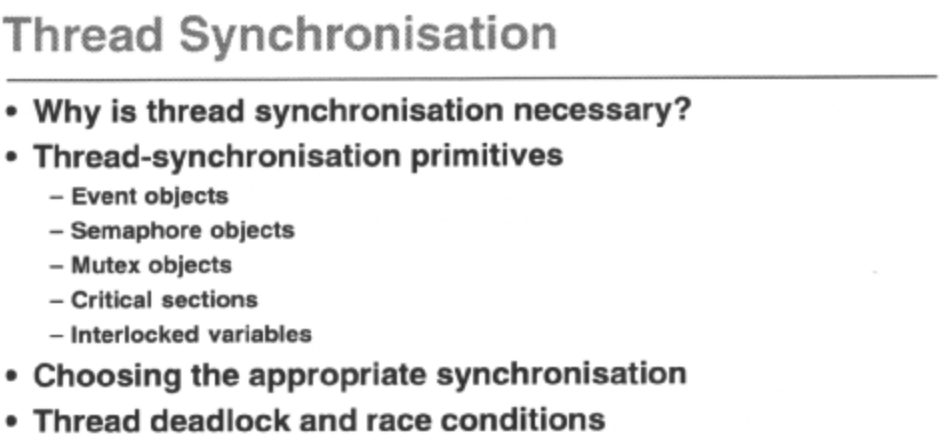


In a multitasking environment it is essential to coordinate the execution of multiple threads in one or more processes. In a multitasking environment where thread execution is conceptually concurrent (actually concurrent in Windows NT SMP systems) programmers must be careful to coordinate the execution of multiple threads in one or more processes.

The scheduler may pre-empt a thread at any time, including while it is in the middle of accessing a data area, device, or section of non-reentrant code. Such *'serially reusable resources'* (SRRs) should only be used by one thread at a time, otherwise their data may become corrupted or a deadlock may occur. Also, it is sometimes important to coordinate threads; one thread may need to wait for another to complete an action before carrying out its task. The work of ensuring that only one thread at a time accesses an SRR is known as *'arbitration'* or *'mutual exclusion',* while the coordination of threads is referred to as *synchronization*

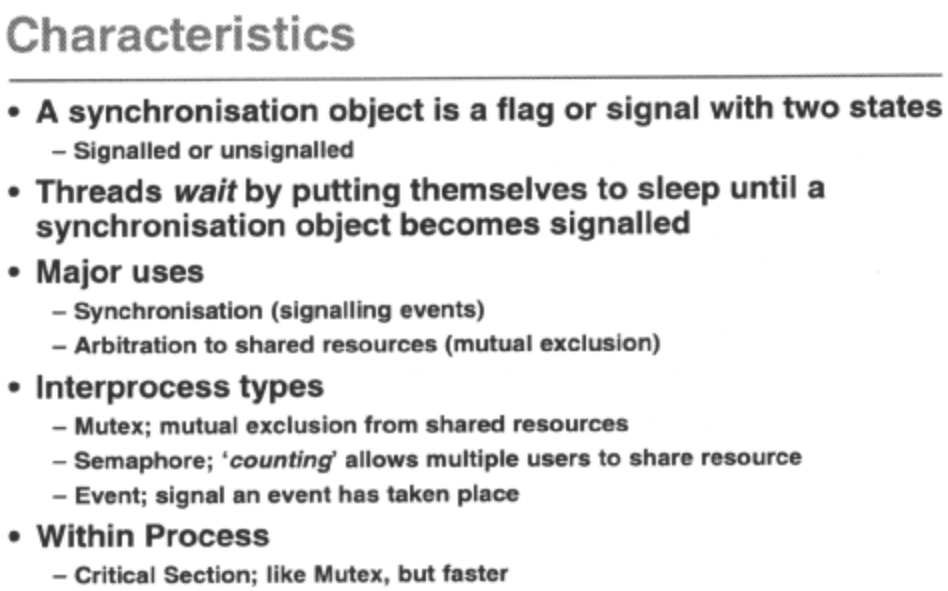
The Win32 synchronization objects are essentially flags maintained by the operating system, which enable threads to signal each other in order to synchronize their activities and to protect non-reentrant code and resources by providing mutual exclusion.

This chapter looks at the four different types of synchronization object provided by Win32, and demonstrates how each kind is used.

Objectives

When you have completed this chapter, you should be able to:

- Explain why thread synchronization is necessary.
- Make an appropriate choice for the method of synchronizing threads.
- Use Event objects to signal that a significant event has taken place.
- Use Semaphore objects to act as a resource gate.
- Use Mutex objects and Critical Section objects to provide mutual exclusion.
- Synchronize threads using one or more Win32 objects.
- Avoid thread deadlock and race conditions.



The Win32 synchronization objects maintain a state of either *'signaled'* or *'unsigned'* and are used to coordinate the threads of one or more processes. This is a cooperative affair and the programmer must code the synchronization logic.

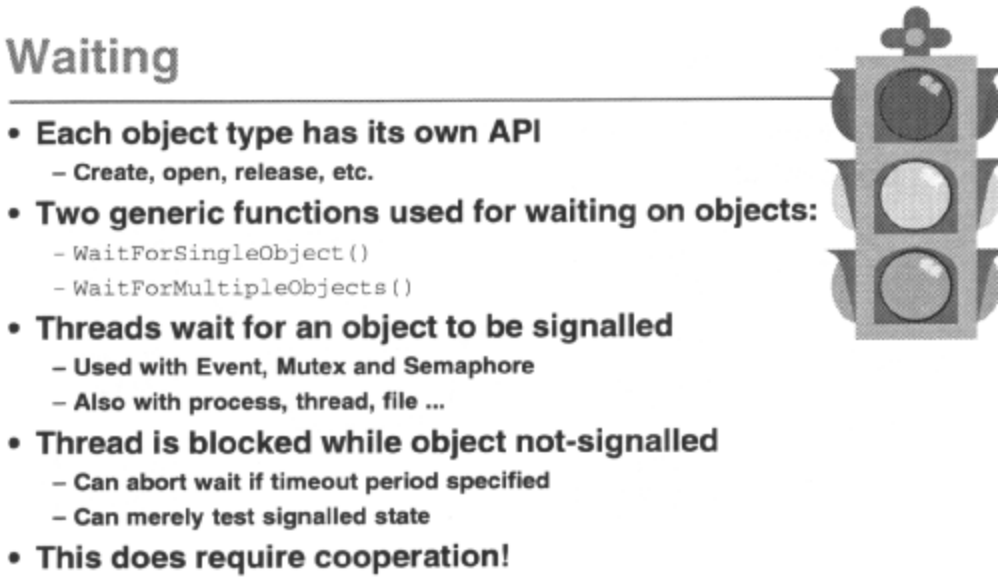
A thread tests to see whether a synchronization object is signaled just before using an SRR or performing an operation that must be synchronized with other cooperating threads. If the synchronization object is signaled, the thread carries on. If it is not signaled, the thread must wait. A waiting thread is immediately suspended, stays so while the state of the synchronization object is unsigned, and does not use any time slices. One or more of the cooperating threads waiting on the synchronization object is woken up to resume execution when it becomes signaled. The synchronization object typically becomes signaled due to another of the cooperative threads signaling it upon a certain condition being met. Perhaps it has finished accessing an SRR or it has formatted some data ready for another thread to use.

There are three types of synchronization objects that can be used by the threads of one or more processes, for intra-process or inter-process synchronization:

- Event objects, which signal that a significant logical event has taken place.
- Semaphore objects, which act as a resource gate, allowing limited multiple users of SRRs.
- Mutex objects, which provide mutual exclusion to SRRs.

These objects are identified by a handle. Any thread can control a synchronization object if it has a handle to it with appropriate access rights. By default, synchronization objects are created with *'synchronize'* and *'modify'* access, so that the object can be waited on and signaled / unsignalled. These objects can be named for use across processes, or unnamed. One of a group of cooperating threads creates a synchronization object, thus obtaining a handle, and all other threads open a handle to it. Each of these three objects has a different API to create it and modify its state, but they all share the same *'waiting'* API.

Another type of Win32 synchronization primitive is a Critical Section, which is similar to a Mutex but faster, as it can only be used by the threads of one process. This object has a different API for create/destroy, modify and wait from the three previously mentioned.



All synchronization in Win32 applications is via cooperative waiting. To wait on a Win32 object, an open handle to the object with *'synchronize'* access is required. `WaitForSingleObject()` will then wait for any appropriate object to become signaled.
The Event, Mutex and Semaphore objects we are about to discuss are specifically designed for the purpose of arbitrating access or serializing operation. However, other types of Win32 object can be waited on. For instance, a Win32 thread object is unsignalled for the whole of its lifetime and becomes signaled when it terminates. So you could wait for a thread to die by waiting for it to become signaled. A Win32 file object is signaled when an outstanding I/O request on it is completed. So you could perform asynchronous I/O on a file by issuing an I/O request and waiting on its handle for it to become signaled. There are good reasons, discussed later, why you would not want to do this.

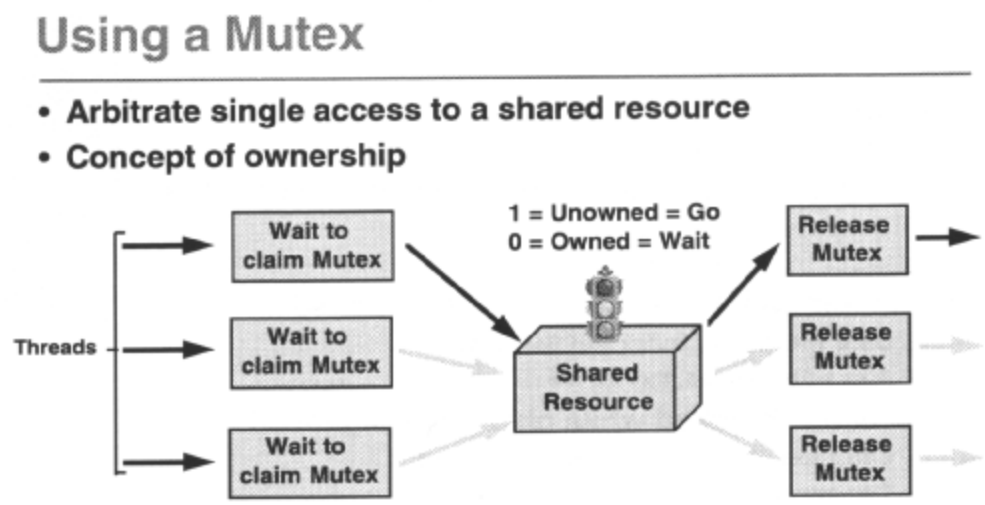
A timeout period can be specified, in ins, as the maximum time `WaitForSingleObject()` will wait for the object before returning.

If an object is not signaled, the function blocks until another thread signals it, or the timeout elapses. A timeout value of INFINITE will wait indefinitely.
A timeout value of 0 causes `WaitForSingleObject()` to return immediately, even if the object is not signaled. This is useful if you want a thread to do something while it is waiting for clearance. It will cause the calling thread to lose the rest of its time slice.

If the state of the object is signaled, or becomes signaled before the time period elapses, the function returns WAIT_OBJECT_0 and the thread can resume execution.

If the timeout elapses before the object is signaled, the function returns WAIT_TIMEOUT.
`WaitForMultipleObjects()` waits for a multiple number of objects, potentially of different types. For instance, you might wait for a thread to terminate and for a Mutex to be unowned. An array of object handles is

specified, and another parameter dictates whether all objects need to be signaled before the function returns, or whether the signaling of any one object will satisfy. Again, a timeout period can be specified with mostly the same operation as above. If it is waiting for only one of many objects to become signaled, the function returns (WAIT OBJECT_0 + *index)*, where *index* is the array item that satisfied the wait. If several objects in the array are signaled, the object with the lowest array index is returned.

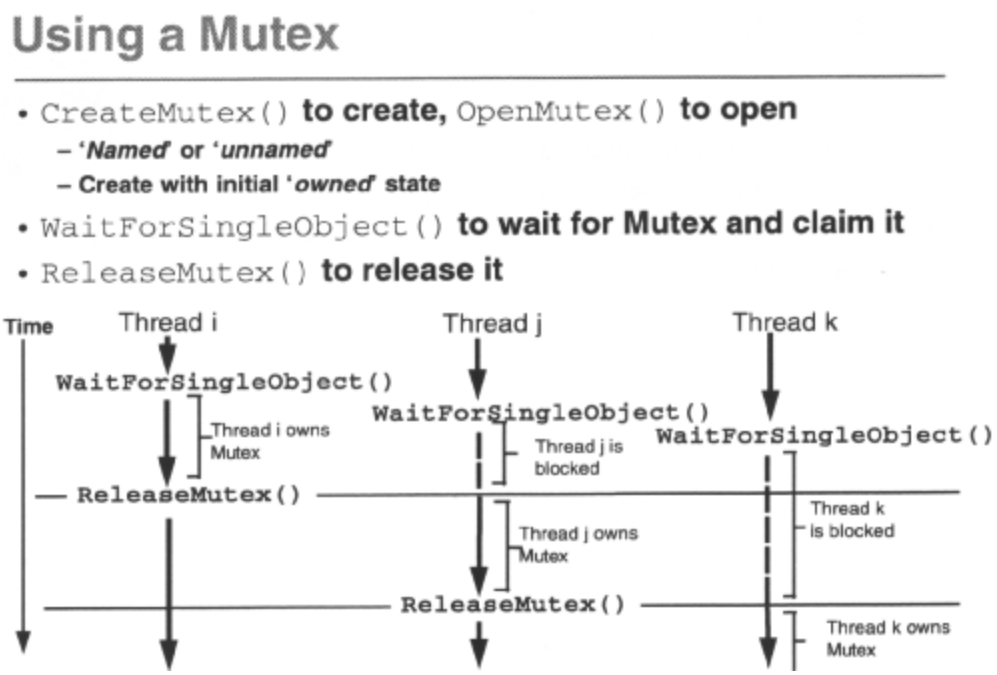


Multiple threads needing to use a serially reusable resource (SRR), such as a data area, device or section of non-reentrant code, should use a Mutex to ensure that only one thread at a time has access to the resource. The threads may be in the same process, in which case they are likely to use an unnamed Mutex, or in different processes, when they should use a named Mutex. If a process other than that which created the Mutex wishes to use it, it must first open a handle to it, specifying its name.

Each thread should request ownership of the Mutex before attempting to access the resource. Only the thread that owns the Mutex can proceed to enter the resource, and it should release the Mutex as soon as it has finished using it. The system queues subsequent requests to claim the Mutex, and transfers ownership to one of the waiting threads as soon as the Mutex is released. Threads, which have requested the Mutex, are blocked until it is their turn to own it.

A typical situation in which one would need to synchronize the operation of different threads in order to guard a SRR, would be when multiple threads have read and write access to shared memory. Each thread would need to request ownership of the Mutex (wait on it) before it could perform read/write operations on the memory. When access to the memory was finished, the Mutex would be released.

It is always important to have the smallest possible *critical section* of code (that which is guarded by a Mutex) to avoid unnecessary waiting.



A Mutex is created by `CreateMutex()` and opened with `OpenMutex()` `CreateMutex()` will open the Mutex if it already exists. They both take the name of the Mutex as a parameter and return a handle, which is used in all Mutex API calls. The name may be `NULL`, in which case the Mutex is unnamed, and will usually be private to the process in which it was created. `CreateMutex()` also specifies whether the Mutex is initially owned or unowned and, optionally, a security descriptor can be specified, which will dictate how the Mutex may be accessed. Default access is 'modify' and 'synchronize'. `OpenMutex()` specifies which kind of access is required to the semaphore. Asking for all access gives modify and synchronize access.

Threads wishing to use a resource protected by a Mutex may request ownership by calling `WaitForSingleObject()`, specifying the handle. If the Mutex is unowned, one of the threads requesting ownership will be given it, and may then access the resource. When finished with the resource, it should release the Mutex by calling `ReleaseMutex()` A thread requesting ownership of an owned Mutex will be blocked until the Mutex is unowned once more and it gets a turn to have possession, or until the specified timeout period expires.

While a thread has ownership of a Mutex, it can make additional wait calls on the same Mutex without blocking. However, it must release the Mutex for each satisfied wait before the Mutex is unowned. This is useful when separate service functions, which could be called from any thread at any time, need to protect the same resource with the same Mutex. If one such function is called from another in a thread that currently owns the Mutex, the additional wait is satisfied. If, however, the same function is called from another thread, the wait is not satisfied.

Here is an example of creating an unnamed Mutex with default security, which is initially unowned:

```
HANDLE hMutex = CreateMutex(  NULL,       /* no security */
                              FALSE,      /* unowned */
                              NULL );     /* no name */
```

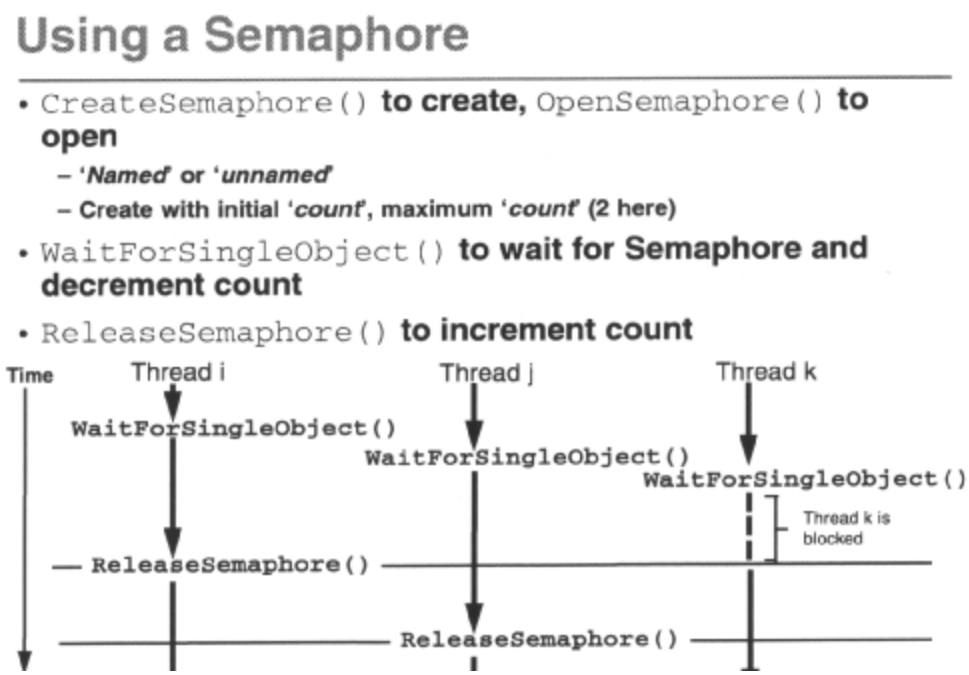
and then to wait on it indefinitely:

```
WaitForSingleObject( hMutex, INFINITE);
```

It is possible that a Mutex may be abandoned, due to the owning thread terminating without releasing the Mutex, and thus locking up the resource. Only Mutex objects can be abandoned, since Event and Semaphore objects cannot be 'owned'.

The system will guard against this. If a thread terminates whilst owning a Mutex, the next wait on that Mutex is satisfied and the waiting thread will be given ownership.
`WaitForSingleObject()` returns with a value of WAIT_ABANDONED to let the successful thread know the circumstances. `WaitForMultipleObjects()` returns with avalue of WAIT_ABANDONED if waiting on all objects to be signaled, or *a value* of (WAIT ABANDONED_0 + *nindex)* if waiting on any object to become signaled, where *nlndex* is the array index of the abandoned Mutex.
Here is an example of waiting for the first of an array of Mutex objects to become signaled within half a second:

```
HANDLE hMutex[3];
DWORD dwResult = WaitForMultipleObjects() 3, hMutex, FALSE, 500 );

switch ( dwResult
      case WAIT_OBJECT_0 + 0:   /* object ahMutex[0] was signaled */
      case WAIT_OBJECT_0 + 2:   /* object ahMutex[2] was signaled */
      case WAIT_ABANDONED_0 + 0: /* object abMutex[0] is an abandoned Mutex */
      case WAIT_ABANDONED_0 + 2: /* object hMutex[2] is an abandoned Mutex */

case WAIT_TIME_OUT:             /* timeout expired before any object became
                                signaled */
```



A Semaphore is created by calling `CreateSemaphore()` and opened with `OpenSemaphore()`. They both take the name of the Semaphore and return a handle to the Semaphore, which is used in all Semaphore API calls. If the name is *NULL*, the Semaphore will be unnamed, and will usually be private to the process that created it. `CreateSemaphore()` specifies a maximum and initial count for the Semaphore and, optionally, a security descriptor can be specified, which will dictate how the Semaphore may be accessed. Default access is modify and synchronize. `OpenSemaphore()` specifies which kind of access is required to the Semaphore. Asking for all access gives modify and synchronize access.

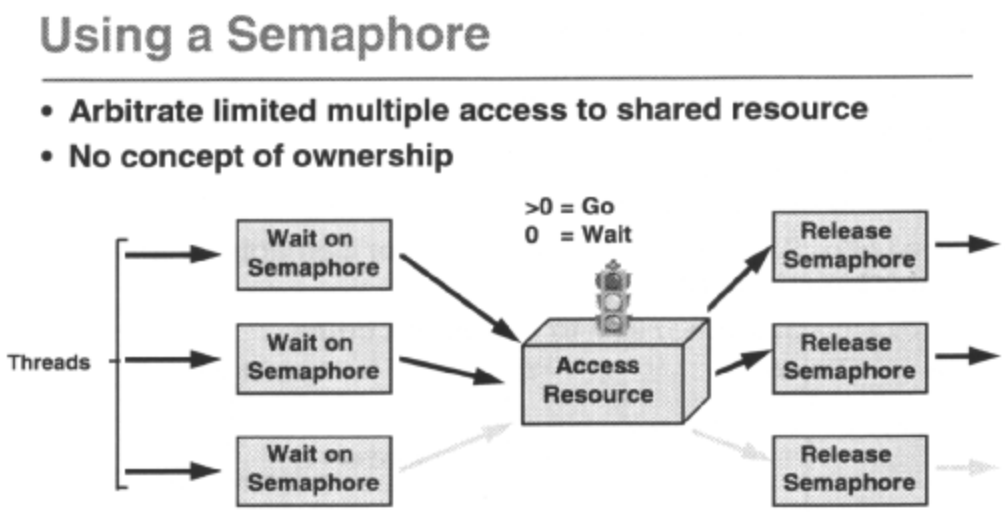
Threads wait on a Semaphore by calling `WaitForSingleObject()`, specifying a Semaphore handle. All waiting threads are suspended until the Semaphore count is greater than zero or their individual wait timeout expires, whichever is sooner. If a thread waiting on the Semaphore is successfully released, the count is decremented. When finished with the resource, it should release the Semaphore by calling `ReleaseSemaphore()`, which will increment the count by the specified amount. This can be useful if the thread that created the Semaphore wants to block access to the resource until it is initialized. It creates the Semaphore with an initial count of zero, then releases the Semaphore later to increment the count to the maximum. A Semaphore has no concept of ownership, so if a thread successfully claims a Semaphore twice, the count goes down by two.

Here is an example of creating an unnamed Semaphore with default security, which has a maximum count of 5 and an initial count of 0:

```
HANDLE hSem = CreateSemaphore( NULL,      /* no security */
      0,    /* initial */
      5,    /* max
      NULL ); /* no name */
```

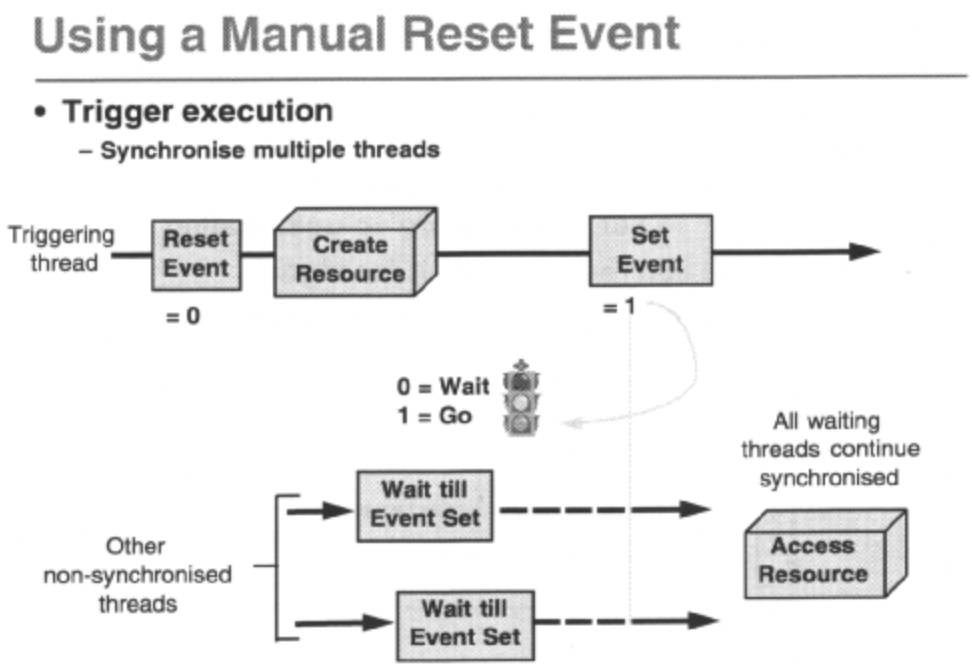
and then later, after initialization of the resource it protects: `ReleaseSemaphore( hSem, 5, NULL );`

and then, later still, waiting on it indefinitely: `WaitForSingleObject ( hSem, INFINITE);`



A Semaphore maintains a count, and is signaled *if* the count is greater than zero. An initial and maximum count is specified when the Semaphore is created. The count is decremented each time a wait on the Semaphore is satisfied, and incremented when the Semaphore is released.

Typical use would be as a resource 'gate' for protecting a shared resource that can support a limited number of multiple accesses. For example, a DLL arbitrating access to communication ports may only be able to support the same number of simultaneous clients, as there are ports. In this case, the DLL client attach routine would use a counting semaphore to limit the number of simultaneous clients.



An Event provides a signaling mechanism to notify cooperating threads that a significant program event has occurred.

There are two types of Event, *'Manual Reset'* and *'Auto Reset'*.

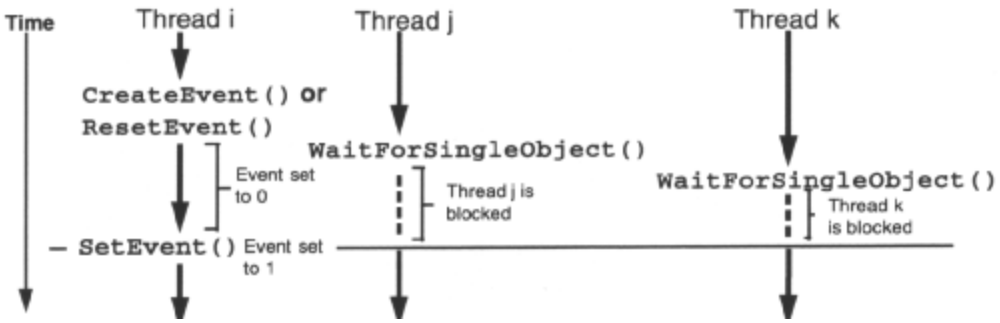
When a Manual Reset (MR) Event is set to signaled, all waiting threads are released. The MR Event must be explicitly reset to unsignaled, unless the Event was *'pulsed'* (set and reset atomically).

A typical situation in which one would need to synchronize the operation of different threads would be when one thread is responsible for creating a file or buffer that other threads are going to write to or read data from. The read and write threads must not attempt to access the data area until it has been created.

The thread responsible for creating the file or buffer must first create a *Manual Reset Event* in the reset state, which it then sets when the file initialization is completed. Other threads wishing to use the file wait on the Event signal, and are suspended until it is set. When this happens they unblock, and can safely start their read and write operations. Of course, at that time they will probably then have to use a Mutex!

Using a Manual Reset Event

- `CreateEvent()` **to create,** `OpenEvent()` **to open**
  - **'*Named*' or '*unnamed*'**
  - **Create with *'manual'* flag `TRUE`, initial *'signal state'***
- `ResetEvent()` **to set *'signal state'* to unsignalled**
- `WaitForSingleObject()` **to wait for signal**
- `SetEvent()` **to set *'signal state'* to signalled**



A Manual Reset Event (MR Event) is created by `CreateEvent()` (setting the *'manual reset'* parameter to TRUE) and opened with OpenEvent() They both take the name of the Event and return a handle to the Event, which is used in all Event API calls. If the name is NULL, the Event will be unnamed, and will usually be private to the process that created it.

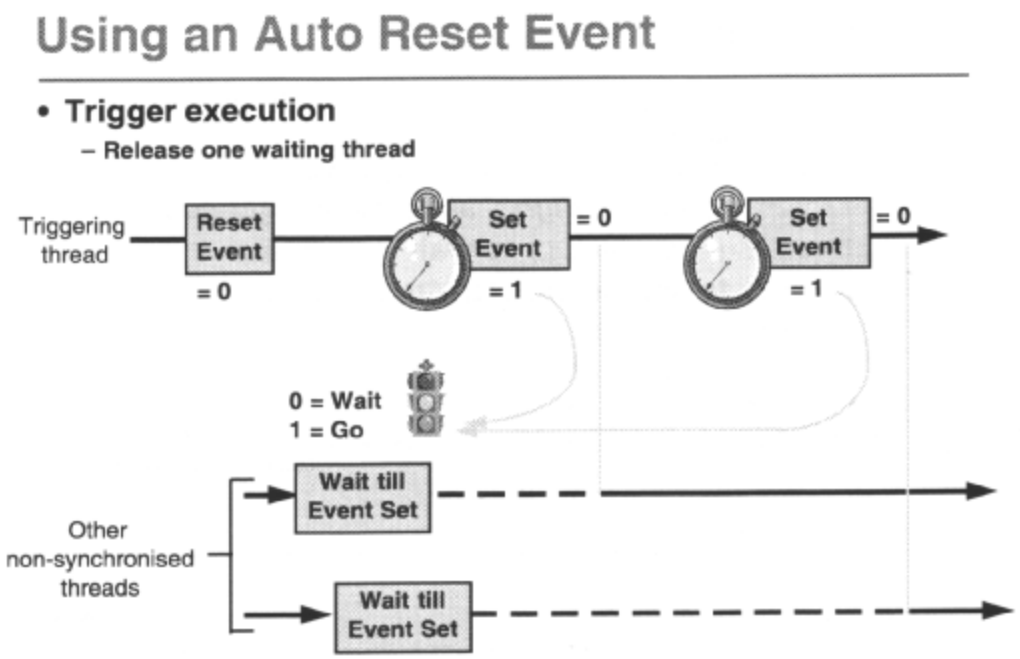
`CreateEvent()` accepts a *'state'* parameter which specifies whether the Event is initialized to signaled (TRUE) or unsignaled (FALSE). It is normally important to set it to the reset state to avoid premature triggering. Optionally, a security descriptor can be specified, which will dictate how the Event may be accessed. Default access is modify and synchronize.

`OpenEvent()` specifies which kind of access is required to the Event. Asking for all access gives modify and synchronize access.

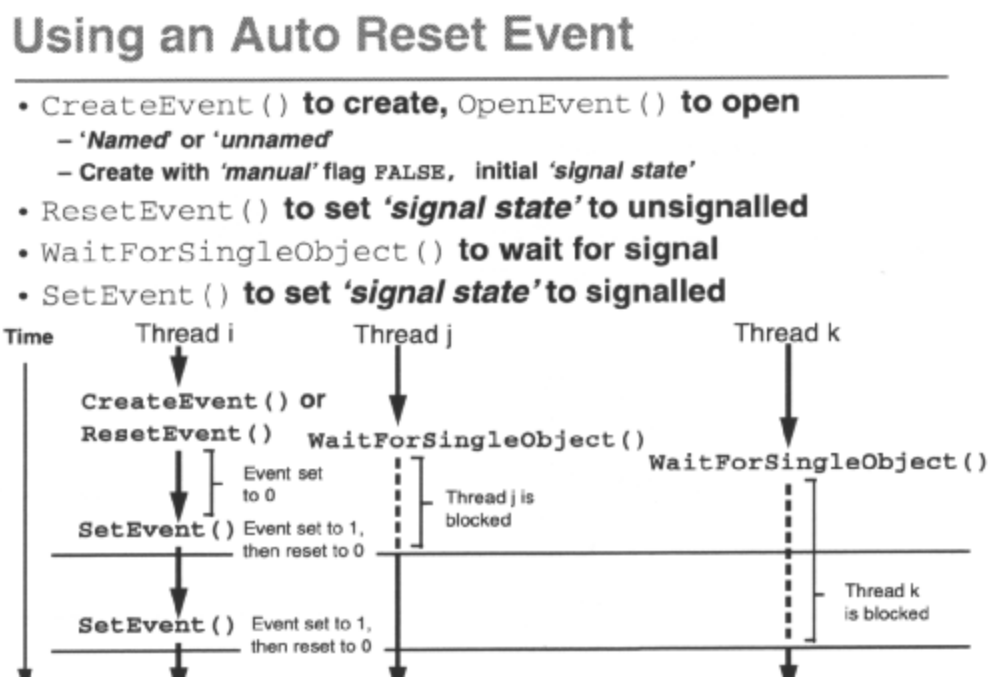
A thread can subsequently set an MR Event by calling `SetEvent()`, which releases all threads waiting on the MR Event, and reset an MR Event by calling `ResetEvent()`, which blocks all threads subsequently attempting to wait.

`SetEvent()` will not automatically reset the MR Event, but `PulseEvent()` will, after releasing all threads waiting on the MR Event.

Threads wait on an Event by calling `WaitForSingleObject()` specifying an Event handle. All waiting threads are suspended until the Event is set or their individual wait timeout expires, whichever is sooner.



The other type of Event is an *'Auto Reset Event'*. When set to signaled, one waiting thread is released and the Event is reset automatically to unsignaled as soon as the thread is released, blocking all other waiters.
This type of Event is used when the signaling thread does not care which of a number of competing threads gets released. For instance, if a print spooler is presiding over a pool of printers it may have a number of threads waiting to dispatch print jobs to different printers.
If a process writing to the print spooler doesn't care which printer the output goes to, perhaps just the first available printer, then it might signal an AR Event when it had prepared its print job. The 'ready' threads in the print spooler wait on this AR Event. It is important that one, and only one, gets released to service the print job.



An Auto Reset Event (AR Event) is created by `CreateEvent()` (setting the *'manual reset'* parameter to `FALSE`) and opened with `OpenEvent()` They both take the name of the Event and return a handle to the Event which is used in all Event API calls. If the name is *NULL*, the Event will be unnamed, and will usually be private to the process that created it.

`CreateEvent()` accepts a *'state'* parameter which specifies whether the Event is initialized to signaled (TRUE) or unsignalled (FALSE). It is normally important to set it to the reset state to avoid premature triggering. Optionally, a security descriptor can be specified, which will dictate how the Event may be accessed. Default access is 'modify' and 'synchronize'.

`OpenEvent()` specifies which kind of access is required to the Event. Asking for all access gives modify and synchronize access.

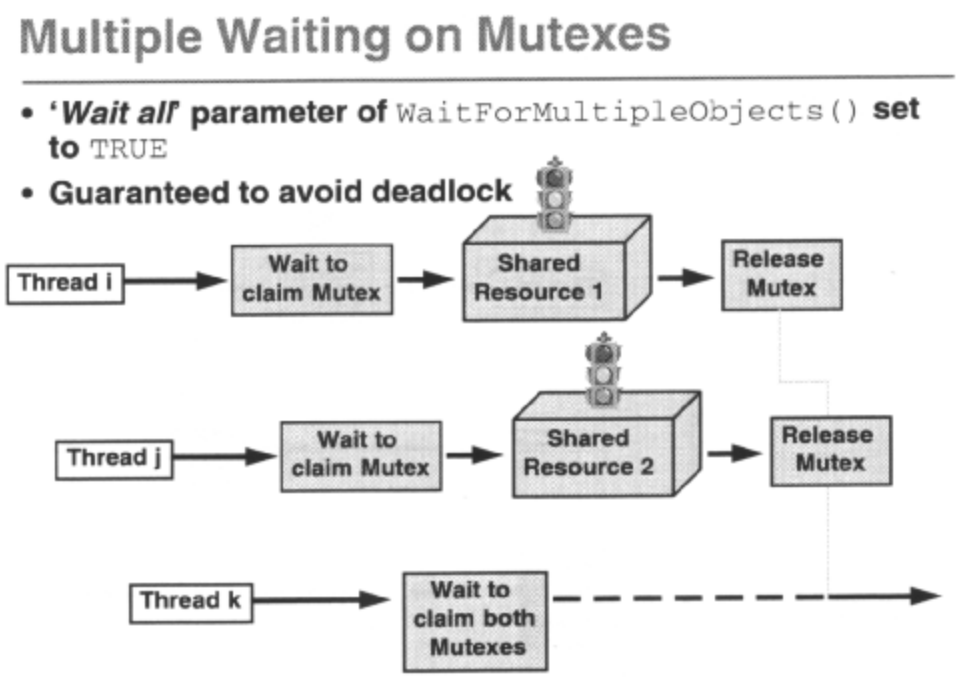
A thread can subsequently set an AR Event by calling `SetEvent()`, which releases one thread waiting on the AR Event, and automatically resets the AR Event. The AR Event can be reset by calling `ResetEvent()` Threads wait on an AR Event by calling `WaitForSingleObject()`, specifying an Event handle. All waiting threads are suspended until the Event is set or their individual wait timeout expires, whichever is sooner.

Here is an example of creating an unnamed AR Event, with default security and initially unsignalled:

```
HANDLE hEvent = CreateEvent(  NULL,       /* no security */
                              FALSE,      /* auto reset */
                              FALSE,      /* not signaled */
                              NULL );     /* no name  */
```

Here is an example of waiting for it indefinitely:

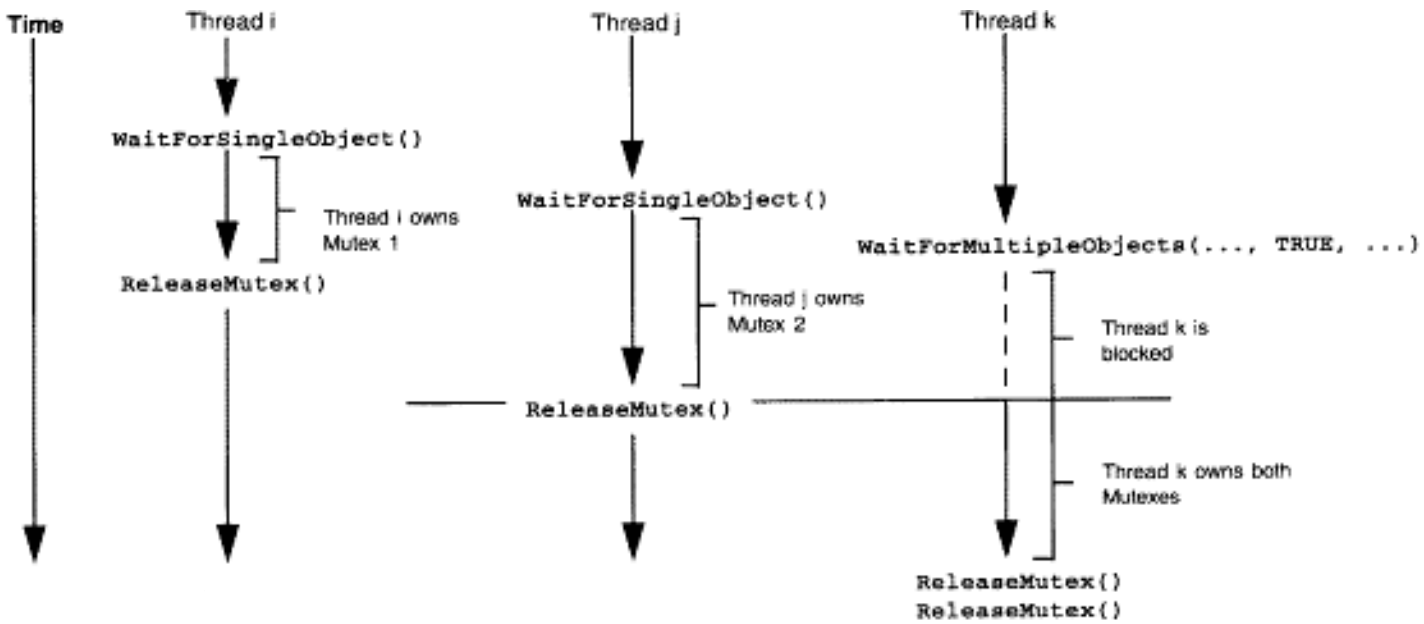
```
WaitForSingleObject( hEvent, INFINITE );
```



Waiting on multiple Mutex objects would typically be used to ensure that a thread had simultaneous exclusive access to two or more resources. To wait for all synchronization objects in a group to become signaled, use `WaitForMultipleObjects()` with a 'wait all' flag set to TRUE.

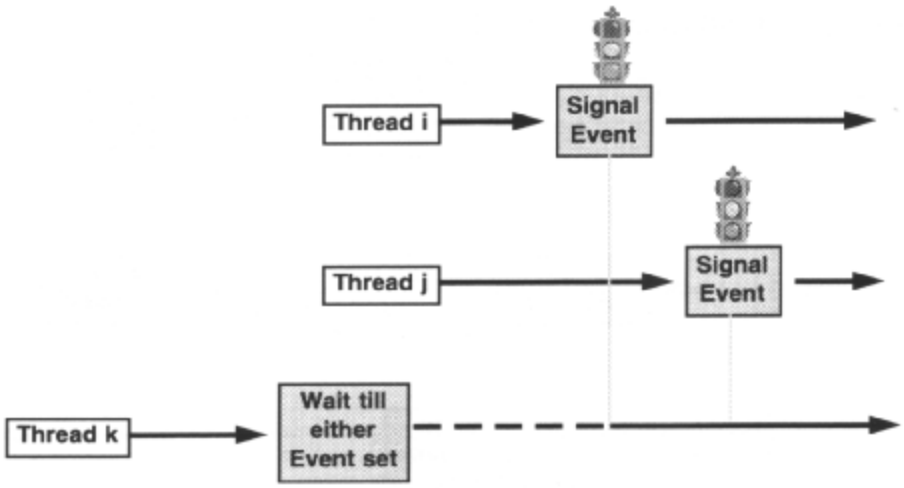
n the diagrams above and below, thread k is waiting for ownership of both shared resource 1 and shared resource 2, each of which is protected by a Mutex object. These Mutexes are currently owned by threads i and j respectively.
Thread k has requested ownership of both of the Mutexes. Thread k is suspended until this dual ownership is established.

When threads i and j relinquish their respective Mutexes and they are both unowned, thread k obtains ownership of both, and prevents any other threads from accessing the associated resources. When it has possession of both Mutexes it has exclusive access to both shared resources. Once it has finished accessing the resources, thread k should release both Mutexes.



Multiple Waiting on Events

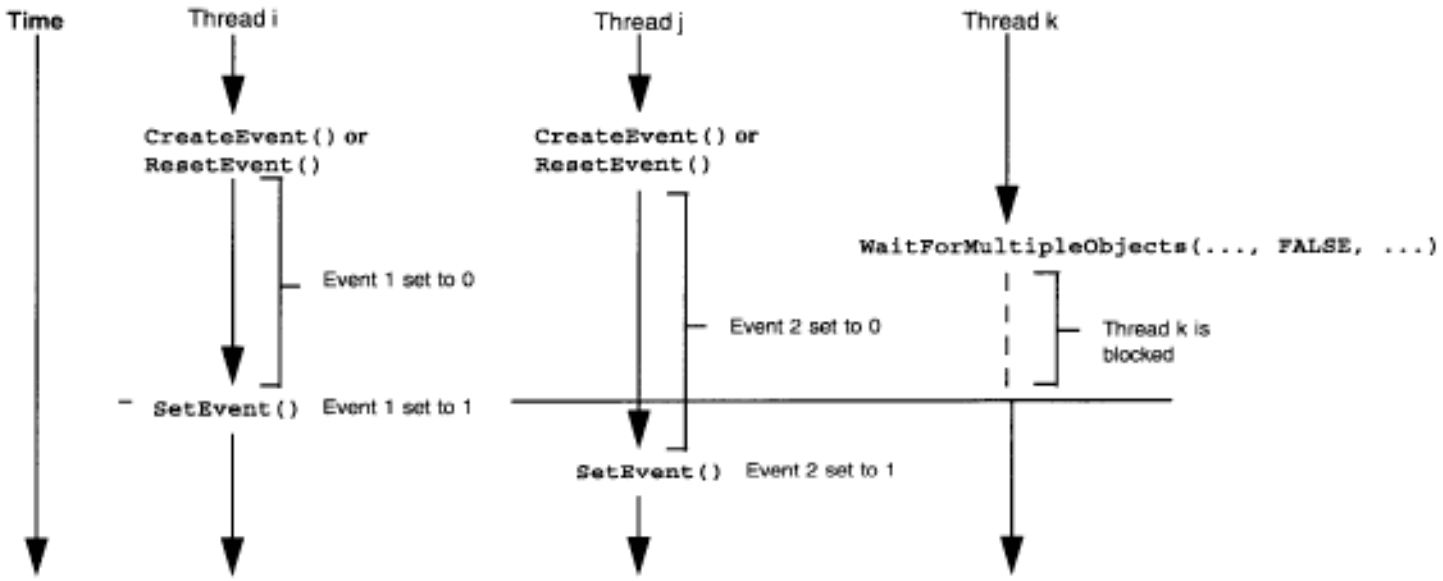
- **'*Wait all*' parameter of** `WaitForMultipleObjects()` **set to** `FALSE`



A thread waiting on the first of a group of Events to be signaled would be suspended until the first of the Events was signaled, and would then resume executing. To wait for one synchronization object in a group to become signaled, use `WaitForMultipleObjects()` with a *'wait all'* flag set to FALSE.

This is the situation illustrated in the diagrams above and below. Thread k might be waiting to respond to data from one of three potential sources. It would perform a compound wait, suspended and therefore not wasting CPU cycles, until one of threads i, j or n signaled an Event, to indicate that data was available.

Note that a wait records that an Event has been signaled, even if it is reset again before the thread waiting on the Event semaphore is next scheduled to run.



Sharing synchronisation Objects

- **Named synchronisation objects**
  - Create with name in one process
  - Open by name in another process
- **Handle duplication**
  - Create unnamed object in one process
  - Pass handle and process ID to another process
  - Duplicate handle in receiving process
- **Handle inheritance**
  - Open handles can be inherited by children
  - Object handles and creating process must allow this

Example of using a named synchronization object:

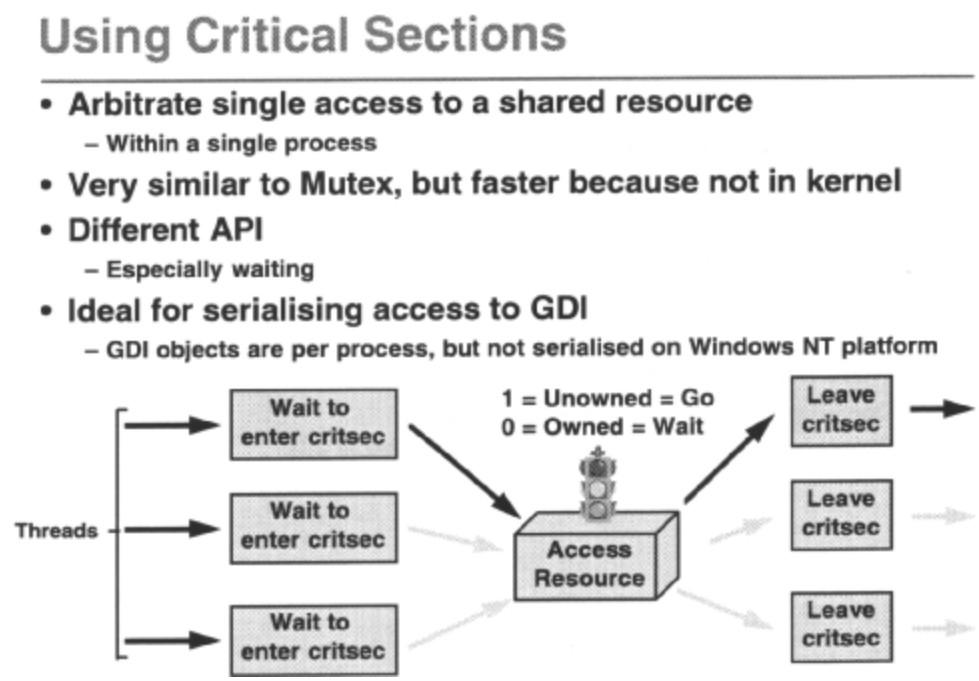
```
/*Any process can try opening, else create if it fails */

HANDLE hMutex = OpenMutex( MUTEX_ALL_ACCESS, FALSE, “sharedmutex” );
if ( !hMutex)
      hMutex = CreateMutex( NULL, FALSE, “sharedmutex”);
```

Example of using an unnamed synchronization object: handle duplication:.

```
/*sending process*/
HANDLE hMutexAnon = CreateMutex( NULL, FALSE, NULL ); /* use IPC to pass
DWORD dwPID = GetCurrentProcessId();      /* these to another process */

/* receiving process */
DWORD dwPID;           /* receive these two from */
HANDLE hMutexAnon;     /* the other process */
HANDLE hMutexDup;
HANDLE hProcessCreate = OpenProcess( PROCESS_DUP_HANDLE, FALSE, dwPID );

DuplicateHandle(  hProcessCreate, hMutexAnon, GetCurrentProcess()
&hMutexDup, 0, FALSE, DUPLICATE_SAME_ACCESS );
```

Example of using an unnamed synchronization object: handle inheritance:

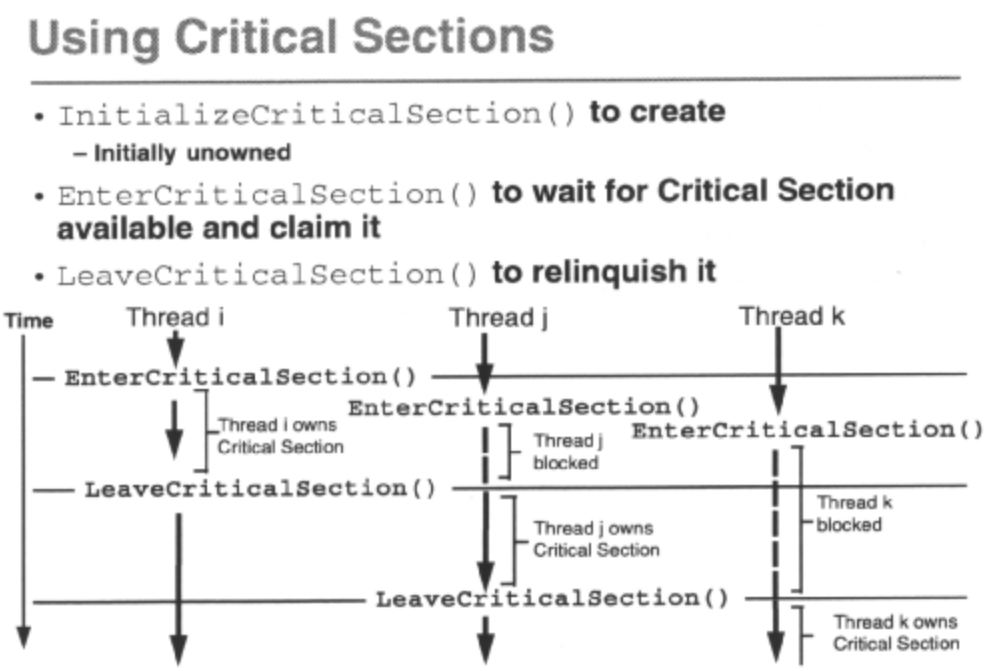
```
/* parent process*/
SECURITY_ATTRIBUTES sa ={sizeof(sa), NULL, TRUE};/*inheritable*/
HANDLE hMutexTolnherit = CreateMutex( &sa, FALSE, NULL);
char szCmdLine [25];
BOOL  blnherit = TRUE;
STARTUPINFO si ={0};
PROCESS_INFORMATION pI;

sprintf( szCmdLine, "%s %d", "child", hMutexTolnherit );
CreateProcess( szCmdLine, NULL, NULL, blnherit, 0, NULL, NULL, &si, &pi );

/* child process*/
HANDLE hMutexInherited;
char szExeName [25];

sscanf( GetCommandLine() "%s %d", sztxeName, &hMutexlnherited )
```



Multiple threads within the same process needing to use a serially reusable resource(SRR), such as a data area, a device, or a section of non-reentrant code, can use a Critical Section to ensure that only one thread at a time has access to the resource. This is similar to a Mutex, but to use a Critical Section the threads must be in the same process. There is no concept of waiting on multiple Critical Sections, nor is it possible to specify a timeout on waiting for one to become unowned. Using a Critical Section is faster than a Mutex because it is purely a Win32 primitive, and not part of the underlying operating system.

Once the Critical Section has been initialized, each thread should request ownership before attempting to access the resource. Only the thread that owns the Critical Section can proceed to enter the protected resource, and it should relinquish ownership as soon as it has finished using it. The system queues subsequent requests for the Critical Section, and transfers ownership to one of the waiting threads as soon as the Critical Section is left. Threads that have requested the Critical Section are blocked until it is their turn to own it.

A typical situation in which one would need to synchronize the operation of different threads in order to guard a SRR, would be when multiple threads have read and write access to a global variable. Each thread would need to request ownership of the Critical Section before it could enter code that performs read / write operations on the variable. When access to the variable was finished, the Critical Section would be relinquished.



A *Critical Section* is created by a call to `InitializeCriticalSection()` This takes as a parameter the address of a variable of type CRITICAL_SECTION. This variable must not be modified, and must be treated as opaque. The address of this variable is used in all subsequent Critical Section API calls. The Critical Section is initialized to unowned.

Threads wishing to use a resource protected by the Critical Section may request ownership by calling `EnterCriticalSection()` If the Critical Section is unowned, the first thread requesting ownership will be given to it, and may then access the resource. When it is finished with the resource, it should relinquish the Critical Section by calling `LeaveCriticalSection()` A thread requesting ownership of an owned Critical Section will be blocked until the Critical Section is unowned once more, and it is that thread's turn to have possession.

While a thread has ownership of a Critical Section, it can repeatedly enter the same Critical Section without blocking. However, it must leave the Critical Section for each satisfied entry, before the Critical Section is unowned.

Once finished with, Critical Sections can be deleted with DeleteCriticalSection(), after which they become invalid. Critical Sections cannot be deleted if they are owned.

Here is an example of creating a Critical Section:

```
CRITICAL_SECTION Cs;
InitializeCriticalSection( &cs);
```

and then to wait on it indefinitely, enter it and relinquish it:

```
EnterCriticalSection( &cs );/*access protected resource*/
LeaveCriticalSection( &cs );
```

and then delete it when finished with:

```
DeleteCriticalSection( &cs );
```

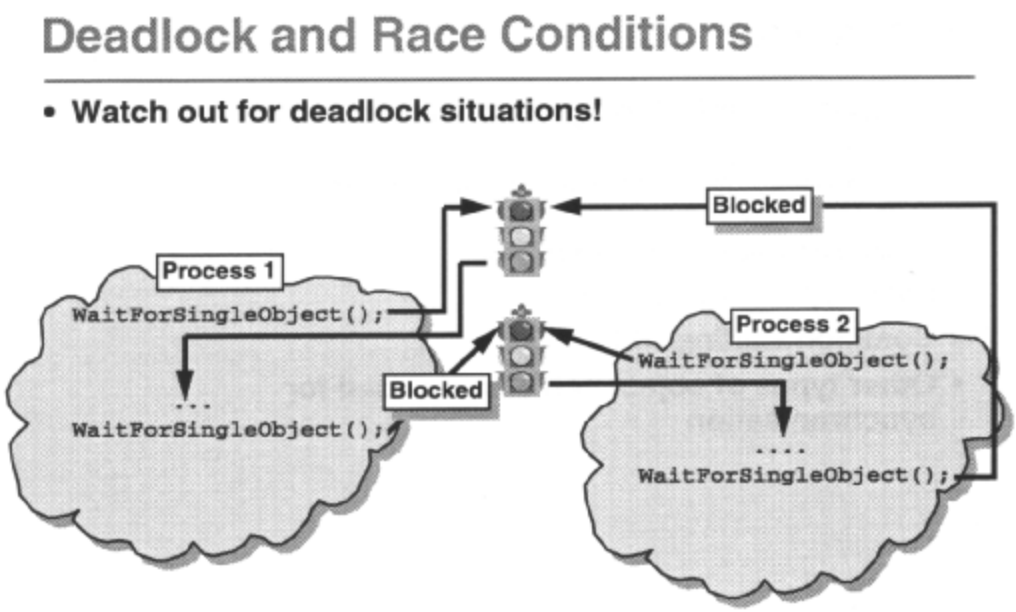
Interlocked Variables

- Atomic operations on a 32-bit variable shared amongst threads
- Increment and test
- Decrement and test
- These are the primitives on which higher-level synchronisation features are built.

`InterlockedIncrement()` and `InterlockedDecrement()` allow synchronized access to a 32-bit variable that is shared by multiple threads in the same or in different processes. These functions automatically increment and test or decrement and test the variable. This avoids the situation where thread I could increment a variable but be interrupted by thread 2, which also increments the same variable, before thread I can check its resulting value.

Obviously, the threads of different processes can use this mechanism only if the variable is in shared memory.

The variable should be 32-bit aligned to work on all Win32 processor configurations. These are the primitives that allow the implementation of higher-level synchronization features.



It is possible to have a situation where two different threads need access to the same two Mutexes simultaneously. If the threads try to claim the Mutexes in the wrong order, they could end up in a position where each has claimed one Mutex and is waiting to claim the other without any possibility of doing so. Thread 1 has claimed Mutex A and is waiting on Mutex B. Thread 2 has claimed Mutex B and is waiting on Mutex A, which it can't get because thread 1 has claimed it and is waiting on Mutex B which has been claimed by Thread 2 which is.... Deadly embrace!

This situation can be simply resolved by using `WaitForMultipleObjects()` with the *'wait all'* flag TRUE. This will not return until it can claim both Mutexes at the same time.

Situations can also occur where threads are racing against each other to complete a task and wait for a synchronization object to become signaled. If one thread has a slightly higher priority it may be the one to keep getting to proceed, in which case it is pointless having multiple threads. Make sure that all threads in your application are getting a chance to execute.

The exact release order of threads of the same, or different priorities is difficult to determine. One might expect that it was based on the priority of the waiting thread or the length of time the thread had been waiting - perhaps it is! It does seem to depend on a number of things - for instance, whether the code is running on Windows *9x* or Windows NT, since the schedulers behave slightly differently, and on the priority class of the waiting threads.

Summary

- 4 types of dedicated synchronisation object
  - Mutex
  - Semaphore
  - Event

  } within or between processes
  - Critical Section — within a process
- Each type has own API
- Common waiting API
- Other types of object can also be used for synchronisation