**Port Windows IPC apps to Linux, Part 2 - Semaphores and events**

http://www.ibm.com/developerworks/linux/library/l-ipc2lin2.html

*A mapping guide for complex, multithreaded, multiprocess applications*

[Srinivasan Muthuswamy](http://www.ibm.com/developerworks/linux/library/l-ipc2lin2.html#author1) ([smuthusw@in.ibm.com](mailto:smuthusw@in.ibm.com?subject=Semaphores%20and%20events)), Software Engineer, IBM Global Services Group



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. You can contact him at [smuthusw@in.ibm.com](mailto:smuthusw@in.ibm.com).

[Kavitha Varadarajan](http://www.ibm.com/developerworks/linux/library/l-ipc2lin2.html#author2) ([vkavitha@in.ibm.com](mailto:vkavitha@in.ibm.com?subject=Semaphores%20and%20events)), Software Engineer, IBM India Software Lab



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 such as the communication server. Varadarajan has a hands-on 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. She can be contacted at [vkavitha@in.ibm.com](mailto:vkavitha@in.ibm.com).

**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), get a mapping guide, complete with examples, to ease your transition from Windows to Linux. Part 2 examines two synchronization object types, semaphores and events.

[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-ipc2lin2.html)

[Update My dW interests](http://www.ibm.com/developerworks/linux/library/l-ipc2lin2.html) ([Log in](https://www.ibm.com/developerworks/dwwi/DWAuthRouter?m=loginpage&d=http%3A%2F%2Fwww.ibm.com%2Fdeveloperworks%2Flinux%2Flibrary%2Fl-ipc2lin2.html) | [What's this?](http://www.ibm.com/developerworks/linux/library/l-ipc2lin2.html#overlay)) [Skip to help for Update My dW interests](http://www.ibm.com/developerworks/linux/library/l-ipc2lin2.html#dwmyinterestaddhelp)

**Date:**  25 May 2005   
**Level:**  Advanced   
  
**Activity:**  15118 views   
**Comments:**   0 ([Add comments](http://www.ibm.com/developerworks/linux/library/l-ipc2lin2.html#icomments)) 

[1 star2 stars3 stars4 stars5 stars](javascript:void(0);)Average rating (based on 47 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.) are being ported to open source Linux platforms.

Many applications are designed without consideration of 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 synchronization and multiprocess apps that require interprocess synchronization.

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:

* In [Part 1](http://www-128.ibm.com/developerworks/linux/library/l-ipc2lin1.html), we dealt with processes and threads.
* This installment handles semaphores and events.
* Part 3 will cover mutexes, critical sections, and wait functions.

We'll continue our Windows-to-Linux mapping guide by starting with synchronization.

Synchronization

In Windows, synchronization is achieved by using one of the synchronization objects in one of the wait functions. The synchronization objects take either a signaled and non-signaled state. When the synchronization object is used in one of the wait functions, the wait function blocks the calling thread until the state of the synchronized object is set to signaled.

Following are some of the synchronization objects available on Windows:

* Events
* Semaphores
* Mutexes
* Critical sections

In Linux, there are different synchronization primitives available. The difference with Linux is that each primitive has its own wait functions (functions for changing the state of the synchronization primitive) -- in Windows there are common wait functions to achieve the same end. The following synchronization primitives are available in Linux:

* Semaphores
* Conditional variables
* Mutexes

Various libraries are available for Linux to provide synchronization using the previously named primitives.

Table 1. Synchronization mapping

|  |  |  |
| --- | --- | --- |
| **Windows** | **Linux -- threads** | **Linux -- process** |
| Mutex | Mutex - pthread library | System V semaphores |
| Critical section | Mutex - pthread library | Not applicable as critical sections are used only between the threads of the same process |
| Semaphore | Conditional Variable with mutex - pthreads POSIX semaphores | System V Semaphores |
| Event | Conditional Variable with mutex - pthreads | System V Semaphores |

[Back to top](http://www.ibm.com/developerworks/linux/library/l-ipc2lin2.html#ibm-pcon)

Semaphores

Windows semaphores are count variables allowing a limited number of threads/processes to access the shared resource. Linux POSIX semaphores are count variables and can be used to achieve the Windows functionality of semaphores on Linux.

The various points that needs to be considered for semaphores during the mapping process are as follows:

* **Type of semaphore:** Windows provides named and unnamed semaphores. Named semaphores extend the synchronization between processes. On Linux, POSIX semaphores are used only between the threads of same process. Between processes, System V semaphores can be used.
* **Timeout in wait functions:** When used in one of the wait functions, timeout value can be specified for Windows semaphore objects. This is not provided for in Linux -- it needs to be handled in the application logic only.

Table 2. Semaphore mapping

|  |  |  |  |
| --- | --- | --- | --- |
| **Windows** | **Linux Threads** | **Linux Process** | **Classification** |
| CreateSemaphore | sem\_init | semget semctl | Context specific |
| OpenSemaphore | Not applicable | semget | Context specific |
| WaitForSingleObject | sem\_wait sem\_trywait | semop | Context specific |
| ReleaseSemaphore | sem\_post | semop | Context specific |
| CloseHandle | sem\_destroy | semctl | Context specific |

Creating a semaphore

In Windows, CreateSemaphore() is used to create or open a named or unnamed semaphore.

|  |
| --- |
| HANDLE CreateSemaphore(  LPSECURITY\_ATTRIBUTES lpSemaphoreAttributes,  LONG lInitialCount,  LONG lMaximumCount,  LPCTSTR lpName  ); |

In this code:

* lpSemaphoreAttributes is a pointer to the security attributes. If this is null, the semaphore cannot be inherited.
* lInitialCount is the initial count of the semaphore.
* lMaximumCount is the maximum count of the semaphore and must be greater than zero.
* lpName is the name of the semaphore. If this is NULL, the semaphore is shared only between the threads of the same process. Otherwise, it can be shared between threads of different process.

This function creates the semaphore and returns the handle to the semaphore. It also sets the initial count to the values specified in the call. Thus it allows limited number of threads to access a shared resource.

In Linux, sem\_init() is used to create an unnamed POSIX semaphore which can be used between threads of the same process. This call also initializes the semaphore count: int sem\_init(sem\_t \*sem, int pshared, unsigned int value). In this code:

* value (semaphore count) is the initial value of the semaphore.
* pshared can be ignored since the POSIX semaphore is not shared between the processes in the current implementation.

It is good to notice here that maximum count value is based on the SEM\_VALUE\_MAX defined in the header file semaphore.h.

In Linux, semget() is used to create the System V semaphore which can be used between threads of different process. This is used to achieve the functionality of a Windows named semaphore. This function returns the semaphore set identifier associated with the argument key. When creating a new semaphore set, semget() initializes the semaphore-associated data structure semid\_ds as follows:

* sem\_perm.cuid and sem\_perm.uid are set to the effective user ID of the calling process.
* sem\_perm.cgid and sem\_perm.gid are set to the effective group ID of the calling process.
* The low-order nine bits of sem\_perm.mode are set to the low-order nine bits of semflg.
* sem\_nsems is set to the value of nsems.
* sem\_otime is set to 0.
* sem\_ctime is set to the current time.

This is the code used to create the System V semaphore: int semget(key\_t key, int nsems, int semflg). Following are the attributes to this code:

* The key is a unique identifier that is used by different processes to identify this semaphore set. A unique key can be generated using ftok(). IPC\_PRIVATE is a special key\_t value; when IPC\_PRIVATE is used as key the system call ignores everything but the lowest order nine bits of semflg and creates a new semaphore set (when successful).
* nsems is the number of semaphores in the semaphore set.
* semflg are permissions on the new semaphore set. To create a new set, you can set bit-wise or the access permissions with IPC\_CREAT. IPC\_CREAT/IPC\_EXCL flags will fail if the semaphore set with that key is already existing.

Notice that in System V semaphores, key is used to uniquely identify the semaphore; in Windows, the semaphore is identified by a name.

In order to initialize the semaphore-set data structure, use the semctl() system call with the IPC\_SET command. Write the values of some members of the semid\_ds structure pointed to by arg.buf to the semaphore set data structure, also updating its sem\_ctime member. The members from the user-supplied struct semid\_ds pointed to by arg.buf are the following:

* sem\_perm.uid
* sem\_perm.gid
* sem\_perm.mode (but only the lowest nine bits)

An effective user ID of the calling process should be that of super-user or at least match the creator or owner of the semaphore set: int semctl(int semid, int semnum, int cmd = IPC\_SET, ...). In this code:

* semid is the semaphore set identifier.
* semnum is the semaphore subset offset (starts at 0 till nsems -1, where n is the number of subsets in the semaphore set). This argument is ignored.
* cmd is the command; it uses IPC\_SET for setting the semaphore value.
* args are the values to be updated in the semaphore set data structure through this IPC\_SET (explained in the sample).

Maximum count value is based on the SEMVMX defined in the header file.

Opening a semaphore

In Windows, OpenSemaphore() is used to open a named semaphore. This is required only if the semaphore is shared between two processes. Upon a successful opening, this function returns the handle to the semaphore so that it can be used in the subsequent calls.

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

In this code:

* dwDesiredAccess is the requested access for the semaphore object.
* bInheritHandle is the flag which controls the inheritance of the semaphore handle. If TRUE, handle can be inherited.
* lpName is the name of the semaphore.

In Linux, the same semget() call is used to open the semaphore with 0 as the value for the semflg: int semget(key,nsems,0). In this code:

* key should point to the semaphore set key, which you want to open.
* nsems and flags can be 0 to open an already existing semaphore. The semflg value is set while creation is verified for the access permissions before returning semaphore set identifier.

Acquiring a semaphore

In Windows, wait functions provide the facility for acquiring the synchronization objects. There are different types of wait functions available -- in this section, we're only considering WaitForSingleObject() (the other types will be discussed separately). This function takes the handle to the semaphore object and waits until it is signaled or a 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, sem\_wait() is used to acquire the semaphore access. This function suspends the calling thread until the semaphore has a non-zero count. It then atomically decreases the semaphore count: int sem\_wait(sem\_t \* sem).

The timeout option is not available in POSIX semaphores. This can be achieved by issuing a non-blocking sem\_trywait() within a loop, which counts the timeout value: int sem\_trywait(sem\_t \* sem).

When a System V semaphore is used, semop() has to be used to acquire the semaphore once the initial value is set through semctl() using the IPC\_SET command. semop() performs the operations specified in the operation set and blocks the calling thread/process until the semaphore value is zero or more: int semop(int semid, struct sembuf \*sops, unsigned nsops).

The function semop() performs the set of operations contained in sops atomically -- that is, the operations are performed at the same time and only if they can all be simultaneously performed. Each of the nsops elements in the array pointed to by sops specifies an operation to be performed on a semaphore by a struct sembuf, including the following members:

* unsigned short sem\_num; (semaphore number)
* short sem\_op; (semaphore operation)
* short sem\_flg; (operation flags)

To acquire the semaphore, semop() is called with sem\_op as -1 to acquire the semaphore; after using the semaphore, semop() is called with sem\_op as 1 to release the semaphore. By calling semop() with sem\_op as -1, the semaphore count is decreased by 1 and if the value falls less than zero (since semaphore values cannot go less than zero), the semaphore count is not decreased but the calling thread/process is blocked until the semaphore is signaled.

Flags recognized in sem\_flg are IPC\_NOWAIT and SEM\_UNDO. If an operation asserts SEM\_UNDO, it will be undone when the process exits. If sem\_op is set to 0, semop() will wait for semval to become 0. This is a "wait-for-zero" operation and can be used to acquire a semaphore.

Remember that the timeout option is not available in System V semaphore. This can be achieved by issuing non-blocking semop() (by setting sem\_flg as IPC\_NOWAIT) within a loop, which counts the timeout value.

Releasing a semaphore

In Windows, ReleaseSemaphore() is used release the semaphore.

|  |
| --- |
| BOOL ReleaseSemaphore(  HANDLE hSemaphore,  LONG lReleaseCount,  LPLONG lpPreviousCount  ); |

In this code:

* hSemaphore is a pointer to the handle of the semaphore.
* lReleaseCount is the semaphore count incremented by the specified amount.
* lpPreviousCount is the pointer to the variable where the previous semaphore count is returned. The parameter can be NULL if the previous semaphore count value is not required.

This function increments the semaphore count by the value specified in lReleaseCount and sets the state of the semaphore to signaled state.

In Linux, sem\_post() is used to release semaphore. This wakes up any of the threads blocked on the semaphore. The count of the semaphore is incremented only by 1. To increment the semaphore count by a specified number (like in Windows), this function can be called multiple times along with a mutex variable: int sem\_post(sem\_t \* sem).

For System V semaphores, semop() has to be used to release the semaphore: int semop(int semid, struct sembuf \*sops, unsigned nsops).

The function semop() performs a set of operations contained in sops atomically (all operations performed at the same time only if they can be performed at the same time). Each of the nsops elements in the array pointed to by sops specifies an operation to be performed on a semaphore by a struct sembuf including the following members:

* unsigned short sem\_num; (semaphore number)
* short sem\_op; (semaphore operation)
* short sem\_flg; (operation flags)

To release the semaphore, semop() is called with sem\_op as 1. By calling semop() with sem\_op as 1, the semaphore count is incremented by 1 and semaphore is signaled.

Closing/destroying a semaphore

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

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

hObject is the pointer to the handle to the synchronization object.

In Linux, sem\_destroy() destroys the semaphore object, freeing the resources it might hold: int sem\_destroy(sem\_t \*sem). For System V semaphores, semctl() with IPC\_RMID command has to be used to close the semaphore set: int semctl(int semid, int semnum, int cmd = IPC\_RMID, ...).

This command immediately removes the semaphore set and its data structures, awakening all waiting processes (with an error return and errno set to EIDRM). The effective user ID of the calling process must be that of the super-user or match the creator or owner of the semaphore set. The argument semnum is ignored.

Examples

Following are examples for semaphores.

**Listing 1. Windows un-named semaphore code**

|  |
| --- |
| HANDLE hSemaphore;  LONG lCountMax = 10;  LONG lPrevCount;  DWORD dwRetCode;  // Create a semaphore with initial and max. counts of 10.  hSemaphore = CreateSemaphore(  NULL, // no security attributes  0, // initial count  lCountMax, // maximum count  NULL); // unnamed semaphore  // Try to enter the semaphore gate.  dwRetCode = WaitForSingleObject(  hSemaphore, // handle to semaphore  2000L); // zero-second time-out interval  switch (dwRetCode)  {  // The semaphore object was signaled.  case WAIT\_OBJECT\_0:  // Semaphore is signaled  // go ahead and continue the work  goto success:  break;  case WAIT\_TIMEOUT:  // Handle the time out case  break;  }  Success:  // Job done, release the semaphore  ReleaseSemaphore(  hSemaphore, // handle to semaphore  1, // increase count by one  NULL) // not interested in previous count  // Close the semaphore handle  CloseHandle(hSemaphore); |

**Listing 2. Linux equivalent code using POSIX semaphores**

|  |
| --- |
| **// Main thread**  #define TIMEOUT 200 /\* 2 secs \*/  **// Thread 1**  sem\_t sem ; // Global Variable  int retCode ;  // Initialize event semaphore  retCode = sem\_init(  sem, // handle to the event semaphore  0, // not shared  0); // initially set to non signaled state  while (timeout < TIMEOUT ) {  delay.tv\_sec = 0;  delay.tv\_nsec = 1000000; /\* 1 milli sec \*/  // Wait for the event be signaled  retCode = sem\_trywait(  &sem); // event semaphore handle  // non blocking call  if (!retCode) {  /\* Event is signaled \*/  break;  }  else {  /\* check whether somebody else has the mutex \*/  if (retCode == EPERM ) {  /\* sleep for delay time \*/  nanosleep(&delay, NULL);  timeout++ ;  }  else{  /\* error \*/  }  }  }  // Completed the job,  // now destroy the event semaphore  retCode = sem\_destroy(  &sem); // Event semaphore handle  **// Thread 2**  // Condition met  // now signal the event semaphore  sem\_post(  &sem); // Event semaphore Handle |

**Listing 3. Linux semaphore code using System V semaphores**

|  |
| --- |
| **// Process 1**  #define TIMEOUT 200  //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, U can also supply a value instead  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  );  //Set Initial value for the resource  semctl\_arg.val = 0; //Setting semval to 0  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 until the semaphore count becomes 0  operation[0].sem\_num = 0;  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**  key\_t key = KEY; // Process 2 should 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,0);  } |

[Back to top](http://www.ibm.com/developerworks/linux/library/l-ipc2lin2.html#ibm-pcon)

Events

In Windows, event objects are one of the synchronization objects whose state needs to be explicitly set to signaled using the SetEvent() function. Event objects come in two types:

* In the *manual reset event*, the state of the object remains signaled until explicitly reset using the ResetEvent() function.
* In the *auto reset event*, the state of the object remains signaled until a single waiting thread is released. When the waiting thread is released, the state is reset to non-signaled state.

Event objects have two states, *signaled* and *non-signaled*. The wait function on the event object blocks the calling thread until its state is set to signaled state.

The following points should be considered during migration:

* Windows provides *named* and *un-named* event objects. Named event objects are provided to provide synchronization between the processes, but in Linux, both the pthreads and POSIX provides synchronization between threads. To achieve functionality of named event objects in Linux, System V semaphore or signals can be used.
* Windows provides two types of event objects -- manual and auto reset. Linux provides only auto-reset event features.
* In Windows, event objects initial state is set to signaled. In Linux, pthreads does not provide an initial state, but POSIX semaphores provide an initial state.
* Windows event objects are asynchronous. In Linux, POSIX semaphores and System V semaphores are asynchronous but pthreads conditional variables are not asynchronous.
* When used in one of the wait functions, Windows event objects timeout value can be specified. In Linux, only pthreads provides a timeout feature in wait functions.

It is also important to note that:

* POSIX semaphores are count semaphores, but when the count is set to 1 they provide similar functionality to the Windows event object. They don't provide timeout in the wait functions. POSIX semaphores are preferred when the timeout is not the factor in the porting.
* When used along with the Mutex, pthreads conditional variables provide event-based synchronization between threads, but they are synchronous. Based on the application logic, this can be selected as a choice for implementing the functionality on Linux during porting.

Table 3. Event objects mapping

|  |  |  |  |
| --- | --- | --- | --- |
| **Windows** | **Linux Threads** | **Linux Process** | **Classification** |
| CreateEvent OpenEvent | pthread\_cond\_init sem\_init | semget semctl | context specific |
| SetEvent | pthread\_cond\_signal sem\_post | semop | context specific |
| ResetEvent | N/A | N/A | context specific |
| WaitForSingleObject | pthread\_cond\_wait pthread\_cond\_timedwait sem\_wait sem\_trywait | semop | context specific |
| CloseHandle | pthread\_cond\_destroy sem\_destroy | semctl | context specific |

Creating/opening an event object

In Windows, CreateEvent() is used to create an event object.

|  |
| --- |
| HANDLE CreateEvent(  LPSECURITY\_ATTRIBUTES lpEventAttributes,  BOOL bManualReset,  BOOL bInitialState,  LPCTSTR lpName  ) |

In this code:

* lpEventAttributes is a pointer to the attributes that determines whether the handle can be inherited or not. If this is NULL, the object handle cannot be inherited.
* bManualReset is a flag and if it is TRUE, a manual-reset event is created and ResetEvent() should be called explicitly to set the state to non-signaled.
* bInitialState is the initial state of the event object. If true, the initial state is set to signaled.
* lpName is the pointer to the name of the event object. It is kept NULL for un-named event object.

This function creates a manual-reset or auto-reset event object and also sets the initial state of the object. This function returns the handle to the event object and can be used in subsequent calls to the event object.

OpenEvent() is used to open an existing named event object. This function returns handle to the event object.

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

In this code:

* dwDesiredAccess is the requested access for the event object.
* bInheritHandle is a flag. If true, the handle can be inherited; otherwise, it cannot be inherited.
* lpName is a pointer to the name of the event object.

In Linux, the call sem\_init() creates a POSIX semaphore: int sem\_init(sem\_t \*sem, int pshared, unsigned int value) (in which value (semaphore count) is set to the initial value of the semaphore).

Linux pthreads uses pthread\_cond\_init() to create a conditional variable: int pthread\_cond\_init(pthread\_cond\_t \*cond, pthread\_condattr\_t \*cond\_attr).

Conditional variables of type pthread\_cond\_t can be initialized statically using the constant PTHREAD\_COND\_INITIALIZER. They can also be initialized using pthread\_condattr\_init() which initializes the attributes associated with the conditional variable. The call pthread\_condattr\_destroy() is used to destroy the attributes:

|  |
| --- |
| int pthread\_condattr\_init(pthread\_condattr\_t \*attr)  int pthread\_condattr\_destroy(pthread\_condattr\_t \*attr) |

Waiting on an event

In Windows, wait functions provide the facility of acquiring the synchronization objects. Different types of wait functions are available (we're only considering WaitForSingleObject() here). 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.

Linux POSIX semaphores use sem\_wait() to suspend the calling thread until the semaphore has a non-zero count. It then atomically decreases the semaphore count: int sem\_wait(sem\_t \* sem).

The timeout option is not available in the POSIX semaphore. This can be achieved by issuing non-blocking sem\_trywait() within a loop which counts the timeout value: int sem\_trywait(sem\_t \* sem).

Linux pthreads uses pthread\_cond\_wait() to block the calling thread indefinitely: int pthread\_cond\_wait(pthread\_cond\_t \*cond, pthread\_mutex\_t \*mutex). On the other hand, if the calling thread needs to be blocked for a specific time, then pthread\_cond\_timedwait() is used to block the thread. If the conditional variable is not posted within the specified time, pthread\_cond\_timedwait() returns with an error: int pthread\_cond\_timedwait(pthread\_cond\_t \*cond, pthread\_mutex\_t \*mutex,const struct timespec \*abstime). Here, the abstime parameter specifies an absolute time (specifically, the time elapsed since 00:00:00 GMT, January 1, 1970.)

Signaling an event object

The function SetEvent() is used to set the state of the event object to signaled state. Setting an already-set event object has no effect .

|  |
| --- |
| BOOL SetEvent(  HANDLE hEvent  ) |

Linux POSIX semaphores use sem\_post() to post an event semaphore. This wakes any of the threads blocked on the semaphore: int sem\_post(sem\_t \* sem).

The call pthread\_cond\_signal() is used in LinuxThreads to wake a thread waiting on the conditional variable, while pthread\_cond\_broadcast() is used to wake all the threads that are waiting on the conditional variable.

|  |
| --- |
| int pthread\_cond\_signal(pthread\_cond\_t \*cond)  int pthread\_cond\_broadcast(pthread\_cond\_t \*cond) |

Note that condition functions are not asynchronously signal-safe and should not be called from a signal handler. In particular, calling pthread\_cond\_signal() or pthread\_cond\_broadcast() from a signal handler may deadlock the calling thread.

Resetting an event

In Windows, ResetEvent() is used to reset the state of the event object to a non-signaled state.

|  |
| --- |
| BOOL ResetEvent(  HANDLE hEvent  ); |

In Linux, conditional variable and POSIX semaphores are of the auto-reset type.

Closing/destroying an event object

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

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

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

In Linux, sem\_destroy()/ pthread\_cond\_destroy() destroys semaphore objects or conditional variables, freeing the resources each might hold:

|  |
| --- |
| int sem\_destroy(sem\_t \*sem)  int pthread\_cond\_destroy(pthread\_cond\_t \*cond) |

Named event object

In Linux, the named event objects functionality between processes can be achieved by using a System V semaphore. System V semaphores are count variables, so to achieve the Windows event-object functionality, the initial count of the semaphore is set to 0 using semctl().

To signal an event, semop() is used with sem\_op value as 1. To wait on an event, semop() function is used with sem\_op value as -1 thus blocking the calling process until it is signaled.

A semaphore can be owned by setting the initial count of the semaphore to 0 using semctl(). After using the shared resource, the semaphore count can be set to 1 by using semop(). Refer to the section on semaphores in this article for the prototype for each of these System V semaphores.

Examples

Following are examples to help illustrate what we've discussed in this section.

**Listing 4. Windows un-named event object code**

|  |
| --- |
| **// Main thread**  HANDLE hEvent; // Global Variable  **// Thread 1**  DWORD dwRetCode;  // Create Event  hEvent = CreateEvent(  NULL, // no security attributes  FALSE, // Auto reset event  FALSE, // initially set to non signaled state  NULL); // un named event  // Wait for the event be signaled  dwRetCode = WaitForSingleObject(  hEvent, // Mutex handle  INFINITE); // Infinite wait  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(hEvent);  **// Thread 2**  // Condition met for the event hEvent  // now set the event  SetEvent(  hEvent); // Event Handle |

**Listing 5. Linux equivalent code using POSIX semaphores**

|  |
| --- |
| **// Main thread**  sem\_t sem ; // Global Variable  **// Thread 1**  int retCode ;  // Initialize event semaphore  retCode = sem\_init(  sem, // handle to the event semaphore  0, // not shared  0); // initially set to non signaled state  // Wait for the event be signaled  retCode = sem\_wait(  &sem); // event semaphore handle  // Indefinite wait  // Event Signaled  // a head and proceed the work  // Completed the job,  // now destroy the event semaphore  retCode = sem\_destroy(  &sem); // Event semaphore handle  **// Thread 2**  // Condition met  // now signal the event semaphore  sem\_post(  &sem); // Event semaphore Handle |

**Listing 6. Equivalent code in Linux using conditional variables**

|  |
| --- |
| **// Main thread**  pthread\_mutex\_t mutex = PTHREAD\_MUTEX\_INITIALIZER;  pthread\_cond\_t condvar = PTHREAD\_COND\_INITIALIZER;  **// Thread 1**  ...  pthread\_mutex\_lock(&mutex);  // signal one thread to wake up  pthread\_cond\_signal(&condvar);  pthread\_mutex\_unlock(&mutex);  // this signal is lost as no one is waiting  // Thread 1 now tries to take the mutex lock  // to send the signal but gets blocked  ...  pthread\_mutex\_lock(&mutex);  // Thread 1 now gets the lock and can  // signal thread 2 to wake up  pthread\_cond\_signal(&condvar);  pthread\_mutex\_unlock(&mutex);  **// Thread 2**  pthread\_mutex\_lock(&mutex);  pthread\_cond\_wait(&condvar, &mutex);  pthread\_mutex\_unlock(&mutex);  // Thread 2 blocks indefinitely  // One way of avoiding losing the signal is as follows  // In Thread 2 - Lock the mutex early to avoid losing signal  pthread\_mutex\_lock (&mutex);  // Do work  .......  // This work may lead other threads to send signal to thread 2  // Thread 2 waits for indefinitely for the signal to be posted  pthread\_cond\_wait (&condvar, &Mutex );  // Thread 2 unblocks upon receipt of signal  pthread\_mutex\_unlock (&mutex); |

**Listing 7. Windows example for named events**

|  |
| --- |
| **// Process 1**  DWORD dwRetCode;  HANDLE hEvent; // Local variable  // Create Event  hEvent = CreateEvent(  NULL, // no security attributes  FALSE, // Auto reset event  FALSE, // initially set to non signaled state  "myEvent"); // un named event  // Wait for the event be signaled  dwRetCode = WaitForSingleObject(  hEvent, // Mutex handle  INFINITE); // Infinite wait  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(hEvent);  **// Process 2**  HANDLE hEvent; // Local variable  // Open the Event  hEvent = CreateEvent(  NULL, // no security attributes  FALSE, // do not inherit handle  "myEvent"); // un named event  // Condition met for the event hEvent  // now set the event  SetEvent(  hEvent); // Event Handle  // completed the job, now close the event handle  CloseHandle(hEvent); |

**Listing 7. Windows example for named events**

|  |
| --- |
| **// Process 1**  DWORD dwRetCode;  HANDLE hEvent; // Local variable  // Create Event  hEvent = CreateEvent(  NULL, // no security attributes  FALSE, // Auto reset event  FALSE, // initially set to non signaled state  "myEvent"); // un named event  // Wait for the event be signaled  dwRetCode = WaitForSingleObject(  hEvent, // Mutex handle  INFINITE); // Infinite wait  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(hEvent);  **// Process 2**  HANDLE hEvent; // Local variable  // Open the Event  hEvent = CreateEvent(  NULL, // no security attributes  FALSE, // do not inherit handle  "myEvent"); // un named event  // Condition met for the event hEvent  // now set the event  SetEvent(  hEvent); // Event Handle  // completed the job, now close the event handle  CloseHandle(hEvent); |

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

|  |
| --- |
| **// Process 1**  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, U can also supply a value instead  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 - initially not owned  semctl\_arg.val = 0; //Setting semval to 0  semctl(semid, 0, SETVAL, semctl\_arg);  // wait on the semaphore  // blocked until it is signaled  operation[0].sem\_op = -1;  operation[0].sem\_num = 0;  operation[0].sem\_flg = IPC\_WAIT;  ret = semop(semid, operation,1);  // access the shared resource  ...  ...  //Close semaphore  iRc = semctl(semid, 1, IPC\_RMID , ignored\_argument);  }  **// Process 2**  int main()  {  key\_t key = KEY; //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);  // signal the semaphore by incrementing the semaphore count  operation[0].sem\_op = 1;  operation[0].sem\_num = 0;  operation[0].sem\_flg = SEM\_UNDO;  semop(semid, operation,0);  } |

[Back to top](http://www.ibm.com/developerworks/linux/library/l-ipc2lin2.html#ibm-pcon)

Next in the series

This second part of the series has introduced synchronization objects and primitives, starting with semaphores and events. [Part 3](http://www-128.ibm.com/developerworks/linux/library/l-ipc2lin3.html) covers mutexes, critical sections, and wait functions.

Resources

* 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.amazon.com/exec/obidos/ASIN/1565921151) by Bradford Nichols, Dick Buttlar, and Jacqueline Proulx Farrel (O'Reilly, 1996) 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.
* For more on programming with threads in Linux, see the developerWorks articles, "[Basic use of pthreads](http://www.ibm.com/developerworks/linux/library/l-pthred.html)" (developerWorks, January 2004) and "[POSIX threads explained](http://www.ibm.com/developerworks/linux/library/l-posix1.html)" (developerWorks, July 2000).
* The series of 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 involved in the developerWorks community by participating in [developerWorks blogs](http://www.ibm.com/developerworks/blogs/).
* [Browse for books](http://www.ibm.com/developerworks/apps/SendTo?bookstore=safari) on these and other technical topics.
* Innovate your next Linux development project with [IBM trial software](http://www.ibm.com/developerworks/downloads/?S_TACT=105AGX03), available for download directly from developerWorks.

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. You can contact him at [smuthusw@in.ibm.com](mailto:smuthusw@in.ibm.com).



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 such as the communication server. Varadarajan has a hands-on 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. She can be contacted at [vkavitha@in.ibm.com](mailto:vkavitha@in.ibm.com).