

Andrew Tucker and Johnson M. Hart
Batons: A Sequential Synchronization Object

Multithreaded programming and thread synchronization are fundamental techniques in modern program design, and Windows, Unix, Linux, and most other operating systems now provide thread management and synchronization. This article presents a useful extension to the basic Win32 synchronization objects. The new compound object, the baton, provides thread serialization as well as mutual exclusion.

Win32 supports thread synchronization with a set of three basic kernel objects: mutexes, events, and semaphores [2, 4]. There are also some useful non-kernel objects and operations, such as CRITICAL_SECTIONS and the interlocked operations, which can provide performance advantages and, in some cases, programming simplicity. The Win32 synchronization objects are proprietary. Unix, Linux, and many other systems, on the other hand, support the POSIX Pthreads standard [1], which uses just two objects: mutexes and condition variables.

While the Win32 thread management and synchronization operations are complete and well-defined, multithreaded program development with synchronized threads has always been a challenging task, and defects such as race conditions, deadlocks, lost signals, and priority inversions are common [1, 2]. Such defects can be difficult to detect, isolate, and repair. To make matters worse, operation on SMP (symmetric multiprocessor) systems frequently exposes latent defects not seen on single-processor systems. Furthermore, in the case of Win32, the behavior of events is not always well-understood, as there are subtle distinctions caused by the different event types and signaling mechanisms. Failure to observe these small distinctions is the cause of numerous bugs, and there is not common agreement among Win32 programmers about the best way to use events.

Objectives

The principal objective of this article is to present and implement a new synchronization object, the baton, and to demonstrate its usefulness in several practical applications. Batons perform two functions. First, a baton acts like a mutex, assuring that only one thread (the “owning thread”) can execute a critical code section at a time. Second, a baton forces the threads to execute critical code sections in a specified sequential order.

The secondary objectives are to address several issues that are unavoidable in the course of describing batons and their implementation under Win32:

- Illustrate the construction and usefulness of compound synchronization objects.
- Demonstrate the importance of the distinct Win32 event types and signaling methods.
- Criticize the existing Win32 event design.
- Advocate a simple event/mutex usage model, related to POSIX condition variables, that simplifies Win32 synchronization and improves reliability.
- Compare our solution to a common alternative approach, showing that the alternative is more complex and provides no clear benefits.
- Discuss the advantages of a new open source Pthreads library [5, 6].

Batons

In her book on software safety [3], Nancy Leveson mentions a synchronization primitive, the “baton,” and defines a baton as “a variable that is passed to a routine and checked to determine whether prerequisite tasks have entered their signature.” That is, batons serialize the task (thread) execution while also providing mutual exclusion, similar to that provided by a mutex.

Batons would be a useful addition to the arsenal of Win32 (or other API) thread synchronization primitives. The remainder of this article discusses the implementation of batons for Win32 in C; we will also discuss types of problems that can be solved using batons.

Baton Motivation

Imagine that you have to write code that processes data from several different input files and then writes the results to a single file in a specific order. Although you could do this in the necessary sequence in a single thread, it might be more efficient (especially on a multi-processor machine or systems where files are on different disk drives) to manipulate each file in a separate thread. How do you then ensure that the multiple independent threads write their results to the output file in the required order?

Simply having each thread write to the output file after completing its processing is not sufficient, since the completion order is not deterministic. Requiring the threads to acquire a mutex before writing would ensure that no two threads attempt to append to the output file at the same time, but would not assure that the various threads would perform their writing in the required order.

It is not unusual to see solutions to sequencing problems that depend on thread priorities, knowledge of input file characteristics, alleged knowledge of Win32 kernel scheduling, and similar ad hoc techniques to create the desired sequence. Such solutions, which might appear to work in limited circumstances, are ultimately doomed to failure.

There are many other situations where threads should be forced to execute in a serial order, where execution in a different order would yield different results. For example, banking transactions may need to be executed in the order the transaction requests were received, and the same would be true of a reservation system.

Serializing with Batons

Batons nicely solve this problem. The “signature” previously mentioned is simply an integer specifying that the thread can only acquire the baton after all threads with a lower signature acquired and released the baton. In the previous example problem, you might assign the file output order sequential numbers, one for the first part of the report output, two for the second part of the report output, and so on. Each number would correspond to the thread charged with producing the corresponding output. The goal is then to have a synchronization object that says “block my execution until all lower-numbered threads have completed.”

A thread can acquire and release a baton that was previously acquired. That is, a baton, like a mutex, is owned, and only one thread at a time can own a given baton. Unlike a mutex, however, a baton forces a thread acquisition sequence, assuring that there are no “sequence races.” A thread can acquire a baton more than once, but a different, and larger, sequence number is required for each acquisition. For example, a thread might acquire a baton with a sequence number of three and then later release that baton. A new attempt to acquire the same baton with a sequence number of three would fail, since that sequence has already executed.

Batons versus Win32 Synchronization Objects

Batons behave differently from the Win32 synchronization objects, as they provide both mutual exclusion with ownership and serialization.

Win32 mutexes and CRITICAL_SECTIONS (the latter is not a kernel object) provide mutual exclusion with ownership (only the owning thread can signal, or release, a mutex), but do not provide serialization. The programmer has no direct control over the order in which contending threads acquire a mutex.

Win32 events are used to signal waiting threads. Events can be used in different ways to release all waiting threads or just one thread, but there is no control over which threads are released or the release order. Events can even be used to provide mutual exclusion, although this is not normally their purpose. Manual and auto-reset events behave differently, and `SetEvent()` and `PulseEvent()`, which signal an event, have different effects; there are four distinct combinations, each behaving differently [2, 4].

Win32 semaphores provide neither ownership nor serialization. They can, however, be used for mutual exclusion and they contain a count, which typically is used to indicate the number of units of some resource (buffers, messages in a queue, etc.). A thread will block on a semaphore if the count is zero; otherwise a waiting thread will decrement the count and continue. Win32 does not provide an atomic multiple wait (waiting for a semaphore), and a semaphore can be emulated with an event and mutex [2, 4].

In summary, the ability to force a specified sequence of operation is the unique new feature that batons provide.

Design and Implementation

The interface we designed for batons is modeled after the Win32 interfaces for mutexes, events, and semaphores. If you have ever used any Win32 objects, you are already familiar with how batons are manipulated, although some differences are noted. You can see the C interface to batons in `batons.h` (Listing 1). Four functions are all you need to put batons to work:

```
HBATON CreateBaton(VOID);
BOOL DestroyBaton(HBATON hBaton);
BOOL AcquireBaton(HBATON hBaton, DWORD dwSequence);
BOOL ReleaseBaton(HBATON hBaton);
```

`CreateBaton()` allocates a new `HBATON`, a pointer to the data structure that represents a baton. For simplicity and to concentrate on synchronization issues, there is no provision for object name, security attributes structure, or initial state, although a complete implementation modeled on Win32 objects would require these parameters.

`DestroyBaton()` performs the opposite operation, freeing the resources associated with an `HBATON`.

`AcquireBaton()` attempts to acquire a baton after the baton has been acquired exactly once for every sequence number between one and `dwSequence-1`. The specification requires `dwSequence > 0`. Apart from possible errors, `AcquireBaton()` blocks the calling thread until the desired sequence number is available.

`ReleaseBaton()` signals that the owning thread wishes to relinquish baton ownership, much as a thread would call `ReleaseMutex()`. Suppose that a thread releases a baton that it acquired by passing a sequence number of three to `AcquireBaton()`. If some other thread has called `AcquireBaton()`

with a sequence number of four, then that thread will be unblocked (its call to `AcquireBaton()` will finally return) when the current thread releases the baton. If no threads are waiting for the next sequence number when the current owner calls `ReleaseBaton()`, then the baton will simply be unowned until some thread calls `AcquireBaton()` with the next sequence number. Only the owning thread can release a baton.

`cbatons.h` (Listing 2) provides a C++ class wrapper for batons. This interface automatically takes care of creating, acquiring, releasing, and destroying batons via constructors and destructors.

AcquireBaton() Failure

When you specify a series of steps to be executed in a specific order, there are two possible ways to go awry: skipped sequence numbers and duplicate sequence numbers.

Skips. The first possibility is that you skip a step (sequence number). For example, suppose that three different threads try to acquire baton sequence numbers one, two, and four, but no thread ever tries to acquire baton sequence number three. In this case, attempts to acquire the baton for steps one and two would eventually succeed, but the thread attempting to acquire sequence number four would block forever and never return from the `AcquireBaton()` call. The reason for this is that there is no way for `AcquireBaton()` to know that step three will never be requested; it can only know that it hasn't been requested yet and must assume that the request may arrive at some time in the future.

This situation is no different from the case where a thread waits for mutex ownership using `WaitForSingleObject()`; there is a built-in assumption, but no guarantee, that every owning thread will eventually release the mutex. Similarly, with a baton, there is a built-in assumption that every sequence number, up to some maximum, will be requested eventually.

It is the programmer's responsibility to assure that batons are used correctly in order to avoid leaving threads in a permanently blocked state.

Duplicates. The second problem is that of a duplicate step in the sequence. For example, suppose that a thread is blocked in `AcquireBaton()`, having requested sequence number three, and then a later thread also calls `AcquireBaton()` with a sequence number of three. Luckily, the baton implementation can easily detect this situation and return `FALSE` from one `AcquireBaton()` call when a duplicate sequence is requested.

Baton Data Structure

The baton source code is in `batons.h` (Listing 1) and `batons.c` (Listing 2). The code is built around the `HBATON` type, which is defined as a pointer to a structure with four members. The first two members are Win32 synchronization objects and the last two are "state" variables. Notice that the state variables all have the `volatile` storage modifier to prevent the compiler assuming that no other thread can alter them asynchronously.

`hMutex` is the handle of a mutex that is used to guard (assure mutually exclusive access to) the `HBATON` structure and state variables.

`hEvent` is the handle of an event that is signaled whenever a thread releases the baton, changing baton "state." For technical reasons discussed later, this must be a manual-reset, rather than auto-reset, event.

`dwSeqLastOwner` is the sequence number of the last thread to

own the baton (initially zero). The implementation assures that every sequence number from one to `dwSeqLastOwner` has been owned by exactly one thread.

`dwOwnerId` is the thread identification of the thread that currently owns the baton. This value is zero if the baton is unowned. This state variable is not entirely necessary, but we have included it for safety to prevent the baton from being released by a thread that does not own it.

CreateBaton() and DestroyBaton()

The code for `CreateBaton()` and `DestroyBaton()` is very straightforward. `CreateBaton()` allocates a new baton handle and creates the mutex and event. It also initializes the three state variables. `DestroyBaton()` just closes the two handles and frees the memory allocated for the baton handle.

ReleaseBaton()

You can't understand `ReleaseBaton()` without understanding `AcquireBaton()`, and vice versa, but `ReleaseBaton()` is a simpler place to start. `ReleaseBaton()` starts by acquiring the mutex that protects access to the baton data structure, ensuring that no other thread will be modifying the structure. Once it acquires the mutex, `ReleaseBaton()` checks whether the calling thread is the current owner of this baton and, if not, simply releases the mutex and returns `FALSE`.

The interesting case is when the calling thread really does own this baton. In that case, `ReleaseBaton()` marks the baton as unowned (sets `dwOwnerId` to zero) and then signals the baton's event before releasing the mutex. This event is described in more detail in the description of `AcquireBaton()`, but it is basically an object used to block all threads that are waiting for a sequence number that hasn't arrived yet.

The baton's event is then signaled, indicating that the baton is not owned. The event has no other purpose than to indicate that `dwOwnerId==0`. This could be considered a "condition variable predicate" [1], but it is a "loose predicate" that will only be useful to a thread blocked by a call to `AcquireBaton()` with the correct sequence number. All waiting threads (not just those waiting on the next sequence number) will be released, however, as there is no way to control which thread or threads are released.

We used a manual-reset event and `PulseEvent()` because we believe this is the simplest and most efficient choice, though `SetEvent()` as an alternative is discussed later.

There is no way to signal a specific waiting thread, so all waiting threads must be released, preferably with a single signaling call. Manual-reset events release all waiting threads when signaled, whereas auto-reset events release only a single thread. Therefore, the choice of manual-reset event is natural.

Why not use `SetEvent()`? If you did, then you would need to call `ResetEvent()` either in this function or after the wait in `AcquireBaton()`, or both. There is no assurance as to when any of these resets would be executed, so the event might (or might not) remain signaled for a thread that calls `AcquireBaton()` after the event signal. Likewise, any ready thread can start running at any time.

Therefore, `PulseEvent()` with a manual-reset event is the optimal choice of the four possible combinations. Some readers might be wondering about "missed signals;" the `AcquireBaton()` implementation addresses this concern, which, incidentally, would not be

resolved by using `SetEvent()`. (Consider the possible sequences of event resetting.)

See [2, p. 262] (watch for the error in Table 9-1, corrected on the author's web page!) and [4, p. 185] for a table explaining the four event type-signaling combinations. A review of many other popular Win32 API and multithreading books shows that nearly all other books make incorrect statements or fail to make these distinctions, which are essential for correct synchronization. One author dismisses `PulseEvent()` altogether.

AcquireBaton()

`AcquireBaton()` is where all the remaining action is. Assume, for the time being, Windows NT 4.0 and greater (which includes Windows 2000). [Assume what? Are there some words missing here? -pmv]

`AcquireBaton()` performs all baton state tests and changes to the baton state with the mutex locked; this is essential.

The `while` loop only exits when the baton is unowned and the last baton owner had a sequence number at least as large as the requested sequence minus one. This means that all preceding sequence numbers have been used. The loop body will never be executed if the baton is unowned and all preceding sequence numbers have been satisfied.

The loop is required to implement the policy of: "always test your predicate and then test it again!" [1]. Several readers have objected to this loop as being an inefficient polling loop. While this may be the case, the loop is unavoidable as there is no way for `ReleaseBaton()` to specify which thread to release from the event wait, so each released thread must determine if it is now time to proceed, and, if not, the thread must wait for the next state change. What is more, this loop pattern is standard when using `Pthreads`[1] for the very same reason that the loop is used here. Ironically, several proposed alternatives shown to us have all used loops, although the loops were sometimes disguised. (Any reader with a working loopless alternative that uses a fixed number of synchronization objects is urged to contact the authors.)

Upon exiting the loop, the current thread can acquire the baton if the previous owner had a sequence exactly one less than what is requested. In this case, the thread acquires the baton successfully, the baton state is changed, and `AcquireBaton()` returns. Otherwise, multiple threads requested the same sequence number, this thread lost the thread race, and therefore, must return unsuccessfully.

The loop body, which is entered if the baton is owned or if the requested sequence number is at least two larger than the last sequence that owned the baton, is now critical. There are three steps, the first two of which are combined atomically in the `SignalObjectAndWait()` function call: a) the mutex is released, allowing other threads to change (release) baton state and to try to acquire the baton, b) wait for the event indicating that the baton is unowned, and c) acquire the mutex to test the baton state ("condition variable predicate") again.

Modification for Windows 9x, ME, and CE

`AcquireBaton()` depends upon `SignalObjectAndWait()` to release the mutex and wait on the event. `SignalObjectAndWait()` is "atomic," so no thread can release a baton between the release and the wait, which would cause a "missed signal."

However, `SignalObjectAndWait()` is not supported by Windows 95, 98, ME, CE, and NT 3.51, so a solution is needed for

these systems. The simplest solution is to replace the `SignalObjectAndWait()` call with two function calls:

```
ReleaseMutex (hBaton->hMutex);  
WaitForSingleObject (hBaton->hEvent, EVENT_TIMEOUT);
```

This alternative code can be specified at build time or could be determined at runtime based upon the OS type.

Now, here is the controversial part. `EVENT_TIMEOUT`, a finite value such as 50 (for 50 ms), is required just in case the acquiring thread is preempted between the release and the wait, and another thread performs a release. In this case, there might never be another release and the acquiring thread would block forever. Preemption would not even be required on an SMP system. This “missed signal,” while perhaps improbable, is not impossible and must be anticipated.

Objections and Alternatives

This alternative to using `SignalObjectAndWait()` is very simple, but many readers of drafts have objected to the fact that we really now have a glorified polling loop, assisted by an event. This is correct, and polling loops are not, by their very nature, attractive. One could even consider them a “kludge,” although even Win32 `CRITICAL_SECTIONS` use an adjustable spin count.

Can the timeout in the polling loop be avoided? It can be, although the loop does not go away. A solution was proposed by Ron Burk, and the open source Pthreads code [6] also avoids the timeout. Here is a restatement of the problem and the outline of a solution. A full implementation is in the code archive.

The problem is to prevent a missed signal from the `ReleaseBaton()` thread to `AcquireBaton()` threads. `SignalObjectAndWait()` cannot be used, as operation on all Windows platforms is required.

The solution requires a simple protocol between the threads to notify the releasing thread that all acquiring threads have been released from the event wait.

1. Add a new state variable, `dwWaitCount`, to the `HBATON` structure.
2. Add a new manual-reset event, `hWaitCountZero`, to the `HBATON` structure.
3. Modify `ReleaseBaton()` to replace `PulseEvent(hBaton->hEvent)` with `SetEvent(hBaton->hEvent)`, followed by a wait on `hWaitCountZero` and `ResetEvent(hBaton->hEvent)`.
4. Modify `AcquireBaton()` so that a thread, before the wait on `hBaton->hEvent`, increments `dwWaitCount` and decrements it after the wait. If the count goes to zero, then call `SetEvent(hBaton->hWaitCountZero)` to inform the releasing thread that all waiting threads are past the event wait. `InterlockedDecrement()` is useful in the decrement as the mutex is not locked.

This solution is in the code archive. It works with the tests described later. The benefit is that it removes the timeout. However, the solution adds a state variable and a kernel object, and, in the solution in the archive file, acquire and release require 22 code lines (instead of 15), and 11 (instead of 7) kernel calls. Therefore, this complex alternative can only be justified if it provides performance benefits. We performed extensive tests and found that both solutions had nearly identical performance. (The test scripts are in the archive.)

The need for the timeout is a Win32 event limitation, not a limitation of our solution. The loop itself is unavoidable with the Win32

event model.

Win32 Events versus Pthreads

Win32 events, with their four usage models and unavoidable risk of missed signals, are the source of many synchronization bugs. The introduction of `SignalObjectAndWait()`, which atomically releases one object before waiting on another, can eliminate missed signals but is not useful if you have to create code for all Windows platforms. There are some additional points.

Hart [2, Chapter 10] advocates always using events with a mutex and a predicate-testing loop similar to that in `batons.c` (Listing 2). This programming model, or idiom, is called the “condition variable model.” Briefly, the model uses a mutex to guard the state variable structure and one or more events to signal specific state changes.

Programmers who have used Pthreads [1] will immediately recognize that the `AcquireBaton()` loop body, with the mutex release, event wait, and mutex acquisition, can be replaced by a single `pthread_cond_wait()` call. Pthreads use CVs (condition variables) rather than events, and every CV wait requires a mutex as well. This call avoids missed signals altogether and is relatively efficient, requiring just a single kernel function call.

Windows would benefit by replacing events with Pthreads-like condition variables [5]. Fortunately, there is now an open source Pthreads library implementation [6]. Hart has started to use this library with some success. Not only does the code benefit by using an industry standard with a well-defined synchronization model, but the code is portable to Linux, most Unix systems, Compaq OpenVMS, and other systems. More experience is required, however, before fully endorsing this library for applications that only run on Win32 platforms.

The fact that many authors do not describe events accurately supports the contention that events are confusing and not well-understood.

The complexity of the open source implementation [6], which seeks to perform very reasonable operations, is, in itself, a critique of events. It should not be so difficult to perform such fundamental operations.

As described above, the condition variable loop with the event wait timeout is nothing more than a glorified polling loop, assisted by event signaling that can release threads before the timeout. Unfortunately, this situation is unavoidable without a far more complex implementation. Furthermore, there were no adverse performance effects.

Testing Batons

An example program that uses batons is shown in `test.c` (Listing 3). It creates a global baton and launches several suspended threads with the next sequential sequence number. If requested by a command-line parameter, a sequence number is duplicated in order to test the implementation’s ability to handle this situation.

Additional test features attempt to force errors by using different priorities and by running threads on different processors (on SMP systems) and in the opposite order from their sequence numbers.

The test then resumes the threads in reverse order of creation, which is also in reverse order from their sequence numbers. These steps are taken to affect the thread scheduling as much as possible and test batons more than would be done by simply executing several normal threads. The test code then simply waits for all the threads to complete before exiting.

The thread function acquires the requested sequence for the baton and prints messages stating which sequence has begun and finished executing. If you compile and run the code, you will see that the threads run in sequential order from zero to nine (if 10 is the number of threads you specify).

If you want to test for duplicate sequence detection, enter the duplicated sequence number as the second command-line parameter (after the number of threads). This will cause the thread loop to launch two threads with the same sequence number. `AcquireBaton()` will grant baton ownership to one thread, and it will return `FALSE` for the other. There is no way to assure which of the two threads with duplicate sequence numbers will actually acquire the baton; we developed batons to create such sequencing!

Tests have been run successfully on a variety of Win32 platforms, including SMP systems. Our performance tests used 1,000 threads.

Baton Extensions

The implementation here is straightforward, but is missing several characteristics of Win32 mutexes and events. Here are some possible extensions:

Initial Sequence Number. We have assumed that the initial sequence number is always one. An additional parameter to `CreateBaton()` could be used to set any non-zero value.

Process Sharing. Complete Win32 emulation would require named, process sharable objects with security attributes. See <http://world.std.com/~jmhart/mltwsem.htm> for an example of such an object, which names the event and mutex and stores the object structure in process-shared memory.

Acquire Wait Timeout. A timeout parameter could be added to `AcquireBaton()`; if the baton were not acquired in the specified time, the call would return with a `FALSE` or an integer value to indicate the timeout failure. The Win32 wait functions provide this feature.

Baton State. Currently, a calling thread must determine an appropriate sequence number. If, however, threads are blocked or are timing out, it would be convenient to know the lowest skipped sequence number. An additional `AcquireBaton()` pointer parameter could be used to return `dwSeqLastOwner+1`. A thread could then fill in any sequence gaps, possibly racing other threads to do so.

Granting a Sequence Number. The idea is to allow a thread to call `AcquireBaton()` with a zero sequence number. The thread would then be given the lowest possible number that has not already been requested (a new state variable is needed), and the Boolean return value would be replaced with an integer sequence number. Sequence numbers would then be granted in order of arrival (that is, the order in which the threads call `AcquireBaton()`).

C++ Baton Class. A C++ baton class, wrapping baton objects, is provided in the code archive.

Other Compound Synchronization Objects

The condition variable model enables the development of any number of useful compound synchronization objects. Here are a few examples.

Threshold Barrier. Threads block until a specified number are waiting, and the threads are all released. See [2, pp. 281ff] and [1, pp. 242ff].

Multiple Wait Semaphore. Win32 semaphores are redundant,

and, as implemented, it is not possible to wait atomically for more than one semaphore unit. <http://world.std.com/~jmhart/mltwssem.htm> gives a solution, as does [4, pp. 185ff].

FIFO Queue. See [2, pp. 284ff] for a solution that uses two events, one to indicate that the queue is not full and the other to indicate that it is not empty.

Typed Message Queue. Hart recently completed a consulting engagement that required porting Unix code to Win32. The application used Unix queue functions (`msgget()`, `msgput()`, `msgrcv()`, `msgctl()`), which manage queues with typed messages, where it is possible to wait for a message with a specified type field value. An extension of the FIFO queue solution worked nicely; the essential changes involved the condition variable predicates.

Conclusion

Thread synchronization issues come in many flavors, each with its own set of needs and requirements. Batons are a good solution for some of these issues, especially the parallel I/O example mentioned previously and enforcing order on interdependent threads during program startup and shutdown sequences. This article has shown a baton implementation, discussed how to extend it, and has also showed a technique for reliable Win32 event usage.

References

- [1] David Butenhof. *Programming with POSIX Threads* (Addison-Wesley, 1997).
- [2] Johnson M. Hart. *Win32 System Programming*, Second Edition, (Addison-Wesley, 2000).
- [3] Nancy Leveson. *Safeware: System Safety and Computers* (Addison-Wesley, 1995).
- [4] S. Makofsky, J. Nottingham, and A. Tucker. *Teach Yourself Windows CE Programming in 24 Hours* (SAMS, 1999).
- [5] D. Schmidt and I. Pyralli. *Strategies for Implementing POSIX Condition Variables in Win32*, www.cs.wustl.edu/~schmidt/win32-cv-2.html.
- [6] *Open Source POSIX Threads for Win32*, sources.redhat.com/pthreads-win32. □

Andrew Tucker (ast@halcyon.com, <http://www.halcyon.com/ast>) is a development lead at Aegis Software, specializing in embedded systems consulting. He has written over a dozen articles for technical journals and is co-author of *Teach Yourself Windows CE Programming in 24 Hours*.

Johnson M. Hart (jmhart@world.std.com, <http://world.std.com/~jmhart>) is an independent consultant specializing in Windows, UNIX, multithreaded application development, and professional training. John is the author of *Win32 System Programming*.



Download the code from www.wdj.com/code/.

"This article presents a useful extension to the basic Win32 synchronization objects. The new compound object, the baton, provides thread serialization as well as mutual exclusion."

Listing 1: batons.h — Declaration of baton API

```

/*
  Header file for batons API.
*/

#ifndef __BATONS_H__
#define __BATONS_H__

struct BATON
{
  /* Guard the baton structure */
  HANDLE hMutex;
  /* Signaled when released */
  HANDLE hEvent;
  /* Last owned sequence. Init: 0 */
  volatile DWORD dwSeqLastOwner;
  /* Owner ID. 0 for unowned. */
  volatile DWORD dwOwnerId;
};
typedef struct BATON *HBATON;

#ifdef __cplusplus
extern "C" {
#endif

HBATON CreateBaton(VOID);
BOOL DestroyBaton(HBATON hBaton);
BOOL AcquireBaton(HBATON hBaton, DWORD dwSequence);
BOOL ReleaseBaton(HBATON hBaton);

#ifdef __cplusplus
}
#endif

#endif
/* End of File */

```

Listing 2: batons.c — Implementation of batons

```

/*
  Implementation of batons API
*/
#include <windows.h>
#include "batons.h"
#define EVENT_TIMEOUT 50 /* Tuneable timeout in the wait loop */

/* The baton name is not implemented, but the API provides for
 * naming in the future, along with an OpenBaton() function
 */

HBATON CreateBaton(VOID)
{
  HBATON hBaton = (HBATON)malloc(sizeof(struct BATON));

  if (hBaton != NULL)
  {
    hBaton->hMutex = CreateMutex (NULL, TRUE, NULL);
    hBaton->hEvent = CreateEvent(NULL, TRUE, FALSE, NULL);
    hBaton->dwSeqLastOwner = 0;
    hBaton->dwOwnerId = 0;
    ReleaseMutex (hBaton->hMutex);
  }
  return hBaton;
}

BOOL DestroyBaton(HBATON hBaton)
{
  HANDLE hMutex = hBaton->hMutex;
  BOOL Result = FALSE;

  // Do not destroy an owned baton
  WaitForSingleObject (hMutex, INFINITE);
  if (hBaton->dwOwnerId == 0) {
    CloseHandle(hBaton->hEvent);
    free (hBaton);
    ReleaseMutex (hMutex);
    CloseHandle (hMutex);
    Result = TRUE;
  } else
    ReleaseMutex (hMutex);
  return Result;
}

BOOL AcquireBaton(HBATON hBaton, DWORD dwSequence)
{
  BOOL Result = TRUE;

  if (dwSequence == 0) return FALSE; /* Sequence starts at 1 */
  WaitForSingleObject (hBaton->hMutex, INFINITE);
  while (hBaton->dwOwnerId != 0 ||
        hBaton->dwSeqLastOwner < dwSequence-1) {
    /* The baton is owned,
     * or the baton has not yet
     * been acquired with the preceding sequence */
    ReleaseMutex (hBaton->hMutex);
    WaitForSingleObject (hBaton->hEvent, EVENT_TIMEOUT);
    /* NT 4.0 and greater? Then you can use:
     * SignalObjectAndWait (hBaton->hMutex,
     *                      hBaton->hEvent,
     *                      INFINITE,
     *                      FALSE);
     */
    WaitForSingleObject (hBaton->hMutex, INFINITE);
  }
  /* hBaton->dwOwnerId == 0 &&
   * hBaton->dwSeqLastOwner >= dwSequence-1
   *
   * That is: The baton is unowned and the baton has been
   * acquired with all sequence numbers in the range
   * [1 .. dwSequence-1]
   */
  Result = (hBaton->dwSeqLastOwner == dwSequence-1);
  if (Result) {
    /* The sequence is one more than that of the last owner */
    hBaton->dwSeqLastOwner = dwSequence;
    hBaton->dwOwnerId = GetCurrentThreadId();
  } /* Otherwise, this is a duplicate sequence number */
  ReleaseMutex (hBaton->hMutex);

  return Result;
}

```

Listing 2: batons.c — continued

```

}

BOOL ReleaseBaton(HBATON hBaton)
{
    BOOLEAN Result = FALSE;
    WaitForSingleObject (hBaton->hMutex, INFINITE);
    /* PulseEvent on a manual reset event
     * guarantees all waiting threads are
     * released and then the event is reset
     */
    if (hBaton->dwOwnerId == GetCurrentThreadId()) {
        /* Only the owning thread can release the baton */
        hBaton->dwOwnerId = 0;
        PulseEvent(hBaton->hEvent); /* Baton state change */
        Result = TRUE;
    }
    ReleaseMutex (hBaton->hMutex);
    return Result;
}
/* End of File */

```

Listing 3: test.c — Program to test batons

```

/* Baton test program */

#include <windows.h>
#include <process.h>
#include <stdio.h>
#include <time.h>
#include "batons.h"

HBATON g_hBaton;

DWORD WINAPI ThreadFunc(PVOID pv)
{
    DWORD dw = (DWORD)pv;

    // remove the Acquire/Release calls to see normal
    // thread behavior
    if ( AcquireBaton(g_hBaton, dw) )
    {
        printf("inside ThreadFunc(%2d). Baton acquired.\n", dw);
        Sleep(50 /* * dw */);
        printf("leaving ThreadFunc(%2d). Release baton.\n", dw);
        ReleaseBaton(g_hBaton);
    }
    else {
        printf ("AcquireBaton error: ThreadFunc(%2d).\n", dw);
        return 1; /* Error */
    }
    return 0;
}

```

Listing 3: test.c — continued

```

int main(int argc, char * argv[])
{
#define NTHREADS 20
    /* Usage: baton [nThread [DupTh]]
     * nThread is the number of threads (default: NTHREADS).
     * They will have distinct signatures except for thread
     * number DupTh, which will have signature DupTh-1
     */
    HANDLE *hThreads;
    DWORD i, dwTID, ThExitCode;
    DWORD nThreads = NTHREADS, DupTh = 0xFFFFFFFF;
    SYSTEM_INFO si;

    srand( (unsigned)time( NULL ) ); /* used for random priority */

    GetSystemInfo(&si); /* get sys info for the processor count */

    if (argc > 1) nThreads = atoi(argv[1]);
    if (argc > 2) DupTh = atoi (argv[2]);

    g_hBaton = CreateBaton();

    // create the threads suspended
    hThreads = malloc (((DupTh == 0xFFFFFFFF) ?
        nThreads : nThreads+1)* sizeof(HANDLE));
    if (hThreads == NULL) {
        printf ("Error allocating thread handle array\n");
        return 1;
    }

    for ( i = 0; i < nThreads; i++ )
    {
        hThreads[i] = (HANDLE)_beginthreadex(NULL, 0, ThreadFunc,
            (PVOID)((i==DupTh) ? DupTh-1 : i+1),
            CREATE_SUSPENDED, &dwTID);
    }

    // If a dup thread was requested create one
    // with the missing index
    if ( DupTh != 0xFFFFFFFF )
    {
        hThreads[nThreads] = (HANDLE)_beginthreadex(NULL, 0,
            ThreadFunc, (PVOID)(DupTh+1),
            CREATE_SUSPENDED, &dwTID);

        // bump up the thread count so the following loops
        // include the new thread
        nThreads++;
    }

    // give each one a random priority
    for ( i = 0; i < nThreads; i++ )
    {
        SetThreadPriority(hThreads[i],
            rand()%THREAD_PRIORITY_TIME_CRITICAL);

        //put consecutive threads on alternating processors
        SetThreadAffinityMask(hThreads[i],
            (i%si.dwNumberOfProcessors)+1);
    }

    // resume all threads in backwards order
    for ( i = 0; i < nThreads; i++ ) {
        ResumeThread(hThreads[nThreads - i - 1]);
        Sleep (rand()/1000);
    }

    // Wait for threads to terminate
    for ( i = 0; i < nThreads; i++ ) {
        WaitForSingleObject(hThreads[i], INFINITE);
        GetExitCodeThread (hThreads[i], &ThExitCode);
        printf ("ThreadFunc(%2d) exit code: %d\n", i, ThExitCode);
        CloseHandle(hThreads[i]);
    }
    free (hThreads);

    DestroyBaton(g_hBaton);
    fflush(stdin);
    return 0;
}
/* End of File */

```