instruments provides a useful VI for configuring the ini file used by LabVIEW: threadconf.vi. The Thread Configuration VI should be used to alter LabVIEW’s thread configuration. Vendors are expected to supply tools to easily modify application configurations. The most difficult aspect of using this VI is locating it! The VI can be found in vi.lib/utilities/sysinfo.llb. Figure 9.7 shows the front panel of this VI.

To change the configuration of the thread engine, select Configure. A second dialog box will appear, as shown in Figure 9.8. Each execution subsystem may be configured for up to eight threads per priority. Some readers may feel compelled to review Section 9.5. A plethora of options are available, and the advice is not to alter most of them. In the next section, information on estimating the maximum number of useful threads is presented.

A few words about thread priorities — time-critical priorities are, in general, hazardous. Consider Windows NT and time-critical threads. Mouse clicks and keypad activity is reported to the operating system; the operating system then sends a message to the application that should receive the user input. In the event that time-critical threads are running, they may take priority over operating system threads. The commands you enter to Quit may never arrive, and the machine will need to be power-cycled.

Background priorities may be useful, but, in general, keeping all threads at normal priority is best. If all threads are running on normal priority, none of them will suffer from starvation or priority inversion. The easiest way to avoid complex threading issues is to avoid creating them. This is the biggest caveat in this section; review Section 9.5 for descriptions on priority inversion and starvation. If more threads are available than can be used, LabVIEW’s execution engine will allow the
thread to continue checking the queue for VIs that belong to its subsystem. This will require a minimal amount of CPU time and avoids thread problems.

LabVIEW Versions 5.0 and 5.1 operate somewhat differently when it comes to application startup. LabVIEW 5.0 will create all of the threads specified in the LabVIEW ini file at startup. LabVIEW 5.1 will start only the threads specified for the user subsystem. The reason for the difference is system resource usage. We do not want to create more threads than necessary because that would cause extra
overhead for the operating system to track. We also do not want to create threads during normal application execution. Thread creation takes a quantifiable amount of time, on the order of several milliseconds. This may seem like a minor amount of time, but the thread that is requesting creation of the second thread is halted during this time. Applications requiring high performance would certainly be disappointed with this type of behavior. LabVIEW 5.1 will create threads for other subsystems when it loads a VI that is assigned to a subsystem into memory. The LabVIEW INI file specifies the number of threads created for any particular subsystem. All threads specified will be created, regardless if they will all be used. This is really not a problem for a vast majority of programmers. The act of loading a VI into memory takes significantly longer than starting new threads of execution. The amount of time involved is therefore negligible to start up new threads. Generally, starting new threads during execution is undesirable for high-performance applications, but since we are also performing file access, creating threads at this time is not an issue.

Windows users may see an additional thread created when dialog boxes are used. The dialog boxes for Printers, Save File, Save File As, and Color Dialog are actually the same dialog box. This dialog box is called the common dialog and is responsible for creating that extra thread. This is normal operation for Windows 95, 98, and NT. The common dialog thread is created by the dialog box, and is also destroyed by the dialog box. High-performance applications should refrain from displaying dialog boxes during execution except when absolutely necessary.

When working with the Application Builder, the thread information will be stored in a file that is located with the executable generated by LabVIEW. The same rules apply to application builder programs as mentioned above. Normal priority threads are all that a programmer should ever need; do not create more threads than an application can use. Recall that the existence of threads only translates to performance gains when multiple processors are available on the machine.

Another word of caution when configuring the threads LabVIEW uses. Since LabVIEW is an application like any other running on the system, its threads are scheduled like any other, including many operating system components. When LabVIEW threads are all high priority, they just might steal too much time from the operating system. This could cause system-wide inefficiency for the operating system, and the performance of the entire computer will become degraded or possibly unstable. The fundamental lesson when working with thread priorities is that your application is not the only one running on the system, even if it is the only application you started.

9.8 THREAD COUNT ESTIMATION FOR LABVIEW

When discussing thread count, this section will refer exclusively to the maximum number of useful threads, which is the number of threads that will be of benefit to the execution systems without any threads being left idle. The minimum number of useful threads to the LabVIEW execution engine is always one. If one thread of execution is specified, then LabVIEW 5.0 will behave very much like LabVIEW 4.0.
Consider the VI code diagram presented in Figure 9.9. The number of useful threads for this VI is one. This VI obviously does not consider events such as errors, fewer than 100 bytes read, or the spreadsheet String-to-Array function. In spite of the fact that it is not well-thought-out, it is an excellent dataflow example. Everything will happen in the VI in a well-established order: the TCP Read function will execute, and then the While loop will execute. There is only one path of execution, and multiple threads will do nothing to optimize the execution of this VI. The fact that the While loop will execute multiple times does not suggest multiple threads can help. It does not matter which thread in which subsystem is currently processing this VI, execution will still happen in a defined order. Thread context switches will do nothing to help here. The lesson learned in this simple example: if order of execution is maintained, multithreading is not going to improve application performance.

Alert readers would have noticed the location of the String Length function. In the While loop it is possible for two threads to perform work, but one will only need to return a string length, which is a very simple function. This is not a significant amount of work for a thread to do. Also, it would be far more efficient to locate the String Length function outside the While loop and feed the result into the loop. When optimizing code for threading, look for all performance enhancements, not just the ones that impact threading potential. Having an additional thread for this example will not improve performance as much as moving the string length outside the While loop.

The VI code diagram presented in Figure 9.10 presents a different story. The multiple loops shown provide different paths of execution to follow. Threads can execute each loop and help maintain the illusion that all loops seem to execute at the same time. If multiple CPUs are involved, application speed will improve. The number of useful threads for this example is three. If the internal operations of the loops are time-intensive operations, then a fourth thread may be desirable. This fourth thread will be added to help support the front panel. If there are graphs or other intensive operations, consider an additional thread for supporting the display. Recall that additional threads will take some time away from other threads.

Now consider the VI code diagram shown in Figure 9.11. There appears to be a single path for this VI to take, but considering that several of the VIs are not waiting on inputs, they can be scheduled to run right away. There are four paths of execution that merge into one at the end of the code diagram. Threads can help here, and the maximum number of useful threads will be equal to the number of paths of execution. In this case, the maximum number is four. If several of these subVIs are...
expected to execute quickly, then they do not require a thread to exist on their behalf, and you should consider reducing the number of threads. Recall that each thread is going to be scheduled and requires some time on the CPU. The lower the thread count, the more execution time per thread.

The code diagram presented in Figure 9.12 is basically a mess. Multiple threads could potentially be beneficial here, but if the operations splattered about the display were modularly grouped into subVIs, then the benefit seen in Figure 9.11 would still exist. You can consider prioritizing the subVIs as subroutines; the benefit of reduced overhead would make the VI run nearly as fast, and a lot of readability will be gained. Section 9.9 describes criteria for using subroutine VIs in multithreaded LabVIEW.

We have gone through a few simple examples concerning only a single VI. When a large-scale application is going in development, the maximum number of useful threads will probably not skyrocket, but the determination can be much more difficult. An application consisting of 250 subVIs will be time-intensive for this type of analysis. The programmer’s intuition will come into play for application wide analysis. Also, never forget that at some point, adding threads is not going to make an
improvement. Unless you are working with a quad-CPU system, having 200 threads of execution is not going to buy much in terms of performance!

9.8.1 Same as Caller or Single Subsystem Applications

When attempting to determine the maximum number of useful threads for an application, the maximum number of execution paths for the application must be determined. This can be difficult to accomplish. For example, look at the hierarchy window of a medium to large-scale application. Each branch of a hierarchy window does not equate to one branch of execution. A programmer who is familiar with the functionality of each subVI will have an understanding of the tasks performed in the subVI. Look at subVIs that have descriptions that suggest parallel operations. This is a difficult piece of advice to generalize, but the programmer should be familiar with their application. Order of execution may be forced in a number of branches, and this will limit the number of useful threads. Having more threads than necessary will cause minor performance hits. The number of thread context switches that will be incurred when threads are given time will be increased. If the thread configuration includes threads of differing priority, then lower-priority threads may receive little execution time and not be of much help to the application.

The simplest case to analyze is when all threads are running in a single LabVIEW subsystem or all VIs are assigned to a “same as caller” subsystem. Then there are no needed considerations to be made regarding which subsystems require threads. On a system dedicated to running an application of this type, consider modifying the thread configuration so that only one subsystem has threads — the User subsystem.

The following VI code diagram simply demonstrates a main level VI and its three independent loops. Obviously, three threads may support this VI. If the three loops require heavy numerical processing, then a fourth thread may be desired if a lot of display updates are also desired. Since the three subVIs are going to keep busy running numerical calculations, a fourth thread could be brought in for GUI updates. Understand that LabVIEW is not going to allocate a thread to each loop and the fourth to the VI, but there will always be four threads looking into the run queue for a new task. If threads take a lot of time to complete one iteration of a
loop, then three threads may periodically become bogged down in a loop. The fourth thread exists to help out when circumstances like this arise. When no intensive GUI updates are required, the fourth thread is not desirable. Additional thread context switches can be avoided to improve performance.

Reconsidering the above example, if one of the loops performs a very simple operation, then reducing the number of threads to two may also be beneficial. Having fewer threads means less work for the operating system to do. This is a judgment call the programmer is going to have to consider. The fundamental trade-off is going to be parallel operation versus operating system overhead. The general guideline is to have fewer threads and minimize overhead. When looking to estimate the number of threads, look for operations that are time-consuming. Examples of time-consuming operations are large array manipulation, DLL calls, and slow data communications, such as serial ports.

9.8.2 Multiple Subsystem Applications

Determining how many threads can support a LabVIEW application with VIs running in dedicated subsystems requires additional work. The number of useful threads per subsystem must now be considered. The solution to this problem is to analyze the number of paths of execution per subsystem. Considerations must be made that threads may become blocked while waiting for VIs to be assigned to other subsystems. Again, an additional thread may be considered to help out with display updates. Do not forget that an additional thread will take some time away from other threads in LabVIEW. High-performance applications may still need to refrain from displaying graphs during run-time.

It is still possible to write many multithreading-optimized applications without resorting to using multiple subsystems. LabVIEW’s configuration allows for a maximum of eight threads per subsystem. When you conclude that the maximum number of useful threads is well beyond eight, then forcing some VIs to execute in different subsystems should be considered. If fewer than nine threads can handle a VI, do not force multiple subsystem execution. Performance limitations could arise with the extra contact switching.

A special case of the multiple subsystem application is a distributed LabVIEW application. Optimization of this application should be handled in two distinct parts. Since LabVIEW is executing independently on two different machines, you have two independent applications to optimize. Each machine running LabVIEW has two separate processes and each will have their own versions of subsystems. When the threading model is being customized, each machine should have its own threading configuration. One machine may be used solely for a user-interface, and the other machine may be executing test or control code. Consider using the standard configuration for the user interface machine. It is unlikely that sophisticated analysis of the user interface is required. Consider investing engineering time in the more important task of the control code. In situations where each instance of LabVIEW is performing some hard-core, mission-critical control code, both instances of LabVIEW may have their threading configurations customized.
Your group should deploy a coding standard to indicate information regarding the subsystem a VI is assigned to. When trying to identify problems in an application, the subsystem a VI is assigned to is not obvious. A programmer must actively look for information regarding that. A note above or below the VI should clearly indicate that the VI has been forced into a subsystem. An alternative to using notes is to color-code the icon, or portion of it, to clearly indicate that the VI has been forced into a nonstandard mode of execution. This will simplify debugging and maintenance. Multiple subsystem applications will almost always be very large-scale applications; these types of techniques will simplify maintenance of such large applications.

### 9.8.3 Optimizing VIs for Threading

When you are writing code for which you would like to have the maximum benefit of the threading engine, avoid forcing the order of execution whenever possible. When a VI is coded for tasks to happen in a single-file fashion, the tasks assigned to the run queue must also be assigned in a single-file fashion. This limits the ability of the threads to handle tasks because they will always be waiting for a task to become available. If possible, avoid the use of sequences; they are going to force an order of execution. Obviously, the sequence diagram was put in the code for a reason. Let the error clusters force an order of execution for things like read and write operations. Operations that are handled, such as loading strings into a VI and determining the value of some inputs, can be done in a very parallel fashion. This will maximize the ability of the threads to handle their jobs. All simple operations will have their data available and will be scheduled to run.

As an example of maximizing dataflow, consider the code diagram in Figures 9.13, 9.14, and 9.15. These three diagrams describe three sequences for a simple data acquisition program. The items in the first sequence must be handled and completed before the second can be executed. The second sequence is fairly simple, and the waveform is shipped out. The third sequence reads in a signal and filters it. The DAQ experts may criticize the appearance of this VI, but it serves as an example of how sequences limit the thread’s ability to operate.

In the first sequence there are two paths of execution to follow. The first is the generation of the sine waveform to be used. The second path to follow is the Analog Output and Analog Input VIs. Please note that the error cluster forces an order of execution; the Output VI must be executed, then the Input VI. There is some initial loading of values on the wire table that needs to be done. The threads will also handle this.

The second sequence diagram simply sends out the waveform. The inputs here cannot be processed and moved on the wire table until this sequence starts executing. Had this VI been in the first sequence, the constants could have already been shifted in LabVIEW’s wire table.

The third sequence reads an input waveform and runs it through a Butterworth filter. Many DAQ experts will argue about the timing delays and choice of a Butterworth filter, but we are putting emphasis on the threading issues. The constants in this sequence also may not be loaded into new sections of the wire diagram until this sequence begins execution.
Let us quickly rethink our position on the number of paths that could be followed in the first sequence. Two was the decided number, one for the signal generation, and one for the Configuration VIs. Recall the Setup VIs have multiple subVIs with the possibility of dozens of internal paths. We are unable to maximize the number of executable paths because the order of execution is strongly forced.

The “thread friendly” version is shown in Figure 9.16. Wrapping the Output Generation VI in a sequence was all that was needed to force the Configuration, Generation, and Read functions. The one-step sequence cannot execute until the error cluster output becomes available.

The Configuration VIs are set in parallel with a little VI inserted to add any errors seen in the clusters. This is a handy little VI that is included on the companion CD to this book. The multiple execution paths internal to these VIs are now available to the threading engine.
All constants on the block diagram can be loaded into appropriate slots on the wire table without waiting for any sequences to start. Any of these functions can be encapsulated into subVIs to make readability easier. VIs that easily fit on a 17-inch monitor should not require 46-inch flat-panel displays for viewing after modification.

The lesson of this example is fairly simple, do not force order of execution in multithreaded LabVIEW. If you want to take full advantages of the threading engine, you need to leave the engine a little room to have execution paths. Obviously, some order must exist in a VI, but leave as many execution paths as possible.

This next part of optimization has less to do with threads than the above example, but will stress good programming practice. Polling loops should be minimized or eliminated whenever possible. Polling loops involve some kind of While loop continuously checking for an event to happen. Every time this loop is executed, CPU cycles are burned while looking for an event. In LabVIEW 4.1 and earlier versions,
you may have noticed that the CPU usage of your machine ran up to 100%. That is because the polling loop was “tight.” Tight loops do very little in a cycle. This allows the loop to complete its execution quickly and take more time. Because there is always something in LabVIEW’s run queue to do (run the loop again), LabVIEW appears to be a very busy application to the system. LabVIEW will get all kinds of time from the scheduling algorithms, and the performance of the rest of the system may suffer. In LabVIEW 5.0 the threads that are assigned tasks for the loop will be just as busy, and therefore make LabVIEW again look like a very busy application. Once more, LabVIEW is going to get all kinds of time from the operating system which will degrade performance of the system.

Figure 9.17 shows a simple loop that increments a counter. This is a short example of a tight loop. There is very little activity going on inside the loop, and millions of iterations happen in very little time. If the value of the loop iterator is wired to a terminal, the execution speed will slow down because of the volume of graphics updates that need to happen. The System Monitor (Windows 95), Task Manager (Windows NT), or an application such as Top or Monitor (UNIX) will show a significant amount of CPU usage. The problem is there isn’t much useful happening, but the amount of CPU usage will top out the processor. Other applications will still get time to run on the CPU, but they will not receive as much time because the tight loop will appear to always need time to run.

Tight loops and polling loops cannot always be avoided. When an application is looking for an external event, polling may be the only way to go. When this is the case, use a Wait Milliseconds command if possible. It is unlikely that every polling loop needs to check a value at CPU speeds. If this is the case, the selection of LabVIEW may only be appropriate on Concurrent PowerMAX systems, which are real-time operating systems. If the event does not absolutely need to be detected, make the loop sleep for a millisecond. This will drastically reduce the CPU utilization. The thread executing the wait will effectively be useless to LabVIEW while sleeping, but LabVIEW’s other threads do not need to fight for CPU time while the thread is executing the tight loop.

When waiting on events that will be generated from within LabVIEW, using polling loops is an inefficient practice. Occurrence programming should be used for this. This will prevent any of LabVIEW’s threads from sitting in polling loops. The code that is waiting on an occurrence will not execute until the occurrence is triggered. No CPU cycles are used on nonexecuting code.
9.8.4 **Using VI Priorities**

The priority at which the VI executes may also be considered when configuring VIs. The VI execution priority is not directly related to the threads that execute the VI, or the priority of the threads that execute the VI. Figure 9.18 shows the configuration of a VI. The priority levels assigned to a VI are used when LabVIEW schedules tasks in the run queue. The priority of the VI has nothing to do with the priority of the thread that is executing it. When a thread with high priority is assigned a VI with background priority, the thread will not reduce its priority to accommodate the VI. The background importance VI will be executed with blazing speed. The reverse is also true.

When working with VI priorities, recall multithreading problem definitions; several of them can be caused in LabVIEW’s scheduling routines. Starvation is the easiest problem to cause. When a VI is listed as background priority, VIs with higher priorities will be put into the run queue ahead of the low-priority VI. This will cause the execution of the low-priority VI to be delayed. This could impact the performance of the code diagram. What will end up happening is that all other eligible tasks will be run until the low-priority VI is the only available task to be executed. This would form a bottleneck in the code diagram, potentially degrading performance. The use of VI priorities should not be used to force the order of execution. Techniques using error clusters should be used instead. LabVIEW’s engine makes no promises regarding execution time, much like a multithreaded operating system’s scheduling algorithm. In the event that parallel executing loops are involved, it is possible for the background priority VI to never be executed.

Priority inversion can also be caused by VI priorities. Recall that the priority of a VI does not impact or change the priority of the thread(s) executing it. If a VI with high priority depends on the outputs of a VI with lower priority, execution of the high-priority VI will be delayed until the low-priority VI has completed execution. This potential for performance limitations should be avoided.
Race conditions can also be induced with VI priorities. The threading model used does not induce these race conditions. These would be race conditions caused by the code diagram itself.

The best logic to use to prevent VI priority problems is similar to preventing problems with the threading engine. A priority other than “normal” should be an exception, not the norm. If a convincing case cannot be put into the Description of Logic of the VI, then its execution priority should be normal. In general, we avoid changing VI priorities. Forcing the order of execution is a better mechanism to accomplish control of a code diagram. In addition, it is much easier for a programmer to look at a VI and understand that the order of execution is forced. VI priority is somewhat hidden in the VI’s configuration; a programmer must actively search for this information. Assuming that programmers will examine your code and search the configuration is unwise; most people would not suspect problems with VI priority.

As a coding standard, when a VI has an altered priority, a note should be located above or below to clearly indicate to others who may use the VI that there is something different about it. Another flag that may be used is to color-code the icon or portion of the icon indicating that its priority is something other than normal.

If you absolutely insist on keeping a VI as a priority other than normal, then use the following tip from Steve Rogers (LabVIEW developer extraordinaire): VIs of high priority should never be executing continuously. High-priority VIs should be kept in a suspended mode, waiting on something such as an occurrence, before they are allowed to execute. Once the VI completes executing, it should be suspended again and wait for the next occurrence to happen. This allows for the high-priority VI to execute as the most important VI when it has valid data to process, and to not execute at all when it is not needed. This will prevent programmers from creating priority inversion or starvation issues with LabVIEW’s run queue management.

### 9.9 Subroutines in LabVIEW

As hinted to throughout the chapter, subroutine VIs have strong advantages when using multithreading. First, we need to review the rules on subroutine priority VIs:

1. Subroutine VIs may not have a user interface.
2. Subroutine VIs may only call other subroutine-priority VIs.
3. Subroutines may not call asynchronous nodes (dialog boxes, for example; nodes that do not have a guaranteed return time).

It is important to understand that subroutine classification is not a true priority. “Subroutine” denotes that this VI is no longer a standard VI and that its execution and compilation are radically different from other VIs. Subroutine priority VIs do not have a priority associated with them, and they are never placed into the run queues of LabVIEW. Once all inputs for the subroutine are available, the subroutine will execute immediately, bypassing all run queues. The subsystem associated with the subroutine will stop processing tasks until the subroutine has completed execution. This might sound like a bad idea, but it is not. Having a routine complete execution ASAP is going to get its operation over as quickly as possible and allow
LabVIEW to do other things fairly quickly. Subroutines are a bad idea when very time-intensive tasks need to be done because you will block the run queue for a subsystem for an extended amount of time.

Subroutines execute faster than standard VIs because they use less overhead to represent instructions. You may not have a user interface on subroutine priority VIs because, technically, a subroutine does not have a user interface. This is part of the reduced overhead that subroutines have.

Subroutine VIs may only call other subroutines because they are executed in an atomic fashion. Once execution of a subroutine VI starts, single-threaded LabVIEW execution engines will not do anything else until this subroutine has finished. Multitasking becomes blocked in single-threaded LabVIEW environments. Multithreaded LabVIEW environments will continue multitasking when one thread enters a subroutine. The thread assigned to work on the subroutine may do nothing else until the subroutine is executed. Other threads in the system are free to pull jobs off the run queue. In the next section, we will discuss the data types that LabVIEW supports; this is relevant material when subroutine VIs are considered.

### 9.9.1 LabVIEW Data Types

Every LabVIEW programmer is familiar with the basic data types LabVIEW supports. This section introduces the low-level details on variables and data storage. Table 9.1 shows the LabVIEW data types. Of concern for application performance is how fast LabVIEW can process the various data types. Most numerical processing can always be assumed to be relatively fast.

As stated in Table 9.1, Booleans are simply 16-bit integers in LabVIEW 4.0 and earlier, and 8-bit integers in LabVIEW 5.0 and later. Their storage and creation is fairly quick; arrays of Booleans can be used with minimal memory requirements. It must be noted that current computers are minimum 32-bit machines. Four bytes is the minimum amount of memory that can be addressed at a time. One- and two-byte storage is still addressed as four-byte blocks, and the upper blocks are ignored.

Integer sizes obviously depend on byte, word, or long word selections. Integer arithmetic is the fastest numerical processing possible in modern hardware. We will show in the next section that it is advantageous to perform integer processing in one thread of execution.

Floating-point numbers also support three precision formats. Single- and double-precision numbers are represented with 32- or 64-bit numbers internal to LabVIEW. Extended precision floating-point numbers have sizes dependent on the platform you are using. Execution speed will vary with the types of operations performed. Extended-precision numbers are slower than double-precision, which are slower than single-precision numbers. Floating-point calculations are always slower than integer arithmetic. The selection of precision for floating point numbers needs to be determined based on the numerical accuracy required and the execution speed needed. Each floating point stores sections of a number in various parts of the memory allocated. For example, one bit is used to store the sign of the number, several bytes will be used to store the mantissa, one byte will store the sign of the exponent, and the rest will store the integer exponent of the number. The format for single- and
double-precision numbers is determined by National Instruments, and they are represented internally in LabVIEW. Extended-precision number formats depend on the hardware supporting your system.

Complex numbers use a pair of floating-point numbers for representation. Complex numbers use the same precision as floating-point numbers, but they are slower for processing. Each complex multiplication involves four floating-point calculations. Additions and subtractions involve two floating-point calculations. When necessary, complex calculations need to be done, but their execution speed must be considered in performance-critical applications.

String processing can be very slow. LabVIEW uses four bytes to indicate the length of the string internally, and the contents of the string following the length preamble. This is an advantage LabVIEW programmers have over their C counterparts. C style strings must end with an ASCII 0 (NULL); these NULL-terminated strings assume that there are no NULL values occurring in the middle of the string. LabVIEW strings do not have this requirement. This is advantageous when working with devices such as serial instruments.

Any time you perform an operation on a string, a duplication of the string will be performed. In terms of C programming, this will involve a “memcpy.” Memory copies involve requesting an allocation of memory from the memory manager and then duplicating the memory used. This is a performance hit and, although it cannot be entirely avoided, performance hits can be minimized. Whenever possible, major string manipulation should be avoided when application performance is required. Examine Figure 9.19 for an illustration for where memory copies are made. Memory
copies will be made for other variable types, but sizes for integers are 4 bytes, floating points are a maximum of 8 bytes, and booleans require a minimum of 32 bits for storage. The shortest string representation in LabVIEW is an empty string, which requires five bytes, the four-byte preamble, and one blank byte. Most strings contain information, and longer strings require more time to copy internally.

Array processing can be significantly faster than string processing, but can also be hazardous to application performance. When using arrays in performance-critical applications, predimension the array and then insert values into it. When predimensioning arrays, an initial memory allocation will be performed. This prevents LabVIEW from needing to perform additional allocations, which will cause performance degradation. Figure 9.20 illustrates two array-handling routines. Array copying can be as CPU-intensive as string manipulation. Array variables have four bytes for storage of the array dimensions, and a number of bytes equivalent to the size of the dimensions times the storage size of the type.

---

**TABLE 9.1**
LabVIEW Data Types

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Size</th>
<th>Processing Speed</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>16 bits (LabVIEW 4) 8 bits (LabVIEW 5), high bit determines true/false</td>
<td>Fast</td>
<td>High bit determines true or false.</td>
</tr>
<tr>
<td>Integers</td>
<td>8, 16, or 32 bits</td>
<td>Fast</td>
<td>Signed and Unsigned</td>
</tr>
<tr>
<td>Floating Point</td>
<td>Depends on type and platform</td>
<td>Fast</td>
<td>Extended precision size is machine-dependent; single and double are 32- and 64-bit numbers</td>
</tr>
<tr>
<td>Complex</td>
<td>Depends on type and platform</td>
<td>Medium?</td>
<td>Slower than floating points.</td>
</tr>
<tr>
<td>String</td>
<td>4 bytes + length of string</td>
<td>Slow</td>
<td>First 4 bytes identify length of string.</td>
</tr>
<tr>
<td>Array</td>
<td>Variable on type</td>
<td>Slow</td>
<td>Faster than strings, but can be slow, especially when the array is dimensioned often.</td>
</tr>
<tr>
<td>Cluster</td>
<td>Depends on contents</td>
<td>Slow</td>
<td>Processing speed depends heavily on contents of cluster.</td>
</tr>
</tbody>
</table>

---

**9.9.2 WHEN TO USE SUBROUTINES**

Now that we know the benefits and penalties of using threads and are familiar with implications of data type choices, it is time to determine when subroutines should be used. Knowing that numerical operations are fast and that string and dynamic arrays are slow, the obvious conclusion is that numerical-intensive VIs are prime candidates for subroutine priority.
When handling strings or variable-dimension arrays, do not use subroutines. When LabVIEW tries to copy strings, the thread performing the copying will incur overhead while the memory allocation is being performed. If this thread is running in a subroutine, other threads may become blocked waiting for one thread to cycle through the subroutine. It is preferable in these situations to have multiple threads working in a VI to minimize blocking points.

Numerical processing is fairly fast; it is possible for threads to completely execute most subroutines in a single timeslice. The following example demonstrates calculation speeds to help illustrate the volume of computations a modern CPU can handle.

How many integer additions can a thread execute in a 10-ms timeslice? Assume that the CPU runs at 400 MHz. Integer operations require 3 clock cycles, and access to the integers require an additional 50 clock cycles. The 50 extra clock cycles are needed for access to extended memory.

Solution: A 400-MHz CPU will execute a clock cycle in 1/400 000 000 seconds. This is 0.0000000025 seconds, and 53 clock cycles require 53 * 0.0000000025 = 0.0000001375. If one timeslice is 10 ms, then 10 ms/1.375 ns = 75,471 arithmetic operations!

This example does not correlate well with reality, but it demonstrates the magnitude of numbers that can be crunched during a single timeslice. We are ignoring the threading model; what if the thread’s quantum expired and the thread was booted off the CPU? The 50-clock cycle access to memory was an assumption, but should be fairly close to the amount of activity needed to access extended memory. Floating-point calculations require a significantly higher number of CPU cycles to calculate. Assuming the calculation was an order of magnitude slower, that would still result in over 7,000 floating-point operations in a timeslice. Memory copying requires even more time because the heap manager becomes involved in the memcopy. This is why it is suggested that strings and arrays be handled in fully-threaded VIs. As an example of this, we will consider the following problem: the VI depicted in Figure 9.21 shows a simple string-generation routine. The counter is converted to a string and then concatenated to another string and fed through the shift register. Consider the copying and memory allocations that need to be done. Every time the integer is converted to a string, five to six bytes are allocated from heap memory to store the number; five bytes are used when the number is between zero and nine, and six bytes when the number has two digits. Recall that four bytes of length information are stored in a string preamble. The string concatenation requires an additional allocation of length: four bytes + length of old string + length of new string. These allocations are relatively small but add overhead. Figure 9.22 shows the VI profile from numerous runs of this VI. Execution was performed on a Pentium MMX 200 MHz CPU running Windows 95 with 64 MB of memory.

The timing profile demonstrates that at least one execution required 15 ms to complete. In terms of CPU time, this is significant. A thread may not have enough time in a quantum to always complete this operation. The thread may then be preempted for higher-priority threads and take some time before it can resume execution. This is what happened when the 15 ms sample occurred. During at least one execution, the thread was preempted and took an order of magnitude more time.
to complete. Larger string manipulation routines will even take longer. If a single thread is dedicated to performing all the manipulations, this could reduce perfor-
mance of the application. Outputs of this VI will probably be required by other VIs. These other VIs would be blocked from execution while this current VI is completed. The conclusion this example demonstrates is that intensive string manipulations should be performed in VIs that are not subroutines. This will allow multiple threads to perform tasks contained inside the VI. Other threads will not become blocked waiting on a single thread to perform large amounts of memory allocations.

Integer operations require significantly less time to complete, and are often good candidates for subroutine priority. The VI shown in Figure 9.23 shows a simple 100-element manipulation. This type of manipulation may not be common in everyday computing, but it serves as a similar example to the one mentioned above. Figure 9.24 shows the profile window for a large number of runs. Notice the significantly lower timing requirements. It is much less likely that a thread will become blocked during execution of this subVI; therefore, it is desirable to give this subVI subroutine priority because subroutine VIs have less overhead and will execute faster than standard VIs.

When working with arrays in LabVIEW, try to use fixed-length or predimensioned arrays as much as possible. When using the Initialize Array function, one block of memory will be taken from the heap memory. Replacing individual elements in the array will not require the array to be reallocated. Once a fixed array size is defined, then a VI manipulating this array can be a candidate for subroutine priority. There will not be significant memory allocations that need to be performed. The overhead on the VI will be reduced, improving the performance of the VI.

![Profile Window](image)

**FIGURE 9.24**
9.10 CHAPTER SUMMARY

This chapter began with core multithreading terminology. Threads, processes, applications, and several operating systems were explained. The basics of multithreading — scheduling, priorities, processes, and thread basics — were discussed and defined.

Major myths involving multithreading were addressed. It should never be assumed that threads would improve application performance. Many dataflow applications force a serial order of execution; this is precisely where multithreading will be of the least benefit. Another common misunderstanding regarding threads is the idea that the application performance is proportional to the number of threads running. This is only true if multiple processors are available. Rotating threads of execution in and out of the CPU will cause more performance problems than solutions.

Estimating the optimum number of threads is challenging, but not entirely impossible. The programmer must identify where the maximum number of executable elements is generated in the code. Using this number as the maximum number of useful threads will prevent performance limitations.

Subroutine priority VIs can lead to performance gains.

BIBLIOGRAPHY


