**Port Windows IPC apps to Linux, Part 3 - Mutexes, critical sections, and wait functions**

*Finishing up with synchronization objects and primitives*

[Srinivasan S. Muthuswamy](http://www.ibm.com/developerworks/linux/library/l-ipc2lin3.html#author1) ([smuthusw@in.ibm.com](mailto:smuthusw@in.ibm.com?subject=Mutexes,%20critical%20sections,%20and%20wait%20functions)), Software Engineer, IBM



Srinivasan S. Muthuswamy works as a Software Engineer for IBM Global Services Group. He joined IBM in 2000, and his expertise in programming reaches from scripting languages to object- and procedure-oriented languages on multiple platforms (Linux, Windows, WebSphere, Lotus, and so on). Muthuswamy has developed solutions ranging from system programming on Linux and Windows to Web solutions for J2EE. His primary focus is on integration and porting, and he holds a B.Eng. in Computer Engineering from the Government College of Technology, Coimbatore, India.

[Kavitha Varadarajan](http://www.ibm.com/developerworks/linux/library/l-ipc2lin3.html#author2) ([vkavitha@in.ibm.com](mailto:vkavitha@in.ibm.com?subject=Mutexes,%20critical%20sections,%20and%20wait%20functions)), Software Engineer, IBM



Kavitha Varadarajan has worked as a software Engineer in the IBM India Software Lab from December 2000. Her work experience involves development and support of host-access client products such as PCOMM and networking software like the communication server. Varadarajan has experience with a migration project that involves porting object-oriented IPC Windows applications to Linux. She holds a B.Eng. in Computer Science and Engineering from Shanmugha College of Engineering, Tanjore, India.

**Summary:**  The wave of migration to open source in business has the potential to cause a tremendous porting traffic jam as developers move the pervasive Windows® applications to the Linux™ platform. In this [three-part series](http://www-128.ibm.com/developerworks/views/linux/libraryview.jsp?search_by=port+windows+ipc+apps+linux), you get a mapping guide, complete with examples, to ease your transition from Windows to Linux. This part takes a look at mutexes, critical sections, and wait functions.

[View more content in this series](http://www.ibm.com/developerworks/views/linux/libraryview.jsp?search_by=port+windows+ipc+apps+linux)

[Tag this!](http://www.ibm.com/developerworks/linux/library/l-ipc2lin3.html)

[Update My dW interests](http://www.ibm.com/developerworks/linux/library/l-ipc2lin3.html) ([Log in](https://www.ibm.com/developerworks/dwwi/DWAuthRouter?m=loginpage&d=http%3A%2F%2Fwww.ibm.com%2Fdeveloperworks%2Flinux%2Flibrary%2Fl-ipc2lin3.html) | [What's this?](http://www.ibm.com/developerworks/linux/library/l-ipc2lin3.html#overlay)) [Skip to help for Update My dW interests](http://www.ibm.com/developerworks/linux/library/l-ipc2lin3.html#dwmyinterestaddhelp)

**Date:**  25 Aug 2005   
**Level:**  Advanced   
  
**Activity:**  10742 views   
**Comments:**   1 ([View or add comments](http://www.ibm.com/developerworks/linux/library/l-ipc2lin3.html#icomments)) 

[1 star2 stars3 stars4 stars5 stars](javascript:void(0);)Average rating (based on 55 votes)

Today, many global businesses and services are going open source -- all the major corporate players in the industry are pushing for it. This trend has spurred a major migration exercise in which lots of existing products maintained for various platforms (Windows, OS2, Solaris, etc.) will be ported to open source Linux platforms.

Many applications are designed without considering the need to port them to Linux. This has the potential to be a porting nightmare, but it doesn't have to be. The goal of this series of articles is to help you migrate complex applications involving IPC and threading primitives from Windows to Linux. We will share our experiences in moving these critical Windows IPC applications, applications that include multithreaded apps that require thread syncronization and multiprocess apps that require interprocess syncronization.

In short, this series can be called a mapping document -- it provides mapping of various Windows calls to Linux calls related to threads, processes, and interprocess communication elements (mutexes, semaphores, etc.). To create easily digestible chunks, we've divided the series into three articles:

* [Part 1](http://www.ibm.com/developerworks/linux/library/l-ipc2lin1.html) dealt with processes and threads.
* [Part 2](http://www.ibm.com/developerworks/linux/library/l-ipc2lin2.html) handled semaphores and events.
* This part covers mutexes, critical sections, and wait functions.

Let's finish our Windows-to-Linux mapping guide by starting with mutexes.

Mutexes

A mutex (which stands for "mutual exclusion" lock) is a locking or synchronization object that allows multiple threads to synchronize access to shared resources. It is often used to ensure that shared variables are always seen by other threads in a consistent state.

In Windows, the mutexes are both named and un-named. The named mutex is shared between the threads of different process.

In Linux, the mutexes are shared only between the threads of the same process. To achieve the same functionality in Linux, a System V semaphore can be used (see [Resources](http://www.ibm.com/developerworks/linux/library/l-ipc2lin3.html#resources) for a link to Part 2 of this series).

In Windows, wait functions are used to request the ownership of the mutex. There are different types of wait functions available -- the one we're using as an example is WaitForSingleObject().

The following points should be considered in the mapping process of a mutex:

* In Windows, a mutex can be named and un-named. A named mutex is shared across the process. In Linux, mutexes are shared only among the threads. System V semaphores can be used to provide the named mutex functionality in Linux.
* In Windows, a mutex can be owned during creation; this support is not available in Linux. To achieve the same in Linux, a mutex should be locked explicitly after creation.
* In Windows, timeout can be specified in the wait functions. In Linux, the timeout option is not available. This is handled in application logic.
* A Windows mutex is recursive by default. A Linux mutex needs to have recursion explicitly set. System V semaphores are not recursive.

|  |  |  |  |
| --- | --- | --- | --- |
| *Table 1. Mutex mapping* | | | |
| **Windows** | **Linux threads** | **Linux process** | **Classification** |
| CreateMutex | pthreads\_mutex\_init | semget semctl | context specific |
| OpenMutex | not applicable | semget | context specific |
| WaitForSingleObject | pthread\_mutex\_lock pthread\_mutex\_trylock | semop | context specific |
| ReleaseMutex | pthread\_mutex\_unlock | semop | context specific |
| CloseHandle | pthread\_mutex\_destroy | semctl | context specific |

Creating a mutex

In Windows, CreateMutex() is used to create or open a named or un-named mutex object. Named mutexes are mainly used to provide synchronization between processess: HANDLE CreateMutex (LPSECURITY\_ATTRIBUTES lpMutexAttributes, BOOL bInitialOwner, LPCTSTR lpName). In this code:

* lpMutexAttributes is a pointer to the structure that determines whether the handle can be inherited by the child process or not. If this attribute is null, the handle cannot be inherited.
* bInitialOwner is a boolean value and if this value is TRUE then the calling thread initially owns the mutex.
* lpName is a pointer to the name of the semaphore. If null, then un-named semaphore is created.

In Windows, OpenMutex() is used to open the named mutex. This function returns the handle of the mutex.

|  |
| --- |
| HANDLE OpenMutex(  DWORD dwDesiredAccess,  BOOL bInheritHandle,  LPCTSTR lpName  ) |

In the code:

* dwDesiredAccess is the desired access for the user requesting for the mutex object.
* bInheritHandle is a flag and if true, related process can inherit the handle.
* lpName is the name of the mutex (and is case sensitive).

Notice in this code that the named mutex should have been created already.

In Linux, the pthread library call pthread\_mutex\_init() is used to create the mutex: int pthread\_mutex\_init(pthread\_mutex\_t \*mutex, const pthread\_mutexattr\_t \*mutexattr).

There are three kinds of mutexes in Linux, each type determined by what happens if a thread attempts to lock a mutex it already owns with pthread\_mutex\_lock():

* A fast mutex. While trying to lock the mutex using pthread\_mutex\_lock() the calling thread suspends forever.
* A recursive mutex. pthread\_mutex\_lock() returns immediately with a success return code. This is used as equivalent for a Windows mutex since it is recursive in nature.
* An error check mutex. pthread\_mutex\_lock() returns immediately with the error code EDEADLK.

The mutex kind can be set in two ways. The static way of setting is as follows:

|  |
| --- |
| /\* Fast \*/  pthread\_mutex\_t mutex = PTHREAD\_MUTEX\_INITIALIZER;  /\* Recursive \*/  pthread\_mutex\_t recmutex = PTHREAD\_RECURSIVE\_MUTEX\_INITIALIZER\_NP;  /\* Errorcheck \*/  pthread\_mutex\_t errchkmutex = PTHREAD\_ERRORCHECK\_MUTEX\_INITIALIZER\_NP; |

Another way of setting mutex kind is by using a mutex attribute object. To do this, pthread\_mutexattr\_init() is called to initialize the object followed by pthread\_mutexattr\_settype() which sets the kind of the mutex.

|  |
| --- |
| int pthread\_mutexattr\_init(pthread\_mutexattr\_t \*attr);  int pthread\_mutexattr\_settype(pthread\_mutexattr\_t \*attr, int kind); |

The parameter kind takes the following values:

* PTHREAD\_MUTEX\_FAST\_NP
* PTHREAD\_MUTEX\_RECURSIVE\_NP
* PTHREAD\_MUTEX\_ERRORCHECK\_NP

The attribute can be destroyed using pthread\_mutexattr\_destroy(): int pthread\_mutexattr\_destroy(pthread\_mutexattr\_t \*attr);.

Setting the initial state of the mutex

In Linux the initial state of the mutex cannot be set using the pthread\_mutex\_init() call. This can be achieved by following steps:

1. Create a mutex using pthread\_mutex\_init().
2. Lock/acquire the mutex using pthread\_mutex\_lock().

Acquiring a mutex

In Windows wait funtions provide the facility to acquire the synchronization objects. There are different types of wait functions -- in this section we're using WaitForSingleObject(). This function takes the handle to the mutex object and waits until it is signaled or timeout occurs.

|  |
| --- |
| DWORD WaitForSingleObject(  HANDLE hHandle,  DWORD dwMilliseconds  ); |

In this code:

* hHandle is the pointer to the mutex handle.
* dwMilliseconds is the timeout value in milliseconds. If the value is INFINITE then it blocks the calling thread/process indefinitely.

In Linux, the pthread library call pthread\_mutex\_lock() / pthread\_mutex\_trylock() is used to acquire the mutex.

|  |
| --- |
| int pthread\_mutex\_lock(pthread\_mutex\_t \*mutex);  int pthread\_mutex\_trylock(pthread\_mutex\_t \*mutex); |

pthread\_mutex\_lock() is a blocking call which means that if the mutex is already locked by another thread, pthread\_mutex\_lock() suspends the calling thread until the mutex is unlocked. On the other hand, pthread\_mutex\_trylock() returns immediately if the mutex is already locked by another thread. Remember that in Linux, the timeout option is not available. This can be achieved by issuing a non-blocking pthread\_mutex\_trylock() call along with a delay in a loop which counts the timeout value.

Releasing a mutex

In Windows the function ReleaseMutex() releases the ownership of the mutex and sets the mutex to the signaled state: BOOL ReleaseMutex(HANDLE hMutex). In this code, hMutex is the handle of the mutex.

Notice that mutexes in Windows are basically recursive mutexes -- if the thread which already owns the mutex tries to acquire ownership again, the wait function returns without blocking and deadlock is avoided.

Linux uses pthread\_mutex\_unlock() to release/unlock the mutex: int pthread\_mutex\_unlock(pthread\_mutex\_t \*mutex);.

The mutex functions are not asynchronous signal-safe and should not be called from a signal handler. In particular, calling pthread\_mutex\_lock or pthread\_mutex\_unlock from a signal handler may deadlock the calling thread.

Closing/destroying a mutex

In Windows, CloseHandle() is used to close or destroy the mutex object.

|  |
| --- |
| BOOL CloseHandle(  HANDLE hObject  ); |

In the code, hObject is the pointer to the handle to the synchronization object.

In Linux, pthread\_mutex\_destroy() destroys a mutex object, freeing the resources it might hold. It also checks to determine whether the mutex is unlocked at that time: int pthread\_mutex\_destroy(pthread\_mutex\_t \*mutex).

Named mutex

In Windows, named mutexes are mainly used between processes to achieve synchronization (to access the shared resource in a mutually exclusive manner). The mutex provided by the Linux threads libraries are limited to the threads of the same process. To achieve the same functionality between processes in Linux, a System V semaphore can be used.

System V semaphores are count variables. To achieve the same function as the Windows named mutex, the initial count of the semaphore is set to 0 using semctl() function. To acquire mutually exclusive access to the shared resource, semop() is used with sem\_op value as -1. The calling process is blocked until mutually exclusive access is released. The creating process can acquire the mutually exclusive access by creating a semaphore with the initial count as 0 using semctl() function. After using the shared resource, the semaphore count can be set to 1 by using semop() function, allowing the other processes to access the shared resource. (See [Resources](http://www.ibm.com/developerworks/linux/library/l-ipc2lin3.html#resources) for a link to the Part 2 section on semaphores.)

Examples

Following is code samples dealing with mutexes. Listings 13 and 14 show two scenarios each. In the first, a mutex is used without a specified timeout value. In the second, a mutex is used with a timeout value of two seconds.

**Listing 1. Windows example for un-named mutex**

|  |
| --- |
| HANDLE hMutexWithNoTimeOut, hMutexWithTimeOut;  DWORD dwRetCode;  // create a mutex  hMutexWithNoTimeOut = CreateMutex(  NULL, // no security attriutes  FALSE, // initially not owned  NULL); // un named mutex so NULL  hMutexWithTimeOut = CreateMutex(  NULL, // no security attriutes  FALSE, // initially not owned  NULL); // un named mutex so NULL  // acquire a mutex  dwRetCode = WaitForSingleObject(  hMutexWithNoTimeOut, // Mutex handle  INFINITE); // Infinite wait  if (dwRetCode == WAIT\_OBJECT\_0) {  // success  // access the shared resource  .....  // release mutex  ReleaseMutex(hMutexWithNoTimeOut); // Mutex Handle  }  // case 2, using mutex with timeout specified.  dwRetCode = WaitForSingleObject(  hMutexWithTimeOut,  2000L); // 2 secs timeout  switch(dwRetCode) {  case WAIT\_OBJECT\_0 :  // success  // access the shared resource  .....  // After using the shared resource, release the semaphore  ReleaseMutex(hMutexWithTimeOut);  ....  break;  case WAIT\_TIMEOUT :  // Handle the timeout case  ...  break;  case WAIT\_ABANDONED :  // probe for abandoned mutex  break;  }  ....  // close all the mutex  CloseHandle(hMutexWithTimeout);  CloseHandle(hMutexWithNoTimeout); |

**Listing 2. Equivalent Linux code**

|  |
| --- |
| #define TIMEOUT 200 // 2 Secs delay time  struct timespec delay; // structure for providing timeout  pthread\_mutexattr\_t mutexattr; // Mutex Attribute  pthread\_mutex\_t mutexWithNoTimeOut, mutexWithTimeOut; // Mutex variables  // Set the mutex as a recursive mutex  pthread\_mutexattr\_settype(&mutexattr, PTHREAD\_MUTEX\_RECURSIVE\_NP);  // Create the mutex with the attributes set  pthread\_mutex\_init(&mutexWithNoTimeOut, &mutexattr);  pthread\_mutex\_init(&mutexWithTimeOut, &mutexattr);  // destroy the attribute  pthread\_mutexattr\_destroy(&mutexattr)  // Lock/Acquire the mutex and access the shared resource  pthread\_mutex\_lock (&mutexWithNoTimeOut);  // access the shared resource  .. ...  // Unlock the mutex  pthread\_mutex\_unlock (&mutexWithNoTimeOut);  ...  // Case 2, Accessing share resource with time out value specified in the  // Mutex call  while (timeout < TIMEOUT ) {  delay.tv\_sec = 0;  delay.tv\_nsec = 1000000; // 1 milli sec delay  // Tries to acquire the mutex and access the shared resource,  // if success, access the shared resource,  // if the shared reosurce already in use, it tries every 1 milli sec  // to acquire the resource  // if it does not acquire the mutex within 2 secs delay,  // then it is considered to be failed  irc = pthread\_mutex\_trylock(&mutexWithTimeOut);  if (!irc) {  // Acquire mutex success  // Access the shared resource  // Unlock the mutex and release the shared resource  pthread\_mutex\_unlock (&mutexWithTimeOut);  break;  }  else {  // check whether somebody else has the mutex  if (irc == EPERM ) {  // Yes, Resource already in use so sleep  nanosleep(&delay, NULL);  timeout++ ;  }  else{  // Handle error condition  }  }  }  // Close all the mutex  pthread\_mutex\_destroy (&mutexWithNoTimeOut);  pthread\_mutex\_destroy (&mutexWithTimeOut); |

**Listing 3. Windows example for named mutex**

|  |
| --- |
| **// Process 1**  HANDLE hMutex;  DWORD dwRetCode;  // create a mutex  hMutex = CreateMutex(  NULL, // no security attriutes  FALSE, // initially not owned  NULL); // un named mutex so NULL  // acquire a mutex  dwRetCode = WaitForSingleObject(  hMutex, // Mutex handle  INFINITE); // Infinite wait  if (dwRetCode == WAIT\_OBJECT\_0) {  // success  // access the shared resource  .....  // release mutex  ReleaseMutex(hMutex); // Mutex Handle  }  // close the mutex  CloseHandle(hMutex);  **// Process 2**  HANDLE hMutex;  DWORD dwRetCode;  // Open the mutex created by the Process 1  hMutex = OpenMutex(  NULL, // no security attriutes  NULL, // handle cannot be inhereited  "testMuex"); // named mutex  // acquire a mutex  dwRetCode = WaitForSingleObject(  hMutex, // Mutex handle  INFINITE); // Infinite wait  if (dwRetCode == WAIT\_OBJECT\_0) {  // success  // access the shared resource  .....  // release mutex  ReleaseMutex(hMutex); // Mutex Handle  }  // close the mutex  CloseHandle(hMutex); |

**Listing 4. Linux equivalent code**

|  |
| --- |
| **// Process 1**  #define TIMEOUT 200  int main()  {  //Definition of variables  key\_t key;  int semid;  int Ret;  int timeout = 0;  struct sembuf operation[1] ;  union semun  {  int val;  struct semid\_ds \*buf;  USHORT \*array;  } semctl\_arg,ignored\_argument;  key = ftok(); //Generate a unique key or supply a value  semid = semget(key, // a unique identifier to identify semaphore set  1, // number of semaphore in the semaphore set  0666 | IPC\_CREAT // permissions (rwxrwxrwx) on the new  //semaphore set and creation flag  );  if(semid < 0)  {  printf("Create semaphore set failed ");  Exit(1);  }  //Set Initial value for the resource  semctl\_arg.val = 1; //Setting semval to 1  semctl(semid, 0, SETVAL, semctl\_arg);  //Wait for Zero  while(timeout < TIMEOUT)  {  delay.tv\_sec = 0;  delay.tv\_nsec = 1000000; /\* 1 milli sec \*/  //Call Wait for Zero with IPC\_NOWAIT option,so it will be  // non blocking  operation[0].sem\_op = -1; //Wait  operation[0].sem\_num = subset;  operation[0].sem\_flg = IPC\_NOWAIT;  ret = semop(semid, operation,1);  if(ret < 0)  {  /\* check whether somebody else has the mutex \*/  if (retCode == EPERM )  {  /\* sleep for delay time \*/  nanosleep(&delay, NULL);  timeout++ ;  }  else  {  printf("ERROR while wait ");  break;  }  }  else  {  /\*semaphore got triggered \*/  break;  }  }  //Close semaphore  iRc = semctl(semid, 1, IPC\_RMID , ignored\_argument);  }  **// Process 2**  int main()  {  int key = 0x20; //Process 2 shd know key value in order to open the  // existing semaphore set  struct sembuf operation[1] ;  //Open semaphore  semid = semget(key, 1, 0);  operation[0].sem\_op = 1; //Release the resource so Wait in process  // 1 will be triggered  operation[0].sem\_num = 0;  operation[0].sem\_flg = SEM\_UNDO;  //Release semaphore  semop(semid, operation,1);  } |

[Back to top](http://www.ibm.com/developerworks/linux/library/l-ipc2lin3.html#ibm-pcon)

Critical sections

In Windows, critical sections are synchronization objects that are similar to mutexes but with some limitations. The critical sections can only be used between threads of same process. A mutex uses timeout when requesting access to the mutex, but a critical section does not provide such feature -- it waits indefinitely.

The critical section uses the *spin count*. In the single-processor system, the spin count is ignored and initialized to 0, but in the multiprocessor system, the calling thread will spin dwSpinCount times before it actually waits for the critical section so that if the critical section becomes free during that time, the calling thread does not wait. Since the critical sections are used only between the threads of the same process, a pthreads mutex is used to achieve the same results on Linux systems.

|  |  |  |
| --- | --- | --- |
| *Table 2. Critical section mapping* | | |
| **Windows** | **Linux** | **Classification** |
| InitializeCriticalSection InitializeCriticalSectionAndSpinCount | pthread\_mutex\_init | mappable |
| EnterCriticalSection TryEnterCriticalSection | pthread\_mutex\_lock pthread\_mutex\_trylock | mappable |
| LeaveCriticalSection | pthread\_mutex\_trylock | mappable |
| DeleteCriticalSection | pthread\_mutex\_destroy | mappable |

Creating/initializing a critical section

In Windows, critical sections need to be initialized before the threads of same process can actually be used. InitializeCriticalSection() or InitializeCriticalSectionAndSpinCount() can be used to initialize the critical section.

|  |
| --- |
| void InitializeCriticalSection(  LPCRITICAL\_SECTION lpCriticalSection  ) |

In this code, lpCriticalSection is a pointer to the handle to the critical section. In low memory situations, this function raises the STATUS\_NO\_MEMORY exception.

InitializeCriticalSectionAndSpinCount() is used to initialize and set the spin count.

|  |
| --- |
| BOOL InitializeCriticalSectionAndSpinCount(  LPCRITICAL\_SECTION lpCriticalSection,  DWORD dwSpinCount  ) |

In this code:

* lpCriticalSection is a pointer to the handle of the critical section.
* dwSpinCount is the spin count for the critical section.

In Linux, pthread\_mutex\_init() is used to create or initialize the mutex object.

Entering/acquiring a critical section

In Windows, EnterCriticalSection() or TryEnterCriticalSection() is used to request the ownership of the critical section. If the critical section is already in use, EnterCriticalSection() will block the calling thread, but TryEnterCriticalSection() will attempt to enter the critical section without blocking the thread.

EnterCriticalSection() is used to enter the critical section:

|  |
| --- |
| void EnterCriticalSection(  LPCRITICAL\_SECTION lpCriticalSection  ); |

In this code, lpCriticalSection is a pointer to the handle of the critical section.

The TryEnterCriticalSection() function attempts to enter a critical section without blocking. If the call is successful, the calling thread takes ownership of the critical section:

|  |
| --- |
| BOOL TryEnterCriticalSection(  LPCRITICAL\_SECTION lpCriticalSection  ) |

In this code, lpCriticalSection is the pointer to the handle of the critical section.

In Linux, the same can be achieved using pthread\_mutex\_lock() which blocks the calling thread. pthread\_mutex\_trylock attempts to acquire the mutex without blocking the thread.

Leaving/releasing a critical section

In Windows, LeaveCriticalSection() is used to release the ownership of the critical section. LeaveCriticalSection() needs to be called as many times as the critical section is entered.

void LeaveCriticalSection( LPCRITICAL\_SECTION lpCriticalSection )

In this code, lpCriticalSection is the pointer to the handle of the critical section.

In Linux, pthread\_mutex\_unlock() is used to release the ownership of the mutex object.

Deleting a critical section

In Windows, DeleteCriticalSection() is used to delete the critical section; once used, this critical section can no longer be used for synchronization.

|  |
| --- |
| void DeleteCriticalSection(  LPCRITICAL\_SECTION lpCriticalSection  ) |

In this code, lpCriticalSection is a pointer to the handle of the critical section.

In Linux, pthread\_mutex\_destroy() is used to delete the mutex object.

Example

The following examples are very simple -- they present code snippets for accessing a shared resource using a critical section for mutual exclusion.

**Listing 5. Windows critical section example**

|  |
| --- |
| CRITICAL\_SECTION csCriticalSection;  // Initialize a Critical Section  InitializeCriticalSection(  &csCriticalSection); // Critical Section Object  // Enter a critical Section  EnterCriticalSection(  &csCriticalSection); // Critical Section Object  // Access a shared resource  // Leave a Critical Section  LeaveCriticalSection(  &csCriticalSection); // Critical Section Object  // Delete a Critical Section  DeleteCriticalSection(  &csCriticalSection); // Critical Section Object |

**Listing 6. Equivalent Linux code**

|  |
| --- |
| pthread\_mutex\_t mutex; // Mutex  pthread\_mutexattr\_t mutexattr; // Mutex attribute variable  // Set the mutex as a recursive mutex  pthread\_mutexattr\_settype(&mutexattr, PTHREAD\_MUTEX\_RECURSIVE\_NP);  // create the mutex with the attributes set  pthread\_mutex\_init(&mutex, &mutexattr);  //After initializing the mutex, the thread attribute can be destroyed  pthread\_mutexattr\_destroy(&mutexattr)  // Acquire the mutex to access the shared resource  pthread\_mutex\_lock (&mutex);  // access the shared resource  ..  // Release the mutex and release the access to shared resource  pthread\_mutex\_unlock (&mutex);  ...  // Destroy / close the mutex  irc = pthread\_mutex\_destroy (&mutex); |

[Back to top](http://www.ibm.com/developerworks/linux/library/l-ipc2lin3.html#ibm-pcon)

Wait functions

In Windows, the wait functions block the calling thread/process until the specified criteria is met -- in other words, they allow threads to block their own executions. The type of the wait function determines the set of wait criteria used. There are four types of wait functions:

* Single object (requires a handle to one synchronization object; returns when either the specified object is in the signaled state or the timeout interval elapses).
* Multiple object (enables the calling thread to specify an array containing one or more synchronization object handles; returns when either the state of any one of the specified objects is set to signaled or the states of all objects have been set to signaled or the timeout interval elapses).
* Alertable (the function can return when the specified conditions are met, but it can also return if the system queues an I/O completion routine or an APC for execution by the waiting thread).
* Registered (a multiple wait operation, when the specified conditions are met, the callback function is executed by a worker thread from the thread pool).

We won't be addressing alertable or registered wait functions in this series.

In Linux, wait functions are provided in the respective synchronization library itself (mutexes and semaphores have their own wait functions.)

The following points should be considered when mapping wait functions:

* Windows supports multiple object wait functionality. It allows passing in of multiple synchronization in the same wait function. In Linux, this functionality is not available. This logic needs to be implemented in the application logic.
* Windows supports alertable and registered waits; Linux provides only basic wait functionality. These features can be handled in the application logic for Linux.

|  |  |  |  |
| --- | --- | --- | --- |
| *Table 3. Wait function mapping* | | | |
| **Windows** | **Linux threads** | **Linux process** | **Classification** |
| SignalObjectAndWait | semop | semop | context specific |
| WaitForMultipleObjects | sem\_wait sem\_trywait | semop | context specific |

Signaling and waiting

SignalObjectAndWait() is also an alertable wait function and it is different as it signals an object and waits on another object in an atomic manner.

|  |
| --- |
| DWORD SignalObjectAndWait(  HANDLE hObjectToSignal,  HANDLE hObjectToWaitOn,  DWORD dwMilliseconds,  BOOL bAlertable  ) |

In this code:

* hObjectToSignal is a pointer to the handle to the object to be signaled.
* hObjectToWaitOn is a pointer handle to the object for which the thread has to wait.
* dwMilliseconds is the time out specified in milliseconds.
* bAlertable is a flag and if this parameter is TRUE, then the wait function returns when the system queues an I/O completion routine or APC function and the thread calls the function. For this article we can ignore this flag

In Linux, the same functionality can be achieved by using System V semaphores. These semaphores provide function in which we can specify operation sets. To signal an object and wait for another synchronization object, we can create two operations sets -- one for signaling the object and other to wait on the specified object. The operations sets are performed in an atomic manner, meaning the semop() call succeeds if both the operations succeed; otherwise, it fails.

System V semaphores do not provide timeout functionality. This can be implemented in application logic as we discussed in Part 2 of this series by making semop() call with flag IPC\_NOWAIT. By doing it this way, the calling thread or process is not blocked.

Examples

The following examples should illustrated the wait functions we've discussed.

**Listing 7. Windows example for SignalObjectAndWait()**

|  |
| --- |
| **// Main thread**  HANDLE hEventOne; // Global Variable  HANDLE hEventTwo; // Global Variable  **// Thread 1**  DWORD dwRetCode;  // Create Event One  hEventOne = CreateEvent(  NULL, // no security attributes  TRUE, // Auto reset event  FALSE, // initially set to non signaled state  NULL); // un named event  // Create Event Two  hEventTwo = CreateEvent(  NULL, // no security attributes  TRUE, // Auto reset event  FALSE, // initially set to non signaled state  NULL); // un named event  // Signal hEventOne and Wait for the hEventTwo to be signaled  dwRetCode = SignalObjectAndWait(  hEventOne, // Object to be signaled  hEventTwo, // Object to wait on  INFINITE, // Infinite wait  FALSE); // Not alertable  switch(dwRetCode) {  case WAIT\_OBJECT\_O :  // Event is signaled  // go ahead and proceed the work  default :  // Probe for error  }  // Completed the job,  // now close the event handle  CloseHandle(hEventOne);  CloseHandle(hEventTwo);  **// Thread 2**  // Condition met for the event hEventTwo  // now set the event  SetEvent(  hEventTwo); // Event Handle |

**Listing 8. Linux equivalent using System V semaphores**

|  |
| --- |
| **// Main thread**  int key = 0x20; // Semaphore key  **// Thread 1**  struct sembuf operation[2] ;  // Create 2 semaphores  semid = semget(key, 2, 0666 | IPC\_CREAT);  operation[0].sem\_op = 1; //Release first resource  operation[0].sem\_num = 0;  operation[0].sem\_flg = SEM\_UNDO;  operation[0].sem\_op = -1; // Wait on the second resource  operation[0].sem\_num = 1;  operation[0].sem\_flg = SEM\_UNDO;  //Release semaphore 1 and wait on semaphore 2  // note : thread is suspended until the semaphore 2 is released.  semop(semid, operation, 2);  // thread is released  // delete the semaphore  semctl(semid, 0, IPC\_RMID , 0)  **// Thread 2**  struct sembuf operation[1] ;  // open semaphore  mysemid = semget(key, 2, 0);  operation[0].sem\_op = 1; // Release on the second resource  operation[0].sem\_num = 1;  operation[0].sem\_flg = SEM\_UNDO;  //Release semaphore 2  semop(semid, operation, 1); |

Waiting on an array

WaitForMultipleObjects() is the simplest function available in this type. This function takes an array on one or more synchronized objects as input and blocks the calling thread until any of the following criteria is met:

* Either any one or all of the specified objects are in the signaled state.
* The timeout interval elapses.

|  |
| --- |
| DWORD WaitForMultipleObjects(  DWORD nCount,  const HANDLE\* lpHandles,  BOOL bWaitAll,  DWORD dwMilliseconds  ) |

In this code:

* nCount is the number of object handles in the array pointed to by lpHandles.
* lpHandles is a pointer to array of object handles.
* bWaitAll is a flag and if this parameter is TRUE, the function waits until all the objects are in signaled state.
* dwMilliseconds is the timeout value in milliseconds.

In Linux, the same functionality can be achieved by using additional logic in the code. In the context of threads, POSIX semaphores are used and in the context of processes, System V semaphores are used. In Windows, if the bWaitAll flag is FALSE, the thread/process is released if any one of the synchronization objects are signaled. Linux does not provide this functionality. This logic needs to be handled in the application logic.

Examples

Following are examples for multiple objects wait functions.

**Listing 9. Windows example for any single object to be signaled**

|  |
| --- |
| HANDLE hEvents[2];  DWORD i, dwRetCode;  // Create two event objects.  for (i = 0; i < 2; i++)  {  hEvents[i] = CreateEvent(  NULL, // no security attributes  FALSE, // auto-reset event object  FALSE, // initial state is nonsignaled  NULL); // unnamed object  }  // The creating thread waits for other threads or processes  // to signal the event objects.  dwRetCode = WaitForMultipleObjects(  2, // number of objects in array  hEvents, // array of objects  FALSE, // wait for any  INFINITE); // indefinite wait  // Return value indicates which event is signaled.  switch (dwEvent)  {  // hEvent[0] was signaled.  case WAIT\_OBJECT\_0 + 0:  // Perform tasks required by this event.  break;  // hEvent[1] was signaled.  case WAIT\_OBJECT\_0 + 1:  // Perform tasks required by this event.  break;  // Return value is invalid.  default:  // probe for error  } |

**Listing 10. Linux equivalent code using POSIX**

|  |
| --- |
| **// Semaphore**  sem\_t semOne ;  sem\_t semTwo ;  sem\_t semMain ;  **// Main thread**  sem\_init(semOne,0,0) ;  sem\_init(semTwo,0,0) ;  sem\_init(semMain,0,0) ;  // create 2 threads each one waits on one semaphore  // if signaled signals the main semaphore  sem\_wait(&semMain);  **// Thread 1**  sem\_wait(&semOne);  sem\_post(&semMain);  **// Thread 2**  sem\_wait(&semTwo);  sem\_post(&semMain); |

**Listing 11. Windows example for all objects to be signaled**

|  |
| --- |
| HANDLE hEvents[2];  DWORD i, dwRetCode;  // Create two event objects.  for (i = 0; i < 2; i++)  {  hEvents[i] = CreateEvent(  NULL, // no security attributes  FALSE, // auto-reset event object  FALSE, // initial state is nonsignaled  NULL); // unnamed object  }  // The creating thread waits for other threads or processes  // to signal the event objects.  dwRetCode = WaitForMultipleObjects(  2, // number of objects in array  hEvents, // array of objects  TRUE, // wait for both the objects to be signaled  INFINITE); // indefinite wait  // Return value indicates which event is signaled.  switch (dwEvent)  {  // hEvent[0] and hEvent[1] were signaled.  case WAIT\_OBJECT\_0 :  // Perform tasks required by this event.  break;  // Return value is invalid.  default:  // probe for error  } |

**Listing 12. Linux equivalent code using POSIX**

|  |
| --- |
| **// Semaphore**  sem\_t semOne ;  sem\_t semTwo ;  sem\_t semMain ;  pthread\_mutex\_t mutMain = PTHREAD\_MUTEX\_INITIALIZER;  **// Main thread**  sem\_init(semOne,0,0) ;  sem\_init(semTwo,0,0) ;  sem\_init(semMain,0,0) ;  // create 2 threads each one waits on one semaphore  // if signaled signals the main semaphore  sem\_wait(&semMain);  **// Thread 1**  sem\_wait(&semOne);  // lock the Mutex  pthread\_mutex\_lock(&mutMain);  count ++;  if(count == 2) {  // semaphore semTwo is already signaled  // so post the main semaphore  sem\_post(&semMain);  }  pthread\_mutex\_unlock(&mutMain);  **// Thread 2**  sem\_wait(&semTwo);  // lock the Mutex  pthread\_mutex\_lock(&mutMain);  count ++;  if(count == 2) {  // semaphore semOne is already signaled  // so post the main semaphore  sem\_post(&semMain);  }  pthread\_mutex\_unlock(&mutMain); |

**Listing 13. Linux equivalent code using System V semaphores (wait until all semaphores are signaled)**

|  |
| --- |
| **// Main thread**  int key = 0x20; // Semaphore key  **// Thread 1**  struct sembuf operation[2] ;  // Create 2 semaphores  semid = semget(key, 2, 0666 | IPC\_CREAT);  operation[0].sem\_op = -1; // Wait on first resource  operation[0].sem\_num = 0;  operation[0].sem\_flg = SEM\_UNDO;  operation[0].sem\_op = -1; // Wait on the second resource  operation[0].sem\_num = 1;  operation[0].sem\_flg = SEM\_UNDO;  // Wait on both the semaphores  // Note : thread is suspended until both the semaphores are released.  semop(semid, operation, 2);  // thread is released  // delete the semaphore  semctl(semid, 0, IPC\_RMID , 0)  **// Thread 2**  struct sembuf operation[1] ;  // open semaphore  mysemid = semget(key, 2, 0);  operation[0].sem\_op = 1; // Release on the second resource  operation[0].sem\_num = 1;  operation[0].sem\_flg = SEM\_UNDO;  //Release semaphore 2  semop(semid, operation, 1);  **// Thread 3**  struct sembuf operation[1] ;  // open semaphore  mysemid = semget(key, 2, 0);  operation[0].sem\_op = 1; // Release on the first resource  operation[0].sem\_num = 0;  operation[0].sem\_flg = SEM\_UNDO;  //Release semaphore 1  semop(semid, operation, 1); |

[Back to top](http://www.ibm.com/developerworks/linux/library/l-ipc2lin3.html#ibm-pcon)

In conclusion

In this series, we've provided a guide to help map Windows processes to their functional counterparts in Linux.

In the first article we covered creating, terminating, and exiting a process; we've introduced wait functions (more about them in Part Three); and we've discussed environment variables. On the threads side, we've highlighted creation, parameter passing, specifying the function, setting the stack size, exiting, thread states, and changing priorities. And we addressed the differences in Windows and Linux of normal and time-critical threads and processes.

In the second article, we introduced synchronization objects, discussing semaphores -- creating, opening, acquiring, releasing, closing, and destroying them -- and event objects -- creating, opening, waiting on, signaling, resetting, closing, destroying, and named and un-named. In each section, we illustrated the difference between the functionality of each in Windows and in Linux.

In the last article of the series, we've defined and provided a mapping guide for mutexes, critical sections, and wait functions.

We hope this extensive mapping guide can lay the groundwork for your moving to Linux systems.

Resources

**Learn**

* Read all the articles in this series, "[Port Windows IPC apps to Linux](http://www.ibm.com/developerworks/views/linux/libraryview.jsp?search_by=port+windows+ipc+apps+linux)" (developerWorks, Spring 2005).
* The online code examples in the book [*Pthreads Programming*](http://www.oreilly.com/catalog/pthread/index.html) by Bradford Nichols, Dick Buttlar, and Jacqueline Proulx Farrel (O'Reilly, 1996) will illustrate the concepts in this article.
* Don't forget to check the [Linux Threads FAQ](http://linas.org/linux/threads-faq.html), the [Linux Manpages Online](http://man.he.net/), and the [LinuxThreads Library](http://pauillac.inria.fr/%7Exleroy/linuxthreads/) for specific calls and more details on programming with threads in Linux.
* Two previous IBM developerWorks Linux articles covered threads programming: Peter Seebach's [Basic use of pthreads](http://www.ibm.com/developerworks/linux/library/l-pthred.html) (developerWorks, January 2004) and Daniel Robbin's [POSIX threads explained](http://www.ibm.com/developerworks/linux/library/l-posix1.html) (developerWorks, July 2000).
* Refer to the [MSDN library](http://msdn.microsoft.com/) for more details on the Windows systems calls used in this article.
* A series of IBM developerWorks articles, [Migrate your apps from OS/2 to Linux](http://www.ibm.com/developerworks/linux/library/l-osmig1.html) (developerWorks, February 2004) is a good reference to see what is mapped during migration.
* Find more resources for Linux developers in the [developerWorks Linux zone](http://www.ibm.com/developerworks/linux/).

**Get products and technologies**

* Build your next development project on Linux with [IBM trial software](http://www.ibm.com/developerworks/downloads/?S_TACT=105AGX03), available for download directly from developerWorks.

**Discuss**

* Get involved in the developerWorks community by participating in [developerWorks blogs](http://www.ibm.com/developerworks/blogs/).

About the authors



Srinivasan S. Muthuswamy works as a Software Engineer for IBM Global Services Group. He joined IBM in 2000, and his expertise in programming reaches from scripting languages to object- and procedure-oriented languages on multiple platforms (Linux, Windows, WebSphere, Lotus, and so on). Muthuswamy has developed solutions ranging from system programming on Linux and Windows to Web solutions for J2EE. His primary focus is on integration and porting, and he holds a B.Eng. in Computer Engineering from the Government College of Technology, Coimbatore, India.



Kavitha Varadarajan has worked as a software Engineer in the IBM India Software Lab from December 2000. Her work experience involves development and support of host-access client products such as PCOMM and networking software like the communication server. Varadarajan has experience with a migration project that involves porting object-oriented IPC Windows applications to Linux. She holds a B.Eng. in Computer Science and Engineering from Shanmugha College of Engineering, Tanjore, India.