# Using LabVIEW to Create Multithreaded VIs for Maximum Performance and Reliability

**Norma Dorst**

## Introduction

Modern operating systems, such as Microsoft Windows 2000/NT/9x, Sun Solaris 2, and Linux, provide multithreading technology and the ability to exploit multiprocessor computers. Unfortunately, these technologies and the associated terminology can be confusing to understand and difficult to implement. However, LabVIEW (version 5.0 and later) brings you the advantages of powerful multithreading technology in a simple straightforward way. This application note introduces the concepts of multithreading and explains how you can benefit from multithreading as you build measurement and automation systems. In addition, this application note explains how you can use this technology in LabVIEW without increasing the development time or complexity.

## Definitions – Multitasking, Multithreading, and Multiprocessing

Multitasking refers to the ability of the operating system to quickly switch between tasks, giving the appearance of simultaneous execution of those tasks. For instance, in Windows 3.1, a task is generally an entire application, such as Microsoft Word, Microsoft Excel, or LabVIEW. Each application runs for a small time slice before yielding to the next application. Windows 3.1 uses a technique known as *cooperative multitasking,* where the operating system relies on running applications to yield control of the processor to the operating system at regular intervals. Windows 2000/NT/9x rely on *preemptive multitasking,* where the operating system can take control of the processor at any instant, regardless of the state of the application currently running. Preemptive multitasking guarantees better response to the user and higher data throughput.

Multithreading extends the idea of multitasking into applications, so that specific operations within a single application can be subdivided into individual threads, each of which can theoretically run in parallel. Then, the operating system can divide processing time not only among different applications, but also among each thread within an application. For example, in a LabVIEW multithreaded program, the application might be divided into three threads – a user interface thread, a data acquisition thread, and an instrument control thread – each of which can be assigned a priority and operate independently. Thus, multithreaded applications can have multiple tasks progressing in parallel along with other applications.

Multiprocessing refers to two or more processors in one computer, each of which can simultaneously run a separate thread. Thus, a single-threaded application can run on only one processor at a time. A multithreaded application can have two separate threads executing simultaneously on two processors. In the LabVIEW multithreaded example, the data acquisition thread could run on one processor while the user interface thread runs on another processor. As you will read later in this application note, single-threaded applications can have a serious effect on system performance.

## Multithreaded Operating Systems

Although multithreading technology is not a new concept, only in recent years have PC users been able to take advantage of this powerful technology. Today, LabVIEW users can create multithreaded applications on Windows 2000/NT/9x, Sun Solaris 2, and Linux. With the exception of Windows 9x, these multithreaded operating systems are capable of running on multiprocessor computers for maximum execution performance.

On the LabVIEW nonmultithreaded operating systems, including Windows 3.1, Mac OS (prior to Mac OS X), and HP-UX, LabVIEW programs continue to work in single-threaded mode. In fact, because LabVIEW transparently takes care of all the multithreaded operations for you, a LabVIEW program that runs on Windows 3.1 in single-threaded mode, automatically runs in multithreaded mode on Windows 2000/NT/9x.

## Benefits of Multithreaded Applications

Applications that take advantage of multithreading have numerous benefits, including the following:

- More efficient CPU use
- Better system reliability
- Improved performance on multiprocessor computers

### More Efficient CPU Use

In many LabVIEW applications, you make synchronous acquisition calls to an instrument or data acquisition device. Such calls often take some time to complete. In a single-threaded application, a synchronous call such as this effectively blocks, or prevents, any other task within the LabVIEW application from executing until the acquisition completes. In LabVIEW, multithreading prevents this blocking. While the synchronous acquisition call runs on one thread, other parts of the program that do not depend on the acquisition, such as data analysis and file I/O, run on different threads. Thus, execution of the application progresses during the acquisition instead of stalling until the acquisition completes.

In this way, a multithreaded application maximizes the efficiency of the processor because the processor does not sit idle if any thread of the application is ready to run. Any program that reads and writes from a file, performs I/O, or polls the user interface for activity can benefit from multithreading simply because you can use the CPU more efficiently during these synchronous activities.

### Better System Reliability

By separating the program onto different execution threads, you can prevent other operations in your program from adversely affecting important operations. The most common example is the effect the user interface can have on more time-critical operations. Many times, screen updates or responses to user events can decrease the execution speed of the program. For instance, when someone moves a window, resizes it, or opens another window, the program execution effectively stops while the processor responds to that user interface event.

In a LabVIEW multithreaded application, user interface operations are separated onto a dedicated user interface thread, and the data acquisition, analysis, and file I/O portions of the program can run on different threads. By giving the user interface thread a lower priority than other more time-critical operations, you can ensure that the user interface operations do not prevent the CPU from executing more important operations, such as collecting data from a computer-based instrument. Giving the user interface lower priority improves the overall system reliability, including data acquisition and processing, the performance of LabVIEW, and the performance of the computer as a whole, by ensuring that data is not lost because an operator moves a window, for example.

Another example where multithreading provides better system reliability is when you perform high-speed data acquisition and display the results. Screen updates are often slow relative to other operations, such as continuous high-speed data acquisition. If you attempt to acquire large amounts of data at high speed in a single-threaded application and display all that data in a graph, the data buffer may overflow because the processor is forced to spend too much time on the screen update. When the data buffer overflows, you lose data.

However, in a LabVIEW multithreaded application with the user interface separated on its own thread, the data acquisition task can reside on a different, higher priority thread. In this scenario, the data acquisition and display run independently so the acquisition can run continuously and send data into memory without interruption. The display runs as fast as it can, drawing whatever data it finds in memory at execution time. The acquisition thread preempts the display thread so you do not lose data when the screen updates.

### Improved Performance on Multiprocessor Computers

One of the most promising benefits of multithreading is that it can harness the power of multiprocessor computers. Many high-end computers today offer two or more processors for additional computation power. Multithreaded applications are poised to take maximum advantage of those computers. In a multithreaded application where several threads are ready to run simultaneously, each processor can run a different thread. In a multiprocessor computer, the application can attain true parallel task execution, thus increasing overall system performance.

In contrast, single-threaded applications can run on only a single processor, thus preventing them from taking advantage of the multiple processors to improve performance. Therefore, to achieve maximum performance from multithreaded operating systems and/or multiprocessor machines, an application must be multithreaded.

## Creating Multithreaded Applications

Even though the benefits of multithreading are numerous, many people choose not to exploit this powerful technology because it can be difficult to implement. However, several development tools provide mechanisms to create and debug multithreaded applications.

### Text-Based Programming Approaches

In Visual C/C++ and Borland C++, you can use built-in multithreading libraries to create and manage threads. However, in text-based languages, where code typically runs sequentially, it often can be difficult to visualize how various sections of code run in parallel. Because the syntax of the language is sequential, the code is basically run line by line. And, because threads within a single program usually share data, communication among threads to coordinate data and resources is so critical that you must implement it carefully to avoid incorrect behavior when running in parallel. You must write extra code to manage these threads. Thus, the process of converting a single-threaded application into a multithreaded one can be time consuming and error prone.

Text-based programming languages must incorporate special synchronization functions when sharing resources such as memory. If you do not implement multithreading properly in the application, you may experience unexpected behavior. Conflicts can occur when multiple threads request shared resources simultaneously or share data space in memory. Current tools, such as multithread-aware debuggers, help a great deal, but in most cases you must carefully keep track of the source code to prevent conflicts. In addition, you must often adapt your code to accommodate parallel programming.

### Graphical Programming Approaches

A graphical programming paradigm lends itself to the development of parallel code execution. Because it is much easier to visualize the parallel execution of code in a dataflow environment, where two icons or two block diagrams reside side by side, graphical programming is ideal for developing multithreaded applications. However, the same problems with communication among threads can arise unless LabVIEW manages the threads for you.

## Multitasking in LabVIEW

LabVIEW uses preemptive multithreading on operating systems that offer this feature. LabVIEW also uses cooperative multithreading. Operating systems and processors with preemptive multithreading use a limited number of threads, so in certain cases, these systems return to using cooperative multithreading.

**(Windows 2000/NT/9x and Solaris 2)** The application is multithreaded. The execution system preemptively multitasks VIs using threads. However, a limited number of threads are available. For highly parallel applications, the execution system uses cooperative multitasking when available threads are busy. Also, the operating system handles preemptive multitasking between the application and other tasks.

**(Mac OS and Windows 3.1)** The application is single-threaded. The execution system cooperatively multitasks VIs using its own scheduling system. The application also cooperatively multitasks with other applications by periodically yielding a small amount of time.

**(HP-UX and Solaris 1)** The application is single-threaded. The execution system cooperatively multitasks VIs using its own scheduling system. The operating system handles preemptive multitasking between the application and other tasks.

### Basic Execution System

The following description applies to single-threaded and multithreaded execution systems.

The basic execution system maintains a queue of active tasks. For example, if you have three loops running in parallel, at any given time one task is running and the other two are waiting in the queue. Assuming all tasks have the same priority, one of the tasks runs for a certain amount of time. That task moves to the end of the queue, and the next task runs. When a task completes, the execution system removes it from the queue.

The execution system runs the first element of the queue by calling the generated code of the VI. At some point, the generated code of that VI checks with the execution system to see if the execution system assigns another task to run. If not, the code for the VI continues to run.

#### Synchronous/Blocking Nodes

A few nodes or items on the block diagram are synchronous, meaning they do not multitask with other nodes. In a multithreaded application, they run to completion, and the thread in which they run is monopolized by that task until the task completes.

Code Interface Nodes (CINs), DLL calls, and computation functions run synchronously. Most analysis VIs and data acquisition VIs contain CINs and therefore run synchronously.

Almost all other nodes are asynchronous. For example, structures, I/O functions, timing functions, and subVIs run asynchronously.

The Wait, Wait for Occurrence, Dialog Box, GPIB, and VISA functions and the Serial VIs wait for the task to complete but can do so without holding up the thread. The execution system takes these tasks off the queue until their task is complete. When the task completes, the execution system puts it at the end of the queue. For example, when the user clicks a button on a dialog box the Dialog Box function displays, the execution system puts the task at the end of the queue.

## Managing the User Interface in the Single-Threaded Application

In addition to running VIs, the execution system must coordinate interaction with the user interface. When you click a button, move a window, or change the value of a slide control, the execution system manages that activity and makes sure that the VI continues to run in the background.

The single-threaded execution system multitasks by switching back and forth between responding to user interaction and running VIs. The execution system checks to see if any user interface events require handling. If not, the execution system returns to the VI or accepts the next task off the queue.

When you click buttons or pull-down menus, the action you perform might take a while to complete because LabVIEW runs VIs in the background. LabVIEW switches back and forth between responding to your interaction with the control or menu and running VIs.

## Using Execution Systems in Multithreaded Applications

Multithreaded applications have six multiple execution systems that you can assign by selecting **File»VI Properties** and selecting **Execution** in the VI Properties dialog box. You can select from the following execution systems:

- **User interface** – Handles the user interface. Behaves exactly the same in multithreaded applications as in single-threaded applications. VIs can run in the user interface thread, but the execution system alternates between cooperatively multitasking and responding to user interface events.
- **Standard** – Runs in separate threads from the user interface.
- **Instrument I/O** – Prevents VISA, GPIB, and serial I/O from interfering with other VIs.
- **Data acquisition** – Prevents data acquisition from interfering with other VIs.
- **Other 1 and other 2** – Available if tasks in the application require their own thread.
- **Same as caller** – For subVIs, run in the same execution system as the VI that called the subVI.

These execution systems provide some rough partitions for VIs that must run independently from other VIs. By default, VIs run in the Standard execution system.

The names Instrument I/O and Data Acquisition are suggestions for the type of tasks to place within these execution systems. I/O and data acquisition work in other systems, but you can use these labels to partition the application and understand the organization.

Every execution system except User Interface has its own queue. These execution systems are not responsible for managing the user interface. If a VI in one of these queues needs to update a control, the execution system passes responsibility to the User Interface execution system.

Also, every execution system except User Interface has two threads responsible for running VIs from the queue. Each thread handles a task. For example, if a VI calls a CIN, the second thread continues to run other VIs within that execution system. Because each execution system has a limited number of threads, tasks remain pending if the threads are busy, just as in a single-threaded application.

Although VIs you write run correctly in the Standard execution system, consider using another execution system. For example, if you are developing instrument drivers, you might want to use the Instrument I/O execution system.

Even if you use the Standard execution system, the user interface is still separated into its own thread. Any activities conducted in the user interface, such as drawing on the front panel, responding to mouse clicks, and so on, take place without interfering with the execution time of the block diagram code. Likewise, executing a long computational routine does not prevent the user interface from responding to mouse clicks or keyboard data entry.

Computers with multiple processors benefit even more from multithreading. On a single-processor computer, the operating system preempts the threads and distributes time to each thread on the processor. On a multiprocessor computer, threads can run simultaneously on the multiple processors so more than one activity can occur at the same time.

## Prioritizing Parallel Tasks

You can prioritize parallel tasks by strategically using Wait functions or by changing the priority setting in the **Execution Category** of the **VI Properties** dialog box.

In most cases, you should not change the priority of a VI from the default. Using priorities to control execution order might not produce the results you expect. If used incorrectly, the lower priority tasks might be pushed aside completely. If the higher priority tasks are designed to run for long periods, lower priority tasks do not run unless the higher priority task periodically waits.

### Using Wait Functions

You can use the Wait function to make less important tasks run less frequently. For example, if several loops are in parallel and you want some of them to run more frequently, use the Wait functions for the lower priority tasks. This relinquishes more time to other tasks. In many cases, using Wait functions is sufficient. You probably do not need to change the priorities by selecting **File»VI Properties** and selecting **Execution** in the VI Properties dialog box.

As described in the Synchronous/Blocking Nodes section, when a block diagram waits, the computer removes it from the queue so other tasks can run.

Wait functions are most useful in loops polling the user interface. A wait of 100 to 200 ms is barely noticeable, but it frees up the application to handle other tasks more effectively. In addition, the wait frees the operating system so it has more time to devote to other threads or applications. Consider adding waits to the less time-critical sections of block diagrams to make more time available for lower priority tasks.

### Changing the Priorities

You also can change the priority of a VI by selecting **File»VI Properties** and selecting **Execution** in the VI Properties dialog box. You can select from the following levels of priority, listed in order from lowest to highest:

- Background priority (lowest)
- Normal priority
- Above Normal priority
- High priority
- Time-Critical priority (highest)
- Subroutine priority

The first five priorities are similar in behavior (lowest to highest), but the Subroutine priority has additional characteristics. The following two sections apply to all of these priorities, except the subroutine level.

#### Priorities in the User Interface and Single-Threaded Applications

Within the User Interface execution system, priority levels are handled in the same way for single-threaded and multithreaded applications.

In single-threaded applications and in the User Interface execution system of multithreaded applications, the execution system queue has multiple entry points. The execution system places higher priority VIs on the queue in front of lower priority VIs. If a high-priority task is running and the queue contains only lower priority tasks, the high-priority VI continues to run. For example, if the execution queue contains two VIs of each priority level, the time-critical VIs share execution time exclusively until both finish. Then, the high priority VIs share execution time exclusively until both finish, and so on. However, if the higher priority VIs call a function that waits, the execution system removes higher priority VIs from the queue until the wait or I/O completes, assigning other tasks (possibly with lower priority) to run. When the wait or I/O completes, the execution system reinserts the pending task on the queue in front of lower priority tasks. Refer to the Synchronous/Blocking Nodes section for a list of asynchronous functions that wait.

Also, if a high priority VI calls a lower priority subVI, that subVI is raised to the same priority level as the caller for the duration of that call. Consequently, you do not need to modify the priority levels of the subVIs that a VI calls to raise the priority level of the subVI.

## Priorities in Other Execution Systems and Multithreaded Applications

Each of the execution systems has a separate execution system for each priority level, not including the Subroutine priority level nor the User Interface execution system. Each of these prioritized execution systems has its own queue and two threads devoted to handling block diagrams on that queue.

Rather than having six execution systems, there is one for the User Interface system regardless of the priority and 25 for the other systems – five execution systems multiplied by five for each of the five priority levels.

The operating system assigns operating system priority levels to the threads for each of these execution systems based on the classification. Therefore, in typical execution, higher priority tasks get more time than lower priority tasks. Just as with priorities in the User Interface execution system, lower priority tasks do not run unless the higher priority task periodically waits.

Some operating systems try to avoid this problem by periodically raising the priority level of lower priority tasks. On these operating systems, even if a high priority task wants to run continuously, lower priority tasks periodically get a chance to run. However, this behavior varies from operating system to operating system. On some operating systems, you can adjust this behavior and the priorities of tasks.

The User Interface execution system has only a single thread associated with it. The user interface thread uses the Normal priority of the other execution systems. So if you set a VI to run in the Standard execution system with Above Normal priority, the User Interface execution system might not run, which might result in a slow or nonresponsive user interface. Likewise, if you assign a VI to run at Background priority, it runs with lower priority than the User Interface execution system.

As described earlier, if a VI calls a lower priority subVI, that subVI is raised to the same priority level as the caller for the duration of the call.

## Subroutine Priority Level

The Subroutine priority level permits a VI to run as efficiently as possible. VIs that you set for Subroutine priority do not share execution time with other VIs.

When a VI runs at the Subroutine priority level, it effectively takes control of the thread in which it is running, and it runs in the same thread as its caller. No other VI can run in that thread until the subroutine VI finishes running, even if the other VI is at the Subroutine priority level. In single-threaded applications, no other VI runs. In execution systems, the thread running the subroutine does not handle other VIs, but the second thread of the execution system, along with other execution systems, can continue to run VIs.

In addition to not sharing time with other VIs, subroutine VI execution is streamlined so that front panel controls and indicators are not updated when the subroutine is called. A subroutine VI front panel reveals nothing about its execution.

A subroutine VI can call other subroutine VIs, but it cannot call a VI with any other priority. Use the Subroutine priority level in situations in which you want to minimize the overhead in a subVI that performs simple computations.

Also, because subroutines are not designed to interact with the execution queue, they cannot call any function that causes LabVIEW to take them off of the queue. That is, they cannot call any of the Wait, GPIB, VISA, or Dialog Box functions.

Subroutines have an additional feature that can help in time-critical applications. If you right-click on a subVI and select **Skip Subroutine Call if Busy** from the shortcut menu, the execution system skips the call if the subroutine is currently running in another thread. This can help in time-critical loops where the execution system safely skips the operations the subroutine performs, and where you want to avoid the delay of waiting for the subVI to complete. If you skip the execution of a subVI, all outputs of the subVI become the default value for that data type, not the default value for the indicator on the subVI front panel. For example, numeric outputs are zero, string and array outputs are empty, and Boolean parameters are FALSE. If you want to detect if a subroutine ran, make it return TRUE if it ran successfully and FALSE if it did not.

## Suggestions for Using Execution Systems and Priorities

Following is a summary of some general suggestions about using the execution system options described in this document.

In most applications, it is not necessary to use priority levels or an execution system other than the Standard execution system, which automatically handles the multitasking of the VIs.

By default, all VIs run in the Standard execution system at Normal priority. In a multithreaded application, a separate thread handles user interface activity, so the VIs are insulated from user interface interaction. Even in a single-threaded application, the execution system alternates between user interface interaction and execution of the VIs, giving similar results.

In general, the best way to prioritize execution is to use Wait functions to slow down lower priority loops in the application. This is particularly useful in loops for user interface VIs because delays of 100 to 200 ms are barely noticeable to users.

If you use priorities, use them cautiously. If you design higher priority VIs that operate for a while, consider adding waits to those VIs in less time-critical sections of code so they share time with lower priority tasks.

Be careful when manipulating global variables, local variables or other external resources that other tasks change. Use one of the synchronization techniques described in this application note to protect access to these resources.

Priority use in single-threaded and multithreaded applications gives fairly similar results with the same VIs. However, subtle timing differences might exist. If you distribute VIs to customers who run a different operating system, consider testing the applications under those conditions. Make a multithreaded application behave as a single-threaded application by unchecking **Run with multiple threads** in **Tools»Options»Performance and Disk**.

### Simultaneously Calling SubVIs from Multiple Places

Under normal circumstances, the execution system cannot run multiple calls to the same subVI simultaneously. If you try to call a subVI that is not *reentrant* from more than one place, one call runs and the other call waits for the first to finish before running. You can make a VI reentrant by selecting **File»VI Properties** and selecting **Execution** in the VI Properties dialog box. In a reentrant VI, each instance of the call maintains its own state of information. Then, the execution system runs the same subVI simultaneously from multiple places. Reentrant execution is useful in the following situations:

- When a VI waits for a specified length of time or until a timeout occurs
- When a VI contains data that should not be shared among multiple instances of the same VI

To begin reentrant execution, select **File»VI Properties**, select **Execution** in the VI Properties dialog box, and select **Reentrant Execution**. If you select this option, several other options become unavailable, including the following:

- Run when opened
- Suspend when called
- Auto Handling of Menus at Launch
- Allow Debugging

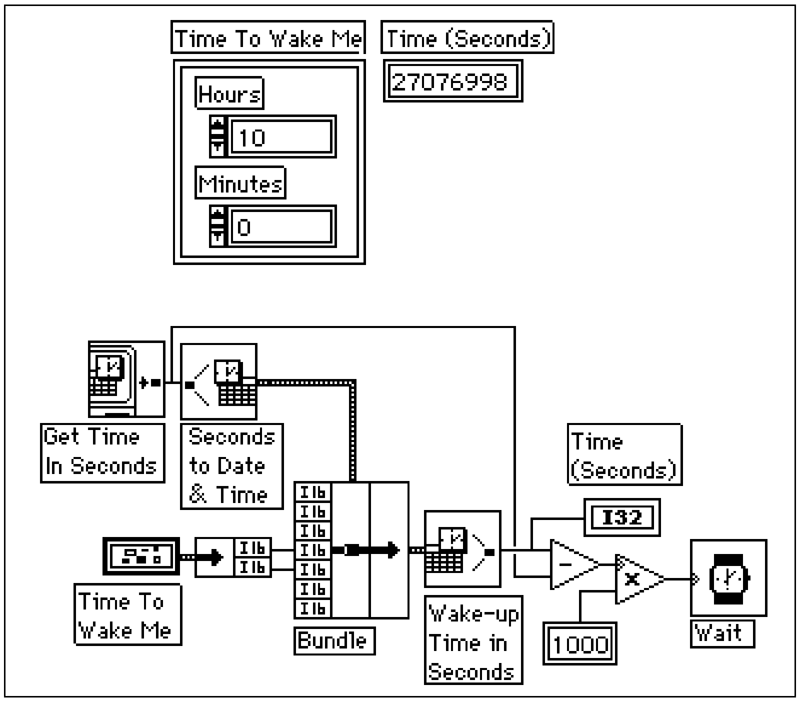
These options are dimmed because the subVI must switch between different copies of the data and different execution states with each call, making it impossible to display the current state continuously.

## Examples of Reentrant Execution

The following two sections describe examples of reentrant VIs that wait and do not share data.

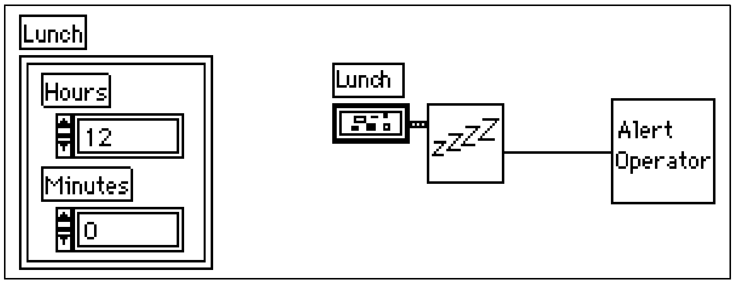
### Using a VI that Waits

The following illustration describes a VI, called Snooze, that takes hours and minutes as input and waits until that time arrives. If you want to use this VI simultaneously in more than one location, the VI must be reentrant.



The Get Time In Seconds function reads the current time in seconds, and the Seconds to Date/Time and converts this value to a cluster of time values (year, month, day, hour, minute, second, and day of week). A Bundle function replaces the current hour and minute with values that represent a later time on the same day from the front panel Time To Wake Me cluster control. The Wake-up Time in Seconds function converts the adjusted record back to seconds, and multiplies the difference between the current time in seconds and the future time by 1,000 to obtain milliseconds. The result passes to a Wait function.

The Lunch VI and the Break VI use Snooze as a subVI. The Lunch VI, whose front panel and block diagram are shown in the following illustration, waits until noon and displays a front panel to remind the operator to go to lunch. The Break VI displays a front panel to remind the operator to go on break at 10:00 a.m. The Break VI is identical to the Lunch VI, except the display messages are different.



For the Lunch VI and the Break VI to run in parallel, the Snooze VI must be reentrant. Otherwise, if you start the Lunch VI first, the Break VI waits until the Snooze VI wakes up at noon, which is two hours late.

### Using a Storage VI Not Meant to Share Its Data

If you make multiple calls to a subVI that stores data, you must use reentrant execution. For example, you create a subVI, ExpAvg, that calculates a running exponential average of four data points.

Another VI uses the ExpAvg subVI to calculate the running average of two data acquisition channels. The VI monitors the voltages at two points in a process and displays the exponential running average on a strip chart. The block diagram of the VI contains two instances of the ExpAvg subVI. The calls alternate – one for Channel 0, and one for Channel 1. Assume Channel 0 runs first. If the ExpAvg subVI is not reentrant, the call for Channel 1 uses the average computed by the call for Channel 0, and the call for Channel 0 uses the average computed by the call for Channel 1. By making the ExpAvg subVI reentrant, each call can run independently without sharing the data.

## Synchronizing Access to Global and Locals Variables and External Resources

Because the execution system can run several tasks in parallel, you must make sure global and local variables and resources are accessed in the proper order.

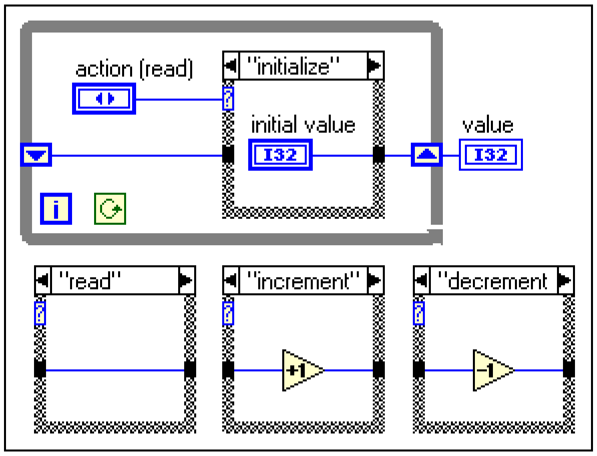
### Preventing Race Conditions

You can prevent race conditions in one of several ways. The simplest way is to have only one place in the entire application through which a global variable is changed.

In a single-threaded application, you can use a Subroutine priority VI to read from or write to a global variable without causing a race condition because a Subroutine priority VI does not share the execution thread with any other VIs. In a multithreaded application, the Subroutine priority level does not guarantee exclusive access to a global variable because another VI running in another thread can access the global variable at the same time.

### Functional Global Variables

Another way to avoid race conditions associated with global variables is to use *functional global* variables are VIs that use loops with uninitialized shift registers to hold global data. A functional global variable usually has an **action** input parameter that specifies which task the VI performs. The VI uses an uninitialized shift register in a While Loop to hold the result of the operation. The following illustration shows a functional global variable that implements a simple count global variable. The actions in this example are **initialize**, **read**, **increment**, and **decrement**.



Every time you call the VI, the block diagram in the loop runs exactly once. Depending on the **action** parameter, the case inside the loop initializes, does not change, incrementally increases, or incrementally decreases the value of the shift register.

Although you can use functional global variables to implement simple global variables, as shown in the previous example, they are especially useful when implementing more complex data structures, such as a stack or a queue buffer. You also can use functional global variables to protect access to global resources, such as files, instruments, and data acquisition devices, that you cannot represent with a global variable.

## Semaphores

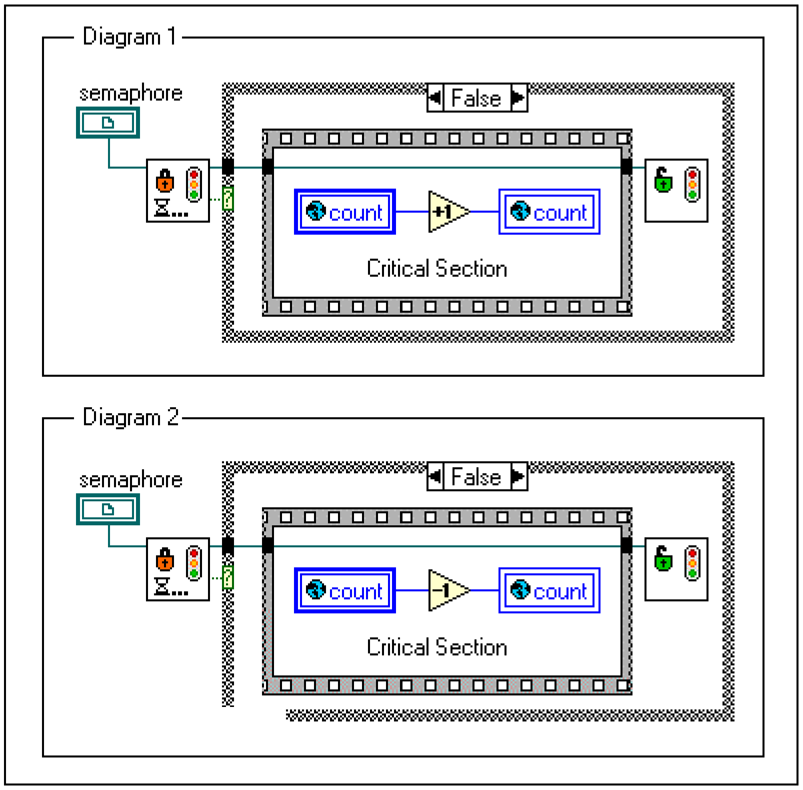
You can solve most synchronization problems with functional global variables, because the functional global VI ensures that only one caller at a time changes the data it contains. One disadvantage of functional global variables is that when you want to change the way you modify the resource they hold, you must change the global VI block diagram and add a new action. In some applications, where the use of global resources changes frequently, these changes might be inconvenient. In such cases, design the application to use a semaphore to protect access to the global resource.

A semaphore, also known as a Mutex, is an object you can use to protect access to shared resources. The code where the shared resources are accessed is called a critical section. In general, you want only one task at a time to have access to a critical section protected by a common semaphore. It is possible for semaphores to permit more than one task (up to a predefined limit) access to a critical section.

A semaphore remains in memory as long as the top-level VI with which it is associated is not idle. If the top-level VI becomes idle, LabVIEW clears the semaphore from memory. To prevent this, name the semaphore. LabVIEW clears a named semaphore from memory only when the top-level VI with which it is associated is closed.

Use the Create Semaphore VI to create a new semaphore. Use the Acquire Semaphore VI to acquire access to a semaphore. Use the Release Semaphore VI to release access to a semaphore. Use the Destroy Semaphore VI to destroy the specified semaphore.

The following illustration shows how you can use a semaphore to protect the critical sections. The semaphore was created by entering **1** in the **size** input of the Create Semaphore VI.



Each block diagram that wants to run a critical section must first call the Acquire Semaphore VI. If the semaphore is busy (its **size** is 0), the VI waits until the semaphore becomes available. When the Acquire Semaphore VI returns false for **timed out**, indicating that it acquired the semaphore, the block diagram starts executing the false case. When the block diagram finishes with its critical section (Sequence frame), the Acquire Semaphore VI releases the semaphore, permitting another waiting block diagram to resume execution.

## LabVIEW Provides Hassle-Free Multithreading

All the complex tasks of thread management are transparently built into the LabVIEW execution system. Text-based programmers must learn new, complex programming practices to create a multithreaded application. However all LabVIEW applications are automatically multithreaded without any code modifications. To make an existing, pre-LabVIEW version 5.0 VI multithreaded, load the VI into LabVIEW version 5.0 or later. Expert users who want to have specific control over threads, such as changing thread priorities, can select **File»VI Properties** and select **Execution**.

## Multithreading Programming Examples

In Example 1, a data acquisition program in C creates and synchronizes the separate threads for acquiring, processing, and displaying the data by using events. Managing the threads and events comprises much of the main program. If the program needs additional threads, you must determine if they can be synchronized in the same way as the current simple program, if synchronization is even necessary, or if you must code another synchronization event structure to incorporate the additional tasks.

### Example 1 – Multithreaded Data Acquisition Program in C

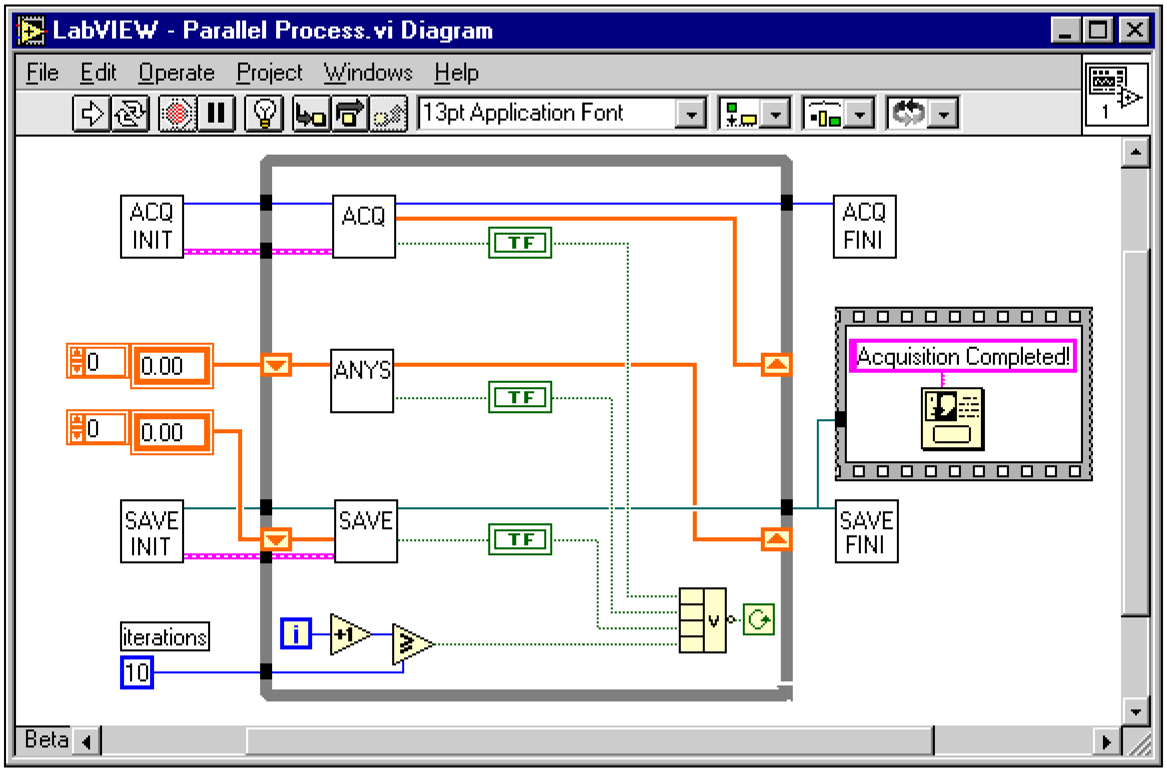
```
#include <windows.h>
#include <winbase.h>
#include <stdio.h>

#include "nidaq.h"              /* for NI-DAQ function prototypes      */
#include "nidaqcns.h"           /* for NI-DAQ constants                */
#include "nidaqerr.h"           /* for NI-DAQ error codes              */

#include "dsp.h"                /* for Analysis prototype              */

// Global buffers to transfer data between threads.
#define kNPts 1024
static i16 gAcquireOut[kNPts];   // acquire
static double gProcessArr[kNPts];// process
static double gSaveArr[kNPts];   // save

// Acquire and Save helper functions.
int InitAcquire(void);
void FinishAcquire(void);
int InitSave(void);
void FinishSave(void);

// Structure passed to each thread.
typedef struct {
       int kind;
       HANDLE doneEvent;
       HANDLE waitEvent;
} SyncRec;

// List of threads.
enum { kAcquireThread, kProcessThread, kDisplayThread, kChildren };
```

```
// Thread sychronization process and the actual work procedure.
DWORD WINAPI ThreadShell(LPVOID arg);
void DoAcquire(void);
void DoProcess(void);
void DoSave(void);

volatile BOOL gExitThreads = FALSE;
volatile int gAcqFailed = 0;      // set to TRUE if the acquisition failed
volatile int gProcessFailed = 0;  // set to TRUE if the process failed
volatile int gSaveFailed = 0;     // set to TRUE if the save failed

main()
{
        int i, j, k;
        char buf[256];
        DWORD id;
        HANDLE shellH, evArr[kChildren];
        SyncRec kidsEv[kChildren];

        printf("Initializing acquire\n");
        // Create synchronization events and threads.
        for (i = 0; i < kChildren; i++) {
                // Set info for this thread.
                kidsEv[i].kind = i;
                evArr[i] = kidsEv[i].doneEvent = CreateEvent(NULL, FALSE, FALSE, NULL);
                kidsEv[i].waitEvent = CreateEvent(NULL, FALSE, FALSE, NULL);

                shellH = CreateThread(NULL, 0, ThreadShell, &kidsEv[i], 0, &id);
                if (! (kidsEv[i].doneEvent && kidsEv[i].waitEvent && shellH)) {
                        printf("Couldn't create events and threads\n");
                        ExitProcess(1);
                }
        }

        if (InitAcquire() && InitSave()) {
                printf("Starting acquire\n");
                for (j = 0; (j < 10) && !gAcqFailed && !gProcessFailed && !gSaveFailed;
                   j++) {
                        // Tell children to stop waiting.
                        for (i = 0; i < kChildren; i++)
                              SetEvent(kidsEv[i].waitEvent);

                        // Wait until all children are done.
                        WaitForMultipleObjects(kChildren, evArr, TRUE, INFINITE);

                        // Main thread coordination goes here...

                        // Copy from process buffer to save buffer.
                        memcpy(gSaveArr, gProcessArr, sizeof(gSaveArr));
                        // Copy from acquire buffer to process buffer.
                        for (k = 0; k < kNPts; k++)
                              gProcessArr[k] = (double) gAcquireOut[k];
                }
                printf("Acquire finished\n");
        }

        // Tell children to stop executing.
        gExitThreads = TRUE;
        // Release children from wait.
        for (i = 0; i < kChildren; i++)
                SetEvent(kidsEv[i].waitEvent);

        // Clean up.
        FinishAcquire();
        FinishSave();

        // Do (minimal) error reporting.
        if (gAcqFailed)
                printf("Acquire of data failed\n");
        if (gProcessFailed)
                printf("Processing of data failed\n");
        if (gSaveFailed)
                printf("Saving data failed\n");

        // Acknowledge finish.
        printf("Cleanup finished.  Hit <ret> to end...\n");
        gets(buf);
        return 0;
}
/*
A shell for each thread to handle all the event
sychronization.  Each thread knows what to do by
the kind field in SyncRec structure.
*/
DWORD WINAPI ThreadShell(LPVOID arg)
{
        SyncRec *ev = (SyncRec *) arg;
        DWORD res;

        while (1) {
                // Wait for main thread to tell us to go.
                res = WaitForSingleObject(ev->waitEvent, INFINITE);
                if (gExitThreads) break;

                // Call work procedure.
                switch (ev->kind) {
                        case kAcquireThread: DoAcquire();break;
                        case kProcessThread: DoProcess();break;
                        case kDisplayThread: DoSave();break;
                        default:
                              printf("Unknown thread kind!\n");
                              ExitProcess(2);
                }

                // Let main thread know we're done.
                SetEvent(ev->doneEvent);
        }
        return 0;
}
// DAQ Section -------------------------------------------------
#define kBufferSize (2*kNPts)
static i16 gAcquireBuffer[kBufferSize] = {0};

static i16 gDevice = 1;
static i16 gChan = 1;

#define kDBModeON  1
```

```
#define kDBModeOFF 0
#define kPtsPerSecond 0
/*
Initialize the acquire.  Return TRUE if we succeeded.
*/
int InitAcquire(void)
{
       i16 iStatus = 0;
       i16 iGain = 1;
       f64 dSampRate = 1000.0;
       i16 iSampTB = 0;
       u16 uSampInt = 0;
       i32 lTimeout = 180;
       int result = 1;
       /*
       This sets a timeout limit (#Sec * 18ticks/Sec) so that if there
       is something wrong, the program won't hang on the DAQ_DB_Transfer
       call.
       */
       iStatus = Timeout_Config(gDevice, lTimeout);
              result = result && (iStatus >= 0);
       /*
       Convert sample rate (S/sec) to appropriate timebase and sample
       interval values.
       */
       iStatus = DAQ_Rate(dSampRate, kPtsPerSecond, &iSampTB, &uSampInt);
              result = result && (iStatus >= 0);
       /* Turn ON software double-buffered mode. */
       iStatus = DAQ_DB_Config(gDevice, kDBModeON);
              result = result && (iStatus >= 0);
       /*
       Acquire data indefinitely into circular buffer from a single channel.
       */
       iStatus = DAQ_Start(gDevice, gChan, iGain, gAcquireBuffer, kBufferSize, iSampTB,
          uSampInt);
              result = result && (iStatus >= 0);
              gAcqFailed = !result;
              return result;
}
void FinishAcquire(void)
{
       /* CLEANUP - Don't check for errors on purpose. */
       (void) DAQ_Clear(gDevice);
       /* Set DB mode back to initial state. */
       (void) DAQ_DB_Config(gDevice, kDBModeOFF);
       /* Disable timeouts. */
       (void) Timeout_Config(gDevice, -1);
}
void DoAcquire(void)
{
       i16 iStatus = 0;
```

```
                i16 hasStopped = 0;
                u32 nPtsOut = 0;

                iStatus = DAQ_DB_Transfer(gDevice, gAcquireOut, &nPtsOut, &hasStopped);
                gAcqFailed = (iStatus < 0);
}

// Analysis Section ---------------------------------------------
void DoProcess(void)
{
        int err;

        /* Perform power spectrum on the data. */
        err = Spectrum(gProcessArr, kNPts);
        gProcessFailed = (err != 0);
}

// Save Section --------------------------------------------------
static HANDLE *gSaveFile;     /* output file pointer */

/*
Initialze save information.  Return TRUE if we succeed.
*/
int InitSave(void)
{
        gSaveFile = CreateFile("data.out", GENERIC_WRITE, 0,
                NULL, CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
        gSaveFailed = gSaveFile == INVALID_HANDLE_VALUE;
        return !gSaveFailed;
}

void FinishSave(void)
{
        CloseHandle(gSaveFile);
}

void DoSave(void)
{
        DWORD nWritten;
        BOOL succeeded;

        succeeded = WriteFile(gSaveFile, gSaveArr, sizeof(gSaveArr), &nWritten, NULL);
        if (!succeeded || nWritten != sizeof(gSaveArr))
        gSaveFailed = 1;
}
```

## Example 2 – G Code Equivalent of C Code in Example 1



The Parallel Process VI, developed using LabVIEW 5.0, contains no additional code for threading because multithreading is built into LabVIEW. All the threads or tasks in the block diagram are synchronized each iteration of the loop. As you add functionality to the LabVIEW VI, LabVIEW handles the thread management automatically.

LabVIEW resolves most of the thread management difficulties. Instead of creating and controlling threads, you simply enable multithreading as a preference, without any additional programming. LabVIEW chooses a multithreaded configuration for the application, or you can customize configurations and priorities by selecting **File»VI Properties** and selecting **Execution** in the VI Properties dialog box. Priority settings automatically translate to set operating system priorities for the multiple threads. You can choose different thread configurations to optimize for data acquisition, instrument control, or other custom configurations. Experimentation in creating multithreading VIs in LabVIEW sometimes yields the best solution. However, if you use C or other text-based languages, rewriting the application to experiment with different configurations can take too much time and effort for the possible rewards.

LabVIEW set in multithreaded execution mode manages threads automatically. With LabVIEW, you do not have to be an expert to write multithreaded applications. However, you can still provide choose custom priorities and configurations if you need more control. Although C users have more low-level direct control of individual threads, they face a more complex set of issues when creating multithreaded applications.

## Conclusion

Used correctly, multithreading offers numerous benefits including more efficient CPU use, better system reliability, and improved performance on multiprocessor computers. As companies continue to standardize on multithreaded operating systems and symmetric multiprocessing computers, the expectations for performance improvements will continue to grow, forcing you to exploit these powerful technologies.

Using LabVIEW, you can start today to maximize performance on multithreaded operating systems and/or multiprocessor computers without increasing either your development time or the complexity of your application. Because the multithreading technology of LabVIEW is implemented transparently, no extra programming is required to take full advantage of multithreading technologies.

341419C-01 Jul00