

Multithreading in LabWindows™/CVI

Updated Nov 4, 2022

Panoramica

[Multicore Programming Fundamentals White Paper Series](#)

While they are often used interchangeably, the terms multitasking, multithreading, and multiprocessing all refer to distinctly different concepts. *Multitasking* refers to the ability of an OS to switch between tasks quickly to give the appearance of simultaneous execution of those tasks. When running in a preemptive multitasking system, applications can be suspended at any time. With *multithreading*, applications can separate their own tasks into individual threads. In a multithreaded program, the OS directs each thread to execute code for a period of time, referred to as a time slice, before switching execution to another thread. The act of stopping execution of one thread and starting execution of another is referred to as a thread switch. The OS typically can perform thread switches quickly enough to give the appearance of concurrent execution of more than one thread at a time. *Multiprocessing* refers to using multiple processors on one computer. In a symmetric multiprocessing (SMP) system, the OS automatically uses all of the processors in the computer to run any threads that are ready to run. With multiprocessing power, your multithreaded application can run multiple threads simultaneously, finishing more tasks in less time.

Single-threaded applications may experience little performance improvement by switching to a multiprocessor machine. They execute on either one processor or another but never on all processors at once as multithreaded applications do. Single-threaded applications can experience an adverse affect on performance through the overhead of the OS switching the application from one processor to another. To achieve maximum performance from multithreaded OSs and/or multiprocessor machines, an application must be multithreaded.

Contents

- [Reasons for Multithreading](#)
- [Choosing Your OS](#)
- [Introduction to Multithreading in LabWindows/CVI](#)
- [Running Code in Secondary Threads in LabWindows/CVI](#)
- [Protecting Data](#)
- [Avoiding Deadlocks](#)
- [Monitoring and Controlling Secondary Threads](#)
- [Process and Thread Priorities](#)
- [Message Processing](#)
- [Using Thread-Local Variables](#)
- [Storing Dynamically Allocated Data in Thread-Local Variables](#)
- [Callbacks Executing in Separate Threads](#)
- [Setting the Preferred Processor for a Thread](#)

- [Additional Multithreading Resources](#)

Reasons for Multithreading

There are four major reasons that you might want to use more than one thread in your program. The most common reason is to separate multiple tasks, one or more of which is time-critical and might be subject to interference by the execution of the other tasks. For example, a program that performs data acquisition and displays a user interface is a good candidate for multithreading. In this type of program, the data acquisition is the time-critical task that might be subject to interference by the user interface task. While using a single-threaded approach in a LabWindows/CVI program, you might decide to pull data from the data acquisition buffer, plot the data to a user interface graph, and then process events to allow the user interface to update. If the user chooses to operate your user interface (for example, by dragging a cursor on a graph), the thread continues to process the user interface events and does not return to the data acquisition task before the data acquisition buffer overflows. Using a multithreaded approach in a LabWindows/CVI program, you might put the data acquisition operations in one thread and display the user interface in another thread. This way, while the user is operating the user interface, the OS performs thread switches to give the data acquisition thread time to perform its task.

The second reason you might want to make your program multithreaded is to perform slow input/output operations simultaneously. For example, a program that uses an instrument to test a circuit board might benefit significantly from multithreading. Using a single-threaded approach in a LabWindows/CVI program, you might send data to the serial port to instruct the circuit board to initialize itself. You wait for the board to complete its operation before initializing the test instrument. You must wait for the test instrument to initialize before taking the measurement. With a multithreaded approach in a LabWindows/CVI program, you might use another thread to initialize the test instrument. This way, you wait for the instrument to initialize while you are waiting for the circuit board to initialize. The slow input/output operations are done simultaneously, thereby reducing the total time you spend waiting.

The third reason you might want to make your program multithreaded is to improve performance on a multiprocessor machine. Each processor on a machine can execute a thread. So while the OS gives the appearance of concurrent execution of multiple threads on a single-processor machine, the OS actually does execute multiple threads concurrently on a multiprocessor machine. A program that would benefit from multithreading on a multiprocessor machine is one that performs more than one task simultaneously. For example, a program that acquires data, streams it to disk, analyzes the data, and displays the analyzed data in a user interface would likely benefit from being multithreaded and running on a multiprocessor machine. Writing the data to disk and analyzing the data for display are tasks that you can perform simultaneously.

The fourth reason you might want to make your program multithreaded is to perform a particular task in more than one context at the same time. For example, you might use multithreading in an application that runs tests on parallel test bays. Using a single-threaded approach, the application has to dynamically allocate space for records to hold the results of the test in each bay. The program has to maintain the association between each record and its test bay manually. Using a multithreaded approach, the application can create a separate thread to handle each test bay. The application can then use thread-local variables to create the results records on a per-thread basis. The association between the test bay and its results record is maintained automatically, thereby simplifying the application code.

Choosing Your OS

The Microsoft Windows 9x OSs do not work with multiprocessor machines. Therefore, you must run Windows Vista/XP/2000/NT 4.0 on multiprocessor machines to enjoy the benefits of multiple processors. However, even on single-processor machines, multithreaded programs perform better when run on Windows Vista/XP/2000/NT 4.0 than when run on Windows 9x. This is true because of more efficient thread switching in Windows Vista/XP/2000/NT 4.0. However, this difference in performance is generally not noticeable in most multithreaded programs.

Windows Vista/XP/2000/NT 4.0 OSs are more stable than Windows 9x OSs for program development, especially when writing and debugging multithreaded applications. Any time you suspend or terminate a thread that is executing OS code, there is a chance that you will leave some portion of the OS in a bad state. It is far more common for such a condition to crash a machine running a Windows 9x OS than it is for such a condition to crash a machine running Windows Vista/XP/2000/NT 4.0. For this reason, NI recommends that you use a machine running Windows Vista/XP/2000/NT 4.0 for developing multithreaded applications.

Introduction to Multithreading in LabWindows/CVI

NI LabWindows/CVI software has natively supported multithreaded application creation since the mid-1990s. Now, with the widespread availability of multicore CPUs, you can use LabWindows/CVI to fully take advantage of multithreaded technology capabilities.

The LabWindows/CVI multithreading functions provide the following multiple performance optimizations over standard Interface to Win32 API threading functions:

- **Thread pools** help you schedule functions for execution in separate threads. Thread pools handle thread caching to minimize the overhead associated with creating and destroying threads.
- **Thread-safe queues** abstract passing data between threads. One thread can read from a queue at the same time that another thread writes to the queue.
- **Thread-safe variables** effectively combine a critical section and an arbitrary data type. You can call a single function to acquire the critical section, set the variable value, and release the critical section.
- **Thread locks** simplify using a critical section or mutex by providing a consistent API and by automatically choosing the appropriate mechanism when necessary. For example, LabWindows/CVI automatically uses a mutex if the lock needs to be shared between processes or if threads need to process messages while waiting for the lock. A critical section is used in other cases because it is more efficient.
- **Thread-local variables** provide per-thread instances of variables. The operating system places a limitation on the number of thread-local variables available to each process. The LabWindows/CVI implementation consolidates the thread-local variables, using only one process thread-local variable for all of the thread-local variables in your program.

You can find all of the multithreading functions in the LabWindows/CVI Library Tree under **Utility Library»Multithreading**.

Running Code in Secondary Threads in LabWindows/CVI

The thread in a single-threaded program is referred to as the main thread. The OS creates the main thread when the user tells the OS to begin executing a particular program. In a multithreaded program, the program itself tells the OS to create threads in addition to the main thread. These threads are referred to as secondary threads. A major difference between the main thread and secondary threads is where each type of thread begins execution. The OS begins execution of the main thread in the mainor WinMainfunction. The developer specifies the function in which each secondary thread begins executing.

In a typical LabWindows/CVI multithreaded program, you use the main thread to create, display, and run the user interface. You use secondary threads to perform other time-critical operations such as data acquisition. LabWindows/CVI provides two high-level mechanisms for running code in secondary threads. These mechanisms are thread pools and

asynchronous timers. A thread pool is appropriate for tasks that you need to perform a discrete number of times or tasks that you need to perform in a loop. An asynchronous timer is appropriate for tasks that you need to perform at regular intervals.

Using a Thread Pool

To run code in a secondary thread using a LabWindows/CVI thread pool, call the Utility Library `CmtScheduleThreadPoolFunction` function. Pass the name of the function that you want to execute in a secondary thread. The thread pool schedules your function for execution in one of its threads. Depending on the configuration and current state of the thread pool, the pool creates a new thread in which to execute your function, uses an existing idle thread to execute your function, or waits until an active thread becomes idle and uses that thread to execute the function that you scheduled.

The function that you pass to `CmtScheduleThreadPoolFunction` is referred to as a thread function. Thread pool thread functions can have any name but must have the following prototype:

```
int CVICALLBACK ThreadFunction (void *functionData);
```

The following code shows how you use `CmtScheduleThreadPoolFunction` to execute, in a secondary thread, a thread function that performs data acquisition.

```

int CVICALLBACK DataAcqThreadFunction (void *functionData);
int main(int argc, char *argv[])
{
    int panelHandle;
    int functionId;

    if (InitCVIRTE (0, argv, 0) == 0)
        return -1; /* out of memory */
    if ((panelHandle = LoadPanel(0, "DAQDisplay.uir", PANEL)) < 0)
        return -1;
    DisplayPanel (panelHandle);

    CmtScheduleThreadPoolFunction (DEFAULT_THREAD_POOL_HANDLE, DataAcqThreadFunction, NULL,
    &functionId);
    RunUserInterface ();
    DiscardPanel (panelHandle);
    CmtWaitForThreadPoolFunctionCompletion (DEFAULT_THREAD_POOL_HANDLE, functionId, 0);
    return 0;
}
int CVICALLBACK DataAcqThreadFunction (void *functionData)
{
    while (!quit) {
        Acquire(. . .);
        Analyze(. . .);
    }
    return 0;
}

```

In the preceding code, the main function call to `CmtScheduleThreadPoolFunction` causes the thread pool to create a new thread to execute the `DataAcqThreadFunction` thread function. The main thread returns from `CmtScheduleThreadPoolFunction` without waiting for the `DataAcqThreadFunction` function to complete. The `DataAcqThreadFunction` function executes in the secondary thread at the same time as the main thread is executing the calls in the main function.

The first parameter to `CmtScheduleThreadPoolFunction` indicates the thread pool with which you want to schedule the function. The LabWindows/CVI Utility Library contains a built-in default thread pool. You pass the constant `DEFAULT_THREAD_POOL_HANDLE` to indicate that you want to use the default thread pool. You cannot customize the behavior of the default thread pool. You can call `CmtNewThreadPool` to create a customizable thread pool. `CmtNewThreadPool` returns a thread pool handle, which you pass in the first parameter to `CmtScheduleThreadPoolFunction`. You must call `CmtDiscardThreadPool` to free the resources of a thread pool that you create with `CmtNewThreadPool`.

The last parameter to `CmtScheduleThreadPoolFunction` returns an identifier that you use to reference the scheduled function in subsequent function calls. The call to `CmtWaitForThreadPoolFunctionCompletion` causes the main thread to wait until the thread pool function has finished executing before completing. If the main thread exits before the secondary threads finish executing, the secondary threads might not get a chance to properly clean up resources that they allocated. Any libraries used by those secondary threads also might be denied an opportunity to clean up properly.

Using an Asynchronous Timer

To run code in a secondary thread using the LabWindows/CVI asynchronous timer, call the Toolslib `NewAsyncTimer` function. Pass the name of the function that you want to execute in a secondary thread and the time interval between executions of the function. The function that you pass to `NewAsyncTimer` is referred to as an asynchronous timer callback. The Asynchronous Timers instrument driver calls the asynchronous timer callback functions at the intervals you specify. Asynchronous timer callbacks can have any name but must have the following prototype:

```
int CVICALLBACK FunctionName (int reserved, int timerId, int event, void *callbackData, int eventData1, int eventData2);
```

Because the LabWindows/CVI Asynchronous Timers instrument driver uses the Windows multimedia timer to implement the asynchronous timer callbacks, the minimum valid interval that you can specify is different for different machines. If you specify an interval that is smaller than the maximum resolution available in your system, unpredictable behavior can occur. Unpredictable behavior typically occurs when you set the timer interval to less than 10 ms. Also, the Asynchronous Timers instrument driver uses one multimedia timer thread to execute all the asynchronous timer callbacks that you register in a single program. Thus, if you want your program to execute multiple functions in parallel, NI recommends that you use the LabWindows/CVI Utility Library thread pool functions instead of the asynchronous timer functions.

Protecting Data

A critical issue that you must address when using secondary threads is data protection. You must protect global variables, static local variables, and dynamically allocated variables from simultaneous access by multiple threads. Failure to do so can cause intermittent logical errors that are difficult to find. LabWindows/CVI provides various high-level mechanisms that help you protect your data from concurrent access. An important consideration when protecting data is avoiding deadlocks.

If a variable is accessed from more than one thread, it must be protected to ensure that its value is not corrupted. For example, consider a multithreaded program that increments an integer global counter variable from multiple threads as follows:

```
count = count + 1;
```

This code is executed as the following sequence of CPU instructions:

1. Move the value in count into a processor register
2. Increment the value in the processor register
3. Write the value in the processor register back into count

Because the OS might interrupt a thread at arbitrary points in its execution, two threads executing this sequence of instructions could execute them in the following order (assume that count starts with the value 5):

```
Thread 1: Move the value in count into a processor register. (count = 5, register = 5)
Switch to Thread 2. (count = 5, register = ?)
Thread 2: Move the value in count into a processor register. (count = 5, register = 5)
Thread 2: Increment the value in the processor register. (count = 5, register = 6)
Thread 2: Write the value in the processor register back into count. (count = 6, register = 6)
Switch to Thread 1. (count = 6, register = 5)
Thread 1: Increment the value in the processor register. (count = 6, register = 6)
```

Thread 1: Write the value in the processor register back into count. (count = 6, register = 6)

Because thread 1 was interrupted before it could increment the value and write it back, the value of count is set to 6 instead of 7. The OS keeps a separate copy of the processor registers for each thread in the system. Even if you write the code as `count++`, you continue to have the same problem because the processor continues to execute the code as multiple instructions. Notice the particular timing condition that causes this failure. This means that your program might run correctly 1000 times for each time it fails. Empirical evidence has shown that multithreaded programs with incorrect data protection typically run correctly during testing but fail immediately when your customer installs and runs them.

Kinds of Data to Protect

Only data that is accessible from more than one thread in a program must be protected. Global variables, static local variables, and dynamically allocated memory are located in common memory areas that are accessible to all threads in a program. Data stored in these types of memory locations must be protected against concurrent access from multiple threads. Function parameters and nonstatic local variables are located on the stack. The OS allocates a separate stack for each thread. Therefore, each thread gets its own copy of parameters and nonstatic local variables, so parameters and nonstatic local variables do not have to be protected against concurrent access. The following code shows which types of data must be protected against concurrent access by multiple threads.

```
int globalArray[1000]; // Must be protected
static staticGlobalArray[500]; // Must be protected
int globalInt; // Must be protected

void foo (int i) // i does NOT need to be protected
{
    int localInt; // Does NOT need to be protected
    int localArray[1000]; // Does NOT need to be protected
    int *dynamicallyAllocdArray; // Must be protected
    static int staticLocalArray[1000]; // Must be protected

    dynamicallyAllocdArray = malloc (1000 * sizeof (int));
}
```

How to Protect Your Data

In general, you protect your data in a multithreaded program by associating an OS thread-locking object with the variable that holds your data. Any time you want to get or set the value of the variable, you first call an OS API function to acquire the OS thread-locking object. After getting or setting the variable, you release the thread-locking object. The OS allows only one thread to acquire a particular thread-locking object at a given time. If a thread calls the OS API function to acquire a thread-locking object while another thread owns it, the thread attempting to acquire the thread-locking object is not allowed to return from the OS API acquire function until the owning thread releases the thread-locking object. A thread that is trying to acquire a thread-locking object that is owned by another thread is referred to as a blocked thread. The LabWindows/CVI Utility Library provides three mechanisms for protecting data – thread locks, thread-safe variables, and thread-safe queues.

A thread lock is a wrapper around a simple OS thread-locking object. There are three circumstances under which you might use a thread lock. If you have a section of code that accesses many shared data variables, you might want to acquire a thread lock before executing the code and release the thread lock after executing the code. The benefit of this approach is that the code is simpler and less error prone than an approach where you protect each piece of data

individually. The drawback is decreased performance because threads in your program will tend to hold the lock longer than is actually necessary, which causes other threads to block (wait) longer than necessary to obtain a lock. Another circumstance under which you might want to use a lock is to protect access to third-party libraries or code that is not thread safe. If, for example, you have a non-thread-safe DLL that controls a piece of hardware and you want to call the DLL from more than one thread, you can create a lock that your threads must acquire before calling into the DLL. The third circumstance under which you might use a thread lock is to protect resources that are shared between programs. Shared memory is an example of such a resource.

A thread-safe variable combines an OS thread-locking object with the data that it protects. This approach is simpler and less error prone than using a thread lock to protect a piece of data. You can use thread-safe variables to protect all types of data, including structure types. Thread-safe variables are less error prone than thread locks because you must call a Utility Library API function to get access to the data. Because the API function acquires the OS thread-locking object, you will not accidentally access the data without acquiring the OS thread-locking object. Thread-safe variables are also simpler to use than thread locks because you need only one variable (the thread-safe variable handle) as opposed to two variables (the thread-lock handle and the protected variable itself), which are needed for thread locks.

A thread-safe queue is a mechanism for safely passing arrays of data between threads. You typically use a thread-safe queue when your program contains one thread that generates an array of data and another thread that must operate on the array of data. An example of such a program is one in which one thread acquires data that another thread analyzes or displays in a LabWindows/CVI user interface. A thread-safe queue has the following advantages over a thread-safe variable of an array type.

- Internally, thread-safe queues use a locking scheme in which one thread can read from the queue at the same time that another thread is writing to the queue (for example, reader and writer threads do not block each other).
- You can configure a thread-safe queue for event-based access. You can register a reader callback that is called when a certain amount of data is available in the queue and/or a writer callback that is called when a specified amount of space is available in the queue.
- You can configure a thread-safe queue to automatically grow if data is added when it is already full.

Thread Lock

Call `CmtNewLock` at program initialization to create a lock for each set of data that you want to protect. This function returns a handle that you use to identify the lock in subsequent function calls. Before accessing the data or code protected by the lock, threads must call `CmtGetLock` to acquire the lock. After accessing the data, the threads must call `CmtReleaseLock` to release the lock. You can call `CmtGetLock` more than once from the same thread (it will not block on subsequent calls), but you must call `CmtReleaseLock` exactly once for each call that you make to `CmtGetLock`. Call `CmtDiscardLock` to free the lock resources before your program exits. The following code demonstrates how to use a LabWindows/CVI Utility Library lock to protect a global variable.


```

int lock;
int count;

int main (int argc, char *argv[])
{
    int functionId;
    CmtNewLock (NULL, 0, &lock);
    CmtScheduleThreadPoolFunction (DEFAULT_THREAD_POOL_HANDLE, ThreadFunction, NULL, &functionId);
    CmtGetLock (lock);
    count++;
    CmtReleaseLock (lock);
    CmtWaitForThreadPoolFunctionCompletion (DEFAULT_THREAD_POOL_HANDLE, functionId, 0);
    CmtDiscardLock (lock);
}

int CVICALLBACK ThreadFunction (void *functionData)
{
    CmtGetLock(lock);
    count++;
    CmtReleaseLock(lock);
    return 0;
}

```

Thread-Safe Variables

Thread-safe variables combine your data and an OS thread-locking object into a single entity. This approach avoids one of the common errors in multithreaded programs: a programmer erroneously forgetting to get a lock before accessing a variable. This approach also makes it easier to pass protected data between functions because you have to pass only the thread-safe variable handle rather than passing a thread-lock handle and the variable that it protects. The LabWindows/CVI Utility Library API contains several functions for creating and accessing thread-safe variables. With these functions, you can create thread-safe variables of any type. Therefore, the parameters to the functions are generic in type and do not provide type safety. Typically, you do not call the LabWindows/CVI Utility Library thread-safe variable functions directly.

The LabWindows/CVI Utility Library header file contains several macros that provide type-safe wrapper functions around the Utility Library API functions. In addition to providing type safety, these macros also help you avoid two other common multithreaded programming errors. These errors are forgetting to release the lock after accessing the data and attempting to release the lock without previously acquiring the lock. Use the `DefineThreadSafeScalarVar` and `DefineThreadSafeArrayVar` macros to create thread-safe variables and the type-safe functions that you use to access them. If you need to access the thread-safe variable from more than one source file, use the `DeclareThreadSafeScalarVar` or `DeclareThreadSafeArrayVar` macro in an include (.h) file to create declarations for the accessor functions. The `DefineThreadSafeScalarVar` (datatype, VarName, maxGetPointerNestingLevel) macro creates the following accessor functions:

```
int InitializeVarName (void);  
void UninitializeVarName (void);  
datatype *GetPointerToVarName (void);  
void ReleasePointerToVarName (void);  
void SetVarName (datatype val);  
datatype GetVarName (void);
```

Note: The macros use the token that you pass as the second parameter to the macro (in this example, VarName) to create customized accessor function names for the thread-safe variable.

Note: The **maxGetPointerNestingLevel** argument is discussed further in the *Detecting Unmatched Calls to GetPointerToVarName* section.

Call `InitializeVarName` once (that is, from only one thread) before accessing the thread-safe variable for the first time. You must call `UninitializeVarName` before your program terminates. If you want to change the value of the variable based on its current value (for example, increment an integer) call `GetPointerToVarName`, change the value, and then call `ReleasePointerToVarName`. You can call `GetPointerToVarName` more than once from the same thread (it does not block on subsequent calls), but you must call `ReleasePointerToVarName` exactly once for each call you make to `GetPointerToVarName`. If you call `ReleasePointerToVarName` without a previous matching call to `GetPointerToVarName` in the same thread, `ReleasePointerToVarName` will report a run-time error.

If you want to set the value of the variable regardless of its current value, call `SetVarName`. If you want to obtain the current value of the variable, call `GetVarName`. It is important to understand that the actual value of the variable could change after `GetVarName` reads the value from memory but before `GetVarName` returns the value to you.

The following code shows how you would use a thread-safe variable as the count variable mentioned in the previous example.

```

DefineThreadSafeScalarVar (int, Count, 0);
int CVICALLBACK ThreadFunction (void *functionData);

int main (int argc, char *argv[])
{
    int functionId;
    int *countPtr;

    InitializeCount();
    CmtScheduleThreadPoolFunction (DEFAULT_THREAD_POOL_HANDLE, ThreadFunction, NULL, &functionId);
    countPtr = GetPointerToCount();
    (*countPtr)++;
    ReleasePointerToCount();
    CmtWaitForThreadPoolFunctionCompletion (DEFAULT_THREAD_POOL_HANDLE, functionId, 0);
    UninitializeCount();
    return 0;
}
int CVICALLBACK ThreadFunction (void *functionData)
{
    int *countPtr;

    countPtr = GetPointerToCount();
    (*countPtr)++;
    ReleasePointerToCount();
    return 0;
}

```

Using Arrays As Thread-Safe Variables

The `DefineThreadSafeArrayVar` macro is similar to `DefineThreadSafeScalarVar` except that it takes an additional parameter that specifies the number of elements in the array. Also, unlike `DefineThreadSafeScalarVar`, `DefineThreadSafeArrayVar` does not define the `GetVarName` and `SetVarName` functions. The following declaration defines a thread-safe array of 10 integers.

```
DefineThreadSafeArrayVar (int, Array, 10, 0);
```

Combining Multiple Variables into a Single Thread-Safe Variable

If you have two or more variables that are somehow related, you must prevent two threads from modifying the values at the same time. An example of this is an array and a count of the number of valid values in the array. If one thread removes values from the array, it must update both the array and the count before allowing another thread to access the data. Although you could use a single LabWindows/CVI Utility Library thread lock to protect access to both of these values, a safer approach is to define a structure and then use that structure as a thread-safe variable. The following example shows how you can use a thread-safe variable in this manner to safely add a value to the array.

```

typedef struct {
    int data[500];
    int count;
} BufType;

DefineThreadSafeVar(BufType, SafeBuf);

void StoreValue(int val)
{
    BufType *safeBufPtr;
    safeBufPtr = GetPointerToSafeBuf();
    safeBufPtr->data[safeBufPtr->count] = val;
    safeBufPtr->count++;
    ReleasePointerToSafeBuf();
}

```

Detecting Unmatched Calls to GetPointerToVarName

You can specify the maximum number of nested calls to `GetPointerToVarName` through the last parameter (**maxGetPointerNestingLevel**) to `DefineThreadSafeScalarVar` and `DefineThreadSafeArrayVar`. You should generally pass 0 for this parameter so that `GetPointerToVarName` reports a run-time error when it detects two consecutive calls to `GetPointerToVarName` from the same thread without an intervening call to `ReleasePointerToVarName`. For example, the following code generates a run-time error the second time it is executed because it is missing a call to `ReleasePointerToCount`.

```

int IncrementCount (void)
{
    int *countPtr;

    countPtr = GetPointerToCount(); /* run-time error on second execution */
    (*countPtr)++;
    /* Missing call to ReleasePointerToCount here */
    return 0;
}

```

Pass an integer greater than zero as the **maxGetPointerNestingLevel** parameter if your code must make nested calls to `GetPointerToVarName`. For example, the following code sets the **maxGetPointerNestingLevel** parameter to 1 because it is valid to make one level of nested calls to `GetPointerToVarName`.

```
DefineThreadSafeScalarVar (int, Count, 1);
int Count (void)
{
    int *countPtr;
    countPtr = GetPointerToCount();
    (*countPtr)++;
    DoSomethingElse(); /* calls GetPointerToCount */
    ReleasePointerToCount ();
    return 0;
}
void DoSomethingElse(void)
{
    int *countPtr;
    countPtr = GetPointerToCount(); /* nested call to GetPointerToCount */
    ... /* do something with countPtr */
    ReleasePointerToCount ();
}
```

If you do not know the maximum nesting level for `GetPointerToVarName`, pass `TSV_ALLOW_UNLIMITED_NESTING` to disable checking for unmatched `GetPointerToVarName` calls.

Thread-Safe Queue

Using the LabWindows/CVI Utility Library thread-safe queue, you can safely pass data between threads. It is most useful when one thread acquires the data and another thread processes that data. The thread-safe queue handles all the data locking internally for you. Generally, a secondary thread in the application acquires the data while the main thread reads the data when it is available and then analyzes and/or displays the data. The following code shows how a thread uses a thread-safe queue to pass data to another thread. The main thread uses a callback to read the data when it is available.

```

int queue;
int panelHandle;

int main (int argc, char *argv[])
{
    if (InitCVIRTE (0, argv, 0) == 0)
        return -1; /* out of memory */
    if ((panelHandle = LoadPanel(0, "DAQDisplay.uir", PANEL)) < 0)
        return -1;
    /* create queue that holds 1000 doubles and grows if needed */
    CmtNewTSQ(1000, sizeof(double), OPT_TSQ_DYNAMIC_SIZE, &queue);
    CmtInstallTSQCallback (queue, EVENT_TSQ_ITEMS_IN_QUEUE, 500, QueueReadCallback, 0,
    CmtGetCurrentThreadID(), NULL);
    CmtScheduleThreadPoolFunction (DEFAULT_THREAD_POOL_HANDLE, DataAcqThreadFunction, NULL, NULL);
    DisplayPanel (panelHandle);
    RunUserInterface();
    ...
    return 0;
}

void CVICALLBACK QueueReadCallback (int queueHandle, unsigned int event, int value, void *callbackData)
{
    double data[500];
    CmtReadTSQData (queue, data, 500, TSQ_INFINITE_TIMEOUT, 0);
}

```

Avoiding Deadlocks

When two threads are waiting for thread-locking objects that are held by each other, execution cannot continue. This condition is referred to as a deadlock. If the user interface thread is deadlocked, it cannot respond to input from the user. The user must exit the application abnormally. The following example illustrates how a deadlock can occur.

```

Thread 1: Calls function to acquire lock A (Thread 1: no locks, Thread 2: no locks)
Thread 1: Returns from acquire lock function (Thread 1: holds A Thread 2: no locks)
Switch to Thread 2: (Thread 1: holds A Thread 2: no locks)
Thread 2: Calls function to acquire lock B (Thread 1: holds A Thread 2: no locks)
Thread 2: Returns from acquire lock function (Thread 1: holds A Thread 2: holds B)
Thread 2: Calls function to acquire lock A (Thread 1: holds A Thread 2: holds B)
Thread 2: Blocked because Thread 1 holds lock A (Thread 1: holds A Thread 2: holds B)
Switch to Thread 1: (Thread 1: holds A Thread 2: holds B)
Thread 1: Calls function to acquire lock B (Thread 1: holds A Thread 2: holds B)
Thread 1: Blocked because Thread 2 holds lock B (Thread 1: holds A Thread 2: holds B)

```

Similar to errors that occur from not protecting data, deadlock errors are often intermittent because of differences in the timing of thread switches in different executions of a program. For example, if the switch to Thread 2 does not occur until after Thread 1 holds both lock A and lock B, Thread 1 is able to complete its work and release the locks for Thread B to acquire later. A deadlock like the one described above can occur only when your threads acquire more than one lock at the same time. You can employ a simple rule to avoid this kind of deadlock. When acquiring more than one thread-locking object, every thread in your program must always acquire the thread-locking objects in the same order. The following LabWindows/CVI Utility Library functions acquire thread-locking objects and return without releasing them.

- CmtGetLock
- CmtGetTSQReadPtr
- CmtGetTSQWritePtr
- CmtGetTSVPtr

Note: Typically, you do not call CmtGetTSVPtr directly. It is called by the GetPtrToVarName function that the DeclareThreadSafeVariable macro creates. Therefore, you must treat every GetPtrToVarName function that you call as a thread-locking object acquire function with respect to deadlock protection.

The following Interface to Win32 API functions can acquire thread-locking objects without releasing them before returning. **Note:** This is not a comprehensive list.

- EnterCriticalSection
- CreateMutex
- CreateSemaphore
- SignalObjectAndWait
- WaitForSingleObject
- MsgWaitForMultipleObjectsEx

Monitoring and Controlling Secondary Threads

When you schedule a function to run in a separate thread, you can monitor the execution status of the scheduled function. To obtain the execution status of a scheduled function, call CmtGetThreadPoolFunctionAttribute to get the value of the ATTR_TP_FUNCTION_EXECUTION_STATUS attribute. You also can register a callback that the thread pool calls immediately before a scheduled function executes and/or immediately after a scheduled function executes. You must use CmtScheduleThreadFunctionAdv to schedule the function if you want to register such a callback.

Typically, secondary threads should finish executing before the main thread exits the program. If the main thread exits before secondary threads finish executing, the secondary threads might not get a chance to clean up resources that they allocated. Any libraries used by those secondary threads also might be denied an opportunity to clean up properly.

You can call CmtWaitForThreadPoolFunctionCompletion to wait safely for your secondary threads to finish execution before allowing your main thread to exit.

In some cases, your secondary thread function must keep performing some task until the main thread signals it to stop. In this case, the secondary thread typically performs its task inside a while loop. The while loop condition is an integer variable that the main thread sets to a nonzero value when it wants to signal the secondary thread to stop executing. The following code shows how to use a while loop to control when a secondary thread finishes executing.

```

volatile int quit = 0;

int main (int argc, char *argv[])
{
    int functionId;
    CmtScheduleThreadPoolFunction (DEFAULT_THREAD_POOL_HANDLE, ThreadFunction, NULL, &functionId);
    // This would typically be done inside a user interface
    // Quit button callback.
    quit = 1;
    CmtWaitForThreadPoolFunctionCompletion (DEFAULT_THREAD_POOL_HANDLE, functionId, 0);
    return 0;
}

int CVICALLBACK ThreadFunction (void *functionData)
{
    while (!quit) {
        ...
    }
    return 0;
}

```

Note: If you use the `volatile` keyword, this code will function properly in an optimizing compiler such as Microsoft Visual C++. An optimizing compiler determines that nothing inside the while loop can change the value of the `quit` variable. Therefore, as an optimization, the compiler might use only the initial value of the `quit` variable in the while loop condition. Use the `volatile` keyword to notify the compiler that another thread might change the value of the `quit` variable. As a result, the compiler uses the updated value of the `quit` variable each time the loop executes.

Sometimes it is convenient to suspend execution of a secondary thread while the main thread is performing another task. If you suspend a thread that is executing OS code, you might leave the OS in an invalid state. Therefore, you should always call the Interface to Win32 API function `SuspendThreadfunction` from the thread you want to suspend. This way, you know that the thread is not suspended while executing critical code. It is safe to call the Interface to Win32 API function `ResumeThreadfunction` from another thread. The following code demonstrates how to do this.


```

volatile int quit = 0;

int main (int argc, char *argv[])
{
    int functionId;
    CmtScheduleThreadPoolFunction (DEFAULT_THREAD_POOL_HANDLE, ThreadFunction, NULL, &functionId);
    // This would typically be done inside a user interface
    // Quit button callback.
    quit = 1;
    CmtWaitForThreadPoolFunctionCompletion (DEFAULT_THREAD_POOL_HANDLE, functionId, 0);
    return 0;
}

int CVICALLBACK ThreadFunction (void *functionData)
{
    while (!quit) {
        ...
    }
    return 0;
}

```

Process and Thread Priorities

Under Windows, you can specify the relative importance, known as *priority*, of the work being done in each process and thread. If you give your process or a thread in your process a higher priority, that process or thread is given preference over other threads that have a lower priority. This means that when there are two or more threads ready to run, the thread with the highest priority runs first.

Windows groups priorities into classes. All threads in a process share the same priority class. Each thread in a process has a priority that is relative to the priority class of the process. Call the Interface to Win32 API function `SetProcessPriorityClass` to set the priority of a process relative to other processes running on your system.

NI recommends that you not set the priority of your process to real-time priority unless you do so for a very short period of time. When your process is set to real-time priority, it blocks out system interrupts while it is running. This results in the mouse, keyboard, hard disk, and other critical system features not functioning and possibly causing your system to lock up.

If you call `CmtScheduleThreadFunctionAdv` to schedule a function to run in a thread pool, you also can specify the priority of the thread that runs the function that you schedule. The thread pool changes the thread priority before it executes the function you schedule. The thread pool restores the thread priority to its default value after your function finishes executing. You can use `CmtScheduleThreadFunctionAdv` to specify thread priority for threads in the default thread pool as well as for custom thread pools.

If you create a custom LabWindows/CVI Utility Library thread pool (by calling `CmtNewThreadPool`), you can set the default priority of the threads in the pool.

Message Processing

Every thread that creates a window must process Windows messages to avoid causing your system to lock up. The User Interface Library `RunUserInterface` function contains a loop that processes LabWindows/CVI user interface events and processes Windows messages. The User Interface Library `GetUserEvent` and `ProcessSystemEvents` functions process Windows messages each time you call them. Each thread in your program must call `GetUserEvent` or `ProcessSystemEvents` regularly to process Windows messages if either of the following circumstances is true:

- The thread creates a window and does not call `RunUserInterface`.
- The thread creates a window and calls `RunUserInterface` but executes callbacks that take a significant amount of time (more than a few hundred milliseconds) before returning back to the `RunUserInterface` loop.

However, there can be places in your code where it is not appropriate to process Windows messages. When you call `GetUserEvent`, `ProcessSystemEvents`, or `RunUserInterface` from a LabWindows/CVI user interface thread, that thread can call a user interface callback. If you call one of these functions in a user interface callback, the thread might call another callback. Unless you have planned for this, this event might generate unexpected behavior.

Utility Library multithreading functions that cause your threads to wait in a loop allow you to specify whether to process messages in the waiting thread. For example, `CmtWaitForThreadPoolFunctionCompletion` has an **Option** parameter with which you can specify that the waiting thread processes Windows messages.

It is not always obvious when a thread creates a window. User Interface Library functions such as `LoadPanel`, `CreatePanel`, and `FileSelectPopup` create windows that you display and discard. These functions also create one hidden window for each thread that calls them. This hidden window is not destroyed when you discard the visible window. In addition to these User Interface Library functions, various other LabWindows/CVI library functions and Windows API functions create hidden, background windows. To avoid causing your system to lock up, you must process Windows messages in threads that create windows in either of these two ways.

Using Thread-Local Variables

Thread-local variables are similar to global variables in that they are accessible from any thread. While global variables hold a single value for all threads, thread-local variables hold separate values for each thread in which they are accessed. You typically use thread-local variables when your program is structured so that it performs a particular task in more than one context at the same time, spawning a separate thread for each context. For example, if you write a parallel tester program that spawns a thread to handle each unit under test, you might use thread-local variables to hold information that is specific to each unit (for example, the serial number).

Although the Windows API provides a mechanism for creating and accessing thread-local variables, this mechanism limits the number of thread-local variables you can have in each process. The LabWindows/CVI Utility Library thread-local variable functions do not have this limitation. The following code demonstrates how to create and access a thread-local variable that holds an integer.

```
volatile int quit = 0;
volatile int suspend = 0;
int main (int argc, char *argv[])
{
    int functionId;
    HANDLE threadHandle;
    CmtScheduleThreadPoolFunction (DEFAULT_THREAD_POOL_HANDLE, ThreadFunction, NULL, &functionId);
    ...
    // This would typically be done in response to user input or a
    // change in program state.
    suspend = 1;
    ...
    CmtGetThreadPoolFunctionAttribute (DEFAULT_THREAD_POOL_HANDLE, functionId,
ATTR_TP_FUNCTION_THREAD_HANDLE, &threadHandle);
    ResumeThread (threadHandle);
    ...
    return 0;
}
int CVICALLBACK ThreadFunction (void *functionData)
{
    while (!quit) {
        if (suspend) {
            SuspendThread (GetCurrentThread ());
            suspend = 0;
        }
        ...
    }
    return 0;
}
```

```

int CVICALLBACK ThreadFunction (void *functionData);
int tlvHandle;
int gSecondaryThreadTlvVal;

int main (int argc, char *argv[])
{
    int functionId;
    int *tlvPtr;

    if (InitCVIRTE (0, argv, 0) == 0)
        return -1; /* out of memory */
    CmtNewThreadLocalVar (sizeof(int), NULL, NULL, NULL, &tlvHandle);
    CmtScheduleThreadPoolFunction (DEFAULT_THREAD_POOL_HANDLE, ThreadFunction, 0, &functionId);
    CmtWaitForThreadPoolFunctionCompletion (DEFAULT_THREAD_POOL_HANDLE, functionId, 0);
    CmtGetThreadLocalVar (tlvHandle, &tlvPtr);
    (*tlvPtr)++;
    // Assert that tlvPtr has been incremented only once in this thread.
    assert (*tlvPtr == gSecondaryThreadTlvVal);
    CmtDiscardThreadLocalVar (tlvHandle);
    return 0;
}

int CVICALLBACK ThreadFunction (void *functionData)
{
    int *tlvPtr;

    CmtGetThreadLocalVar (tlvHandle, &tlvPtr);
    (*tlvPtr)++;
    gSecondaryThreadTlvVal = *tlvPtr;
    return 0;
}

```

Storing Dynamically Allocated Data in Thread-Local Variables

If you use a thread-local variable to store a dynamically allocated resource, you must free each copy of the resource that you allocate. In other words, you must free each copy of the resource for each thread that allocated it. With the LabWindows/CVI thread-local variable functions, you can specify a discard callback for a thread-local variable. When you discard the thread-local variable, the callback is called for each thread that accessed the variable. The following code demonstrates how you create and access a thread-local variable that holds a dynamically allocated string.

```

int CVICALLBACK ThreadFunction (void *functionData);
void CVICALLBACK StringCreate (char *strToCreate);
void CVICALLBACK StringDiscard (void *threadLocalPtr, int event, void *callbackData, unsigned int threadID);
int tlvHandle;
volatile int quit = 0;
volatile int secondStrCreated = 0;

int main (int argc, char *argv[])
{
    int functionId;

    if (InitCVIRTE (0, argv, 0) == 0)
        return -1; /* out of memory */
    CmtNewThreadLocalVar (sizeof(char *), NULL, StringDiscard, NULL, &tlvHandle);
    CmtScheduleThreadPoolFunction (DEFAULT_THREAD_POOL_HANDLE, ThreadFunction, "Secondary Thread",
    &functionId);
    StringCreate ("Main Thread");
    while (!secondStrCreated){
        ProcessSystemEvents ();
        Delay (0.001);
    }
    CmtDiscardThreadLocalVar (tlvHandle);
    quit = 1;
    CmtWaitForThreadPoolFunctionCompletion (DEFAULT_THREAD_POOL_HANDLE, functionId, 0);
    return 0;
}
int CVICALLBACK ThreadFunction (void *functionData)
{
    char **sString;

    // Create thread local string variable
    StringCreate ((char *)functionData);

    // Get thread local string and print it
    CmtGetThreadLocalVar (tlvHandle, &sString);
    printf ("Thread local string: %s\n", *sString);

    secondStrCreated = 1;

    while (!quit)
    {
        ProcessSystemEvents ();
        Delay (0.001);
    }

    return 0;
}
void CVICALLBACK StringCreate (char *strToCreate)
{
    char **tlvStringPtr;
    CmtGetThreadLocalVar (tlvHandle, &tlvStringPtr);
    *tlvStringPtr = malloc (strlen (strToCreate) + 1);

```

```

    strcpy (*tlvStringPtr, strToCreate);
}
void CVICALLBACK StringDiscard (void *threadLocalPtr, int event, void *callbackData, unsigned int threadID)
{
    char *str = *(char **)threadLocalPtr;
    free (str);
}

```

Some resources that you allocate must be freed from the same thread that allocated them. Such resources are said to have thread affinity. For example, a panel must be discarded from the same thread that created it. When you call `CmtDiscardThreadLocalVar`, the Utility Library calls the thread-local variable discard callback in the thread that called `CmtDiscardThreadLocalVar`. The Utility Library calls the discard callback once for each thread that accessed the variable. It passes, in the **threadID** parameter to the discard callback, the thread ID of the thread for which it is calling the discard callback. You can use this thread ID to determine whether you can free your resources with thread affinity directly or if you must call the Toolslib `PostDeferredCallToThreadAndWait` function to free the resources in the correct thread. The following code demonstrates how to change the preceding example to free the strings from the threads that allocated them.

```

void CVICALLBACK StringDiscard (void *threadLocalPtr, int event, void *callbackData, unsigned int threadID)
{
    char *str = *(char **)threadLocalPtr;

    if (threadID == CmtGetCurrentThreadID ())
        free (str);
    else
        PostDeferredCallToThreadAndWait (free, str, threadID, POST_CALL_WAIT_TIMEOUT_INFINITE);
}

```

Callbacks Executing in Separate Threads

With some of the LabWindows/CVI libraries, you can receive callbacks in a system-created thread. Because these libraries automatically create threads that execute your callbacks, you do not have to create threads or schedule a function to run in a separate thread. You still have to protect data that is shared between these threads and other threads in your program. Implementations of these callbacks are typically referred to as asynchronous events.

In the LabWindows/CVI GPIB/GPIB 488.2 Library, you can call `ibnotify` to register a callback that the GPIB/GPIB 488.2 Library calls when events occur. You can specify one callback function per board or device. You can specify the events for which you would like your callback to be called. The GPIB/GPIB 488.2 Library creates a thread that it uses to execute your callback.

In the LabWindows/CVI Virtual Instrument Software Architecture (VISA) Library, you can call `viInstallHandler` to register one or more event handlers (callback functions) for the VISA event types (I/O completion, service request, and so on) you want to receive for a particular **ViSession**. The VISA Library usually uses a separate thread that it creates to execute your callback. VISA might use the same thread for all callbacks in a process or a unique thread for each **ViSession**. You must call `viEnableEvent` for the specified event type(s) to notify the VISA Library that you want it to call the event handlers that you registered.

In the LabWindows/CVI VXI Library, each interrupt or callback type has its own callback registration and enabling functions. For example, to receive an NI-VXI interrupt, you must call both SetVXIIntHandler and EnableVXIInt. The VXI Library uses a separate thread that it creates to execute your callback. The VXI Library uses the same threads for all callbacks in a particular process.

Setting the Preferred Processor for a Thread

You can use the Interface to Win32 API function SetThreadIdealProcessor to specify the processor on which you would like to run a particular thread. The first parameter to this function is a thread handle. The second parameter is the zero-based index of the processor. You can call the LabWindows/CVI Utility Library CmtGetThreadPoolFunctionAttribute function with the ATTR_TP_FUNCTION_THREAD_HANDLE attribute to obtain the handle of a thread pool thread. You can call the LabWindows/CVI Utility Library CmtGetNumberOfProcessors function to programmatically determine the number of processors in the machine that is running your program.

You can use the Interface to Win32 API function SetProcessAffinityMask to specify the processors on which the threads in your program are allowed to run. You can use the Interface to Win32 API function SetThreadAffinityMask to specify processors on which a particular thread in your program is allowed to run. The mask you pass to SetThreadAffinityMask must be a subset of the mask you pass to SetProcessAffinityMask.

These functions have an effect only when your program runs on a multiprocessor machine with Microsoft Windows XP/2000/NT 4.0. The Microsoft Windows 9x OSs do not work with multiprocessor machines.

Additional Multithreading Resources

[Multicore Programming Fundamentals White Paper Series](#)

- [Learn How to Debug Multicore ANSI C Applications with LabWindows/CVI](#)
- [Learn More about LabWindows/CVI](#)

The mark LabWindows is used under a license from Microsoft Corporation.