**Port Windows IPC apps to Linux, Part 1: Processes and threads**

http://www.ibm.com/developerworks/linux/library/l-ipc2lin1.html

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

[Srinivasan Muthuswamy](http://www.ibm.com/developerworks/linux/library/l-ipc2lin1.html#author1) ([smuthusw@in.ibm.com](mailto:smuthusw@in.ibm.com?subject=Processes%20and%20threads)), 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-ipc2lin1.html#author2) ([vkavitha@in.ibm.com](mailto:vkavitha@in.ibm.com?subject=Processes%20and%20threads)), 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 1 introduces processes and threads.

[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-ipc2lin1.html)

[Update My dW interests](http://www.ibm.com/developerworks/linux/library/l-ipc2lin1.html) ([Log in](https://www.ibm.com/developerworks/dwwi/DWAuthRouter?m=loginpage&d=http%3A%2F%2Fwww.ibm.com%2Fdeveloperworks%2Flinux%2Flibrary%2Fl-ipc2lin1.html) | [What's this?](http://www.ibm.com/developerworks/linux/library/l-ipc2lin1.html#overlay)) [Skip to help for Update My dW interests](http://www.ibm.com/developerworks/linux/library/l-ipc2lin1.html#dwmyinterestaddhelp)

**Date:**  14 Apr 2005   
**Level:**  Advanced   
  
**Activity:**  13163 views   
**Comments:**   0 ([Add comments](http://www.ibm.com/developerworks/linux/library/l-ipc2lin1.html#icomments)) 

[1 star2 stars3 stars4 stars5 stars](javascript:void(0);)Average rating (based on 54 votes)

* [+ Expand](http://www.ibm.com/developerworks/linux/library/l-ipc2lin1.html#toggle)**Show articles and other content related to my search: pthread\_join windows**

[Show descriptions](http://www.ibm.com/developerworks/linux/library/l-ipc2lin1.html#show) | [Hide descriptions](http://www.ibm.com/developerworks/linux/library/l-ipc2lin1.html#hide)

(1 - 10 of 47 search results) [View all search results](http://www.ibm.com/developerworks/search/searchResults.jsp?searchType=1&searchSite=dW&searchScope=dW&query=pthread_join%20windows)

[**Port Windows IPC apps to Linux, Part 1: Processes and threads**](http://www.ibm.com/developerworks/linux/library/l-ipc2lin1.html?S_TACT=105AGY82&S_CMP=MAVE)

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, get a mapping guide, complete with examples, to ease your transition from Windows to Linux. Part 1 introduces processes and threads.

[**Windows to UNIX porting, Part 2: Internals of porting C/C++ sources**](http://www.ibm.com/developerworks/aix/library/au-porting2/?S_TACT=105AGY82&S_CMP=MAVE)

Part 1 of this series covered the typical C/C++ project types you work with in a Microsoft(R) Visual Studio(R) environment and introduced the processes of porting dynamic and static library project variants to a UNIX(R) platform. Part 2 delves into some of the compiler options used to build Visual C++ projects and the UNIX and g++ equivalents, takes a closer look at the g++ attribute mechanism as it relates to porting, and examines some common problems you might encounter while porting from a 32-bit Windows(R) environment to a 64-bit UNIX environment. It concludes with an overview of concepts for porting multithreaded applications and an example project that shows you how to pull all these pieces together.

[**Redbooks - AIX 5L Porting Guide**](http://www.redbooks.ibm.com/abstracts/SG246034.html?Open?S_TACT=105AGY82&S_CMP=MAVE)

This IBM Redbooks publication details the types of problems most likely to be encountered when porting applications from other UNIX-based platforms to the AIX 5L Operating System. When porting an application to a new operating system there are things you have to know and questions you have to ask, such as: - What programming models are available? - How are threads implemented? - What link options do I need? - Why do my makefiles not work any more? We have tried to condense all of these questions (and answers) into one document, and this book is the result. It has been designed to provide guidance and reference material for system and application programmers who have been given the task of porting applications written in C and/or C++ to the AIX 5L operating system. This book assumes the reader is familiar with the C and/or C++ programming languages and UNIX operating systems.

[**Redbooks - Developing and Porting C and C++ Applications on AIX**](http://www.redbooks.ibm.com/abstracts/SG245674.html?Open?S_TACT=105AGY82&S_CMP=MAVE)

This IBM Redbooks publication will help experienced UNIX application developers who are new to the AIX operating system. The book explains the many concepts in detail, including the following: - Enhancements and new features provided by the latest C and C++ compilers for AIX - Compiling and linking tasks required to effectively use and manage shared libraries and run-time linking - Use of process heap and shared memory in the 32- and 64-bit user process models - A new programming paradigm in a partitioned environment where resources can be dynamically changed - Parallel programming using POSIX threads and OpenMP The following chapters are also useful for system administrators who are responsible for the software problem determination and application software release level management on AIX systems: Chapter 3, "Understanding user process models" Chapter 7, "Debugging your applications" Chapter 12, "Packaging your applications"

[**Redbooks - Scientific Applications in RS/6000 SP Environments**](http://www.redbooks.ibm.com/abstracts/SG245611.html?Open?S_TACT=105AGY82&S_CMP=MAVE)

The announcement of POWER3-based Thin and Wide nodes in early 1999, along with the addition of High nodes this fall, positions the RS/6000 SP with a new and powerful offering for the scientific and technical community. This IBM Redbooks publication provides a description of the POWER3 SMP architecture exploited by the new POWER3-based nodes and performance information for standard benchmarks, such as LINPACK, NAS 2, and Spec95. This book offers hints and tips as well as sample code for various aspects of parallel programming on POWER3 SMP architectures. Discussions of distributed memory, threads, MPI, OpenMP, LAPI, and several other facilities for developing parallel applications will help the reader understand and use these tools and take advantage of the parallel nature of the RS/6000 SP empowered by these new nodes. Due to the technical nature of the book, it will be most valuable to readers with some background in parallel computing who are familiar with SMP and parallel architectures.

[**Redbooks - Linux on IBM eServer zSeries and S/390: Application Development**](http://www.redbooks.ibm.com/abstracts/SG246807.html?Open?S_TACT=105AGY82&S_CMP=MAVE)

This IBM Redbooks publication describes application development for Linux on the IBM eServer zSeries platform. The target audience is application developers writing primarily in C/C++ and Java. The Linux development environment for zSeries is quite similar to the development environment on other platforms running Linux since the operating system services and development tools share a common code base. We note differences and optimizations specific to the zSeries platform where applicable. The zSeries platform offers unique advantages to Linux application developers. Running Linux images as guests under zVM allows consolidation of development servers onto a centrally managed machine, thus simplifying system administration of the development environment. The hardware virtualization provided by zVM allows physical resources to be shared among multiple Linux guests. In part one, we discuss standard development tools available for Linux on the zSeries platform. We provide complete details for using the IBM Java Software Development Toolkit, CVS, Emacs, the vi editor, and applications that make up the Jakarta Project. In part two, the open source Eclipse IDE is introduced. We describe the basic concepts it incorporates, and provide step-by-step instructions for installing, configuring, and working with Eclipse. In part three, we demonstrate programming techniques using an example J2EE application as an illustration. All the code necessary to implement the sample project in your own environment is included. An append

[**Tivoli Access Manager for e-business Information Center**](http://www.ibm.com/software/info/testinfo.jsp?uid=IC000043?S_TACT=105AGY82&S_CMP=MAVE)

By clicking the above link, the information center will be launched in search mode, using your search terms. Note, particular firewall settings can prevent the search terms from being passed to the information center search.

[**SoC drawer: Shared resource management**](http://www.ibm.com/developerworks/library/pa-soc5/?S_TACT=105AGY82&S_CMP=MAVE)

The goal of a system-on-a-chip (SoC) is to provide a single-chip system, and therefore SoC resource analysis and sizing is critical. Failure to properly size processing, memory, or I/O needed by software services can kill an SoC project. But all too often, SoC design analysis focuses on processing at the expense of memory or I/O sizing. And even when memory and I/O are sized properly, efficient use of these resources by software services can still be tricky. Any mis-sizing or mismanagement of memory and I/O on an SoC can at the least cause significant project delay and rework. This article examines sizing estimation and resource sharing pitfalls that the system architect should know well.

[**Migrating Win32 C/C++ applications to Linux on POWER, Part 1: Process, thread, and shared memory services**](http://www.ibm.com/developerworks/systems/library/es-MigratingWin32toLinux.html?S_TACT=105AGY82&S_CMP=MAVE)

This article covers Win32 API mapping, particularly process, thread, and shared memory services to Linux on POWER. The article can help you decide which of the mapping services best fits your needs. The author takes you through the APIs mapping he faced while porting a Win32 C/C++ application.

[**Redbooks - MQSeries Version 5 Programming Examples**](http://www.redbooks.ibm.com/abstracts/SG245214.html?Open?S_TACT=105AGY82&S_CMP=MAVE)

This IBM Redbooks publication helps you design and develop application programs that use the features of MQSeries Version 5. MQSeries Version 5 is available for five platforms: OS/2, AIX, Windows NT, HP-UX and Sun Solaris. Some of the functions are also available on the AS/400. This book outlines the new features of MQSeries Version 5. It is based on class exercises for an ITSO workshop. Several practical examples demonstrate how to: - Segment large messages of up to 100MB - Group messages for better performance - Send message to multiple destinations using a distribution list - Coordinate queuing functions and database updates using a two-phase commit - Perform remote MQSeries administration in an Windows NT workgroup - Transfer files to another system using reference messages - Improve performance using fastpath bindings - Write multi-threaded programs The first chapter contains an overview of the functions released with MQSeries Version 5. The other chapters are dedicated to specific functions. They include programming hints and examples. This book comes with a diskette that contains the source code of all examples.

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 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, think of this series as a mapping document -- it provides mapping of various Windows calls to Linux calls related to threads, processes, and interprocess communication elements (mutexes, semaphores, etc.). We've divided the mapping into three chunks:

* Part 1 deals with processes and threads.
* Part 2 handles semaphores and events.
* Part 3 covers mutexes, critical sections, and wait functions.

Processes

Basic execution units in Windows and Linux are different. In Windows, the thread is the basic execution unit, and the process is a container that holds this thread.

In Linux, the basic execution unit is the process. The functionalities offered by Windows APIs can be mapped directly to Linux system calls:

Table 1. Process mapping

|  |  |  |
| --- | --- | --- |
| **Windows** | **Linux** | **Classification** |
| CreateProcess() CreateProcessAsUser() | fork() setuid() exec() | Mappable |
| TerminateProcess() | kill() | Mappable |
| SetThreadpriority() GetThreadPriority() | Setpriority() getPriority() | Mappable |
| GetCurrentProcessID() | getpid() | Mappable |
| Exitprocess() | exit() | Mappable |
| Waitforsingleobject() Waitformultipleobject() GetExitCodeProcess() | waitpid() Using Sys V semaphores, Waitforsingleobject/multipleobject can be implemented | Context specific |
| GetEnvironmentVariable SetEnvironmentVariable | getenv() setenv() | Mappable |

The Classification column (which explains classification constructs used in this article) indicates whether the Windows construct is *mappable* or *context specific*:

* If mappable, the Windows construct can be mapped to the specified Linux construct(s) by closely examining the types, parameters, return codes, and such. Both the Windows and Linux constructs provide similar functionality.
* If context specific, the given Windows construct may or may not have an equivalent construct in Linux, or Linux may have more than one construct that provides similar functionality. In either case, the decision to use a specific Linux construct(s) depends on the application context.

Creating a process

In Windows, you can use CreateProcess() to create a new process. The CreateProcess() function creates a new process and its main thread as follows:

|  |
| --- |
| BOOL CreateProcess(  LPCTSTR lpApplicationName, // name of executable module  LPTSTR lpCommandLine, // command line string  LPSECURITY\_ATTRIBUTES lpProcessAttributes, // SD  LPSECURITY\_ATTRIBUTES lpThreadAttributes, // SD  BOOL bInheritHandles, // handle inheritance option  DWORD dwCreationFlags, // creation flags  LPVOID lpEnvironment, // new environment block  LPCTSTR lpCurrentDirectory, // current directory name  LPSTARTUPINFO lpStartupInfo, // startup information  LPPROCESS\_INFORMATION lpProcessInformation // process information  ) |

bInheritHandles determines whether the handles have to be inherited to the child from the parent. lpApplicationName and lpCommandLine give the name and path of the process to be started. lpEnvironment defines the environment that has to be visible for the process.

In Linux, the exec\* family of functions replace the current process image with a new process image (as shown in the following):

|  |
| --- |
| int execl(const char \*path, const char \*arg, ...);  int execlp(const char \*file, const char \*arg, ...);  int execle(const char \*path, const char \*arg , ..., char \* const envp[]);  int execv(const char \*path, char \*const argv[]);  int execvp(const char \*file, char \*const argv[]); |

These versions of exec\* are just various calling interfaces for core function execve(): int execve(const char \*filename, char \*const argv [], char \*const envp[]). Here argv is the pointer containing arguments list and envp is the pointer containing list of environment variables which are basically key=value pairs.

This must be used along with the fork() command so both the parent and child processes are running: pid\_t fork(void). fork() creates a child process that differs from the parent process only in its PID and PPID; in fact, the resource utilizations are set to 0.

By default, the exec() function inherits the group and user IDs from the parent process, which makes it dependent on the parent process. This can be changed by:

* Setting the set-uid and set-gid bit on the program file pointed
* Using the setpgid() and setuid() system call

The CreateProcessAsUser() function is similar to CreateProcess() except that the new process runs in the security context of the user represented by the hToken parameter. There is no one-to-one equivalent for this function in Linux, but it can be replicated using the following logic:

* fork() to create a new child process with new PID
* setuid() to switch to the new PID
* exec() to change the existing process image with the process to execute

Terminating a process

To forcibly terminate a running process, you can use TerminateProcess() in Windows.

|  |
| --- |
| BOOL TerminateProcess(  HANDLE hProcess, // handle to the process  UINT uExitCode // exit code for the process  ); |

This function terminates the running process and all the associated threads. Use this function only in extreme scenarios.

In Linux, you can use kill() to forcibly kill a process: int kill(pid\_t pid, int sig). This system call terminates the process of id PID. You can also use it to signal to any group or process.

Using wait functions

In cases when the child process is dependent on the parent process, you can use wait functions in parent process to wait for the child process termination. In Windows, you can use the WaitForSingleObject() function call to achieve this.

You can use the WaitForMultipleObject() function to wait for more than one object.

|  |
| --- |
| DWORD WaitForMultipleObjects(  DWORD nCount, // number of handles in array  CONST HANDLE \*lpHandles, // object-handle array  BOOL bWaitAll, // wait option  DWORD dwMilliseconds // time-out interval  ); |

You can populate the object-handle array with the number of objects to wait for. Based on the bWaitALL option, you can either wait for all the objects to be signaled or wait for any of them to be signaled.

In both of these functions, if you want to wait for a finite time, you can specify the time interval in the second parameter. If you want to wait infinitely, use INFINITE as the value for dwMilliseconds. Setting dwMilliseconds to 0 will just test the state of the object and return.

You can use waitpid() in Linux if you want to just wait infinitely for the process to die. In Linux, there is no way to do a timed wait on a waitpid() call.

In this code: pid\_t waitpid(pid\_t pid, int \*status, int options), waitpid() infinitely waits for the child process to terminate. Wait functions, in both Windows and Linux, suspend the execution of the current process until it completes, but in Windows there is an option to exit by specifying a time value. You can implement a timed wait or NO WAIT functionality similar to WaitForSingleObject() and WaitForMultipleObject() using System V semaphores, which is discussed in Part 2 of this series. Part 3 of this series further discusses wait functions.

Exiting a process

Exiting a process means a graceful exiting of the process with a proper cleanup. In Windows, you use ExitProcess() to perform this operation.

|  |
| --- |
| VOID ExitProcess(  UINT uExitCode // exit code for all threads  ); |

ExitProcess() is the preferred method of ending a process. This function provides a clean process shutdown. This includes calling the entry-point function of all attached dynamic-link libraries (DLLs) with a value indicating that the process is detaching from the DLL.

The Linux equivalent for ExitProcess() is exit(): void exit(int status);.

The exit() function causes normal program termination and the value of status &0377 is returned to the parent. The C standard specifies two definitions (EXIT\_SUCCESS and EXIT\_FAILURE) that can be passed to the status parameter to indicate successful or unsuccessful termination.

Environment variables

Each process has an environment block associated with it, basically name=value pairs that specify various environments the process can access. Even though we can specify the environment when we create the process, there are also specific functions to set and obtain environment variables after the process is created.

In Windows, you can use GetEnvironmentVariable() and SetEnvironmentVariable() to get and set the environment variables.

|  |
| --- |
| DWORD GetEnvironmentVariable(  LPCTSTR lpName, // environment variable name  LPTSTR lpBuffer, // buffer for variable value  DWORD nSize // size of buffer  ); |

This function returns the size of the value buffer on success and 0 if the name specified is not a valid environment variable name. The SetEnvironmentVariable() function sets the contents of the specified environment variable for the current process.

|  |
| --- |
| BOOL SetEnvironmentVariable(  LPCTSTR lpName, // environment variable name  LPCTSTR lpValue // new value for variable  ); |

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero.

In Linux, getenv() and setenv() system calls provide the equivalent functionality.

|  |
| --- |
| char \*getenv(const char \*name);  int setenv(const char \*name, const char \*value, int overwrite); |

The getenv() function searches the environment list for a string that matches the string pointed to by name. This function returns a pointer to the value in the environment or NULL if there is no match. The setenv() function adds the variable name to the environment with the value if the name does not already exist. If the name does exist in the environment, then its value is changed to value if overwrite is non-zero. If overwrite is zero, then the value of name is not changed. The setenv() function returns zero on success or -1 if there was insufficient space in the environment.

Examples

The following examples illustrate what we've discussed in this section.

**Listing 1. Windows process code**

|  |
| --- |
| //Sample Application that explain process concepts  //Parameters Declaration/Definition  int TimetoWait;  STARTUPINFO si;  PROCESS\_INFORMATION pi;  LPTSTR lpszCurrValue,LPTSTR lpszVariable;  TCHAR tchBuf[BUFSIZE];  BOOL fSuccess;  if(argc > 2)  {  printf("InvalidArgument");  ExitProcess(1); //Failure  }  //Get and display an environment variable PATH  lpszCurrValue = ((GetEnvironmentVariable("PATH",tchBuf, BUFSIZE) > 0) ? tchBuf : NULL);  lpszVariable = lpszCurrValue;  //Display the environment variable  while (\*lpszVariable)  putchar(\*lpszVariable++);  putchar('\n');  //Initialise si and pi  ZeroMemory( &si, sizeof(si) );  si.cb = sizeof(si);  ZeroMemory( &pi, sizeof(pi) );  //Create a childProcess  if( !CreateProcess( NULL, // No module name (use command line).  "SomeProcess", // Command line.  NULL, // Process handle not inheritable.  NULL, // Thread handle not inheritable.  FALSE, // Set handle inheritance to FALSE.  0, // No creation flags.  NULL, // Use parent's environment block.  NULL, // Use parent's starting directory.  &si, // Pointer to STARTUPINFO structure.  &pi ) // Pointer to PROCESS\_INFORMATION structure.  )  {  printf( "CreateProcess failed." );  }  // Wait until child process exits.  if(argc == 2)  {  TIMEOUT = atoi(argv[1]);  ret = WaitForSingleObject( pi.hProcess, TIMEOUT );  if(ret == WAIT\_TIMEOUT)  {  TerminateProcess(pi.hProcess);  }  }  else  {  WaitForSingleObject( pi.hProcess, INFINITE );  ...  }  ExitProcess(0); //Success |

**Listing 2. Equivalent Linux process code**

|  |
| --- |
| #include <stdlib.h>  int main(int argc,char \*argv[])  {  //Parameters Declaration/Definition  char PathName[255];  char \*Argptr[20];  int rc;  char \*EnvValue,\*lpszVariable;  if(argc > 1)  {  printf(" Wrong parameters !!");  exit(EXIT\_FAILURE);  }  //Get and display an environment variable PATH  EnvValue = getenv("PATH");  if(EnvValue == NULL)  {  printf("Invalid environment variable passed as param !!");  }else  {  lpszVariable = EnvValue;  while (\*lpszVariable)  putchar(\*lpszVariable++);  putchar('\n');  }  rc = fork(); //variable rc's value on success would be process ID in the parent  //process, and 0 in the child's thread of execution.  switch(rc)  {  case -1:  printf("Fork() function failed !!");  ret = -1;  break;  case 0:  printf("Child process...");  setpgid(0,0); //Change the parent grp ID to 0  ret = execv(PathName,Argptr); // there are other flavours of exec available,  // u can use any of them based on the arguments.  if(ret == -1)  {  kill(getpid(),0);  }  break;  default:  // infinitely waits for child process to die  Waitpid(rc,&status,WNOHANG);  //Note RC will have PID returned since this is parent process.  break;  }  exit(EXIT\_SUCCESS);  } |

[Back to top](http://www.ibm.com/developerworks/linux/library/l-ipc2lin1.html#ibm-pcon)

Threads

In Windows, the thread is the basic unit of execution. One or more threads run in the context of the process. The scheduling code is implemented in the kernel. There is no single "scheduler" module or routine.

The Linux kernel uses a process model rather than a threading model. The Linux kernel provides a lightweight process framework for creating threads; the actual thread implementation is in the user space. There are various threading libraries available (LinuxThreads, NGPT, NPTL, and so on) in Linux. The information in this article is based on the LinuxThreads library, but the information here is also applicable to Red Hat's Native POSIX Threading Library (NPTL).

This section describes threading in Windows and in Linux. It covers the calls for creating a thread, setting its attributes, and changing its priority.

Table 2. Thread mapping

|  |  |  |
| --- | --- | --- |
| **Windows** | **Linux** | **Classification** |
| CreateThread | pthread\_create pthread\_attr\_init pthread\_attr\_setstacksize pthread\_attr\_destroy | Mappable |
| ThreadExit | pthread\_exit | Mappable |
| WaitForSingleObject | pthread\_join pthread\_attr\_setdetachstate pthread\_detach | Mappable |
| SetPriorityClass SetThreadPriority | setpriority sched\_setscheduler sched\_setparam  pthread\_setschedparam pthread\_setschedpolicy pthread\_attr\_setschedparam pthread\_attr\_setschedpolicy | Context Specific |

Creating a thread

In Windows, you can use CreateThread() to create a thread to execute under the virtual address space of the calling process.

|  |
| --- |
| HANDLE CreateThread(  LPSECURITY\_ATTRIBUTES lpThreadAttributes, // SD  SIZE\_T dwStackSize, // initial stack size  LPTHREAD\_START\_ROUTINE lpStartAddress, // thread function  LPVOID lpParameter, // thread argument  DWORD dwCreationFlags, // creation option  LPDWORD lpThreadId // thread identifier  ); |

lpThreadAttributes is a pointer to the thread attributes that determines whether the thread handle can be inherited by the child process.

Linux uses the pthread library call pthread\_create() to spawn a thread:

|  |
| --- |
| int pthread\_create (pthread\_t \*thread\_id, pthread\_attr\_t \*threadAttr,  void \* (\*start\_address)(void \*), void \* arg); |

**Note:** In Windows, the number of threads a process can create is limited by the available virtual memory. By default, every thread has one megabyte of stack space. Therefore, you can create at most 2,028 threads. If you reduce the default stack size, you can create more threads. In Linux, the maximum number of process per user can be found using ULIMIT -a (limits for all users), and you can update it by using ULIMIT -u, but it would be valid only for that logon. The header files under /usr/Include/limit.h and ulimit.h define these constants. You can modify them and recompile kernel to hv permanent effect. For POSIX threadlimits, the THREAD\_THREADS\_MAX macro defines the maximum limit and is defined in local\_lim.h.

Specifying the thread function

The parameter lpStartAddress in the CreateThread() is the address of the function that the newly created thread will execute.

The parameter start\_address for the Linux library call pthread\_create() is the address of the function that the newly created thread will execute.

Parameter passing to the thread function

In Windows, the parameter lpParameter for the system call CreateThread() specifies the parameter to be passed to the newly created thread. It specifies the address of the data item to be passed to the new thread.

In Linux, the parameter arg for the library call pthread\_create() specifies the parameter to be passed to the new thread.

Setting the stack size

In Windows, the parameter dwStackSize for the CreateThread() is the size of stack in bytes that is to be allocated for the new thread. The stack size should be a non-zero multiple of 4 KB and a minimum of 8 KB.

In Linux, the stack size is set in the pthread attributes object; that is, the parameter threadAttr of type pthread\_attr\_t is passed to the library call pthread\_create(). This object needs to be initialized by the call pthread\_attr\_init() before any attributes are set. The attribute object is destroyed using the call pthread\_attr\_destroy():

|  |
| --- |
| int pthread\_attr\_init(pthread\_attr\_t \*threadAttr);  int pthread\_attr\_destroy(pthread\_attr\_t \*threadAttr); |

Note that all of the pthread\_attr\_setxxxx calls achieve similar functionality to the pthread\_xxxx calls (if available) except that you can use pthread\_attr\_xxxx only before thread creation to update the attribute object that will be passed as a parameter to pthread\_create. Meanwhile, you can use pthread\_xxxx calls at any time after the thread has been created.

The stack size is set using the call pthread\_attr\_setstacksize(): int pthread\_attr\_setstacksize(pthread\_attr\_t \*threadAttr, int stack\_size);.

Exiting a thread

In Windows, the system call ExitThread() terminates the thread. The dwExitCode is the return value of the thread, and it can be retrieved from another thread by calling GetExitCodeThread().

|  |
| --- |
| VOID ExitThread(  DWORD dwExitCode // exit code for this thread  ); |

The Linux equivalent for this is the library call pthread\_exit(). The retval is the return value of the thread, and you can retrieve it from another thread by calling pthread\_join(): int pthread\_exit(void\* retval);.

Thread states

In Windows, there are no explicit thread states maintained with respect to thread termination. However, WaitForSingleObject() allows a thread to wait explicitly on the termination of a specific or non-specific thread within the process.

In Linux, threads are by default created in joinable state. In joinable state, another thread can synchronize on the thread's termination and recover its termination code using the function pthread\_join(). The thread resources of the joinable thread are released only after it is joined.

Windows uses WaitForSingleObject() to wait for a thread to terminate:

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

Where:

* hHandle is the pointer to the thread handle.
* dwMilliseconds is the time out value in milliseconds. If the value is set to INFINITE, then it blocks the calling thread/process indefinitely.

Linux uses pthread\_join() to do the same: int pthread\_join(pthread\_t \*thread, void \*\*thread\_return);.

In the detached state, the thread resources are immediately freed when it terminates. The detached state can be set by calling pthread\_attr\_setdetachstate() on the thread attribute object: int pthread\_attr\_setdetachstate (pthread\_attr\_t \*attr, int detachstate);. A thread created in a joinable state can later be put into a detached state using the pthread\_detach() call: int pthread\_detach (pthread\_t id);.

Changing priority

In Windows, the priority of the thread is determined by the priority class of its process and the priority level of the thread within the priority class of the process. In Linux, the thread itself is the unit of execution and has its own priority. It has no dependency on the priority of its process.

In Windows, you can use SetPriorityClass() to set the priority class for the specified process:

|  |
| --- |
| BOOL SetPriorityClass(  HANDLE hProcess, // handle to the process  DWORD dwPriorityClass // Priority class  ); |

dwPriorityClass is the priority class of the process, and it is set to any of the following values:

* IDLE\_PRIORITY\_CLASS
* BELOW\_NORMAL\_PRIORITY\_CLASS
* NORMAL\_PRIORITY\_CLASS
* ABOVE\_NORMAL\_PRIORITY\_CLASS
* HIGH\_PRIORITY\_CLASS
* REALTIME\_PRIORITY\_CLASS

Once the priority class of the process is set, SetThreadPriority() is used to set the priority level of the thread within the priority class of the process:

|  |
| --- |
| BOOL SetThreadPriority(  HANDLE hThread,  int nPriority  ); |

nPriority is the priority value of the thread, and it is set to one of the following values:

* THREAD\_PRIORITY\_ABOVE\_NORMAL sets the priority to 1 point above the priority class.
* THREAD\_PRIORITY\_BELOW\_NORMAL sets the priority to 1 point below the priority class.
* THREAD\_PRIORITY\_HIGHEST sets the priority to 2 points above the priority class.
* THREAD\_PRIORITY\_IDLE sets base priority to 1 for IDLE\_PRIORITY\_CLASS, BELOW\_NORMAL\_PRIORITY\_CLASS, NORMAL\_PRIORITY\_CLASS, ABOVE\_NORMAL\_PRIORITY\_CLASS, or HIGH\_PRIORITY\_CLASS processes, and sets base priority to 16 for REALTIME\_PRIORITY\_CLASS processes.
* THREAD\_PRIORITY\_LOWEST sets the priority to 2 points below the priority class.
* THREAD\_PRIORITY\_NORMAL sets to normal priority for the priority class.
* THREAD\_PRIORITY\_TIME\_CRITICAL sets the base priority to 15 for IDLE\_PRIORITY\_CLASS, BELOW\_NORMAL\_PRIORITY\_CLASS, NORMAL\_PRIORITY\_CLASS, ABOVE\_NORMAL\_PRIORITY\_CLASS, or HIGH\_PRIORITY\_CLASS processes, and sets base priority to 31 for REALTIME\_PRIORITY\_CLASS.

[Back to top](http://www.ibm.com/developerworks/linux/library/l-ipc2lin1.html#ibm-pcon)

Examples of processes and threads

To wrap up this installment, let's look at some examples of the following types of processes and threads:

* Normal or regular processes and threads
* Time-critical and real-time processes and threads

Normal or regular processes/threads

The Linux system call setpriority() is used to set or modify priority levels for normal processes and threads. The parameter scope is PRIO\_PROCESS. Set id to 0 to change the current process (or thread) priority. Again, delta is the priority value -- this time in the range -20 to 20. Note also that in Linux, a lower delta value means a higher priority. So you set +20 for IDLETIME priority and 0 for REGULAR priority.

In Windows, the priority range is from 1 (lower priority) to 15 (higher priority) for the regular threads. But in Linux, the priority range for normal non-real-time processes is from -20 (higher) to +20 (lower priority). This has to be mapped before being used: int setpriority(int scope, int id, int delta);.

Time-critical and real-time processes and threads

You can use the Linux system call sched\_setscheduler() to change the scheduling priority and policy of a running process: int sched\_setscheduler(pit\_t pid, int policy, const struct sched\_param \*param);.

The parameter policy is the scheduling policy. The possible values for policy are SCHED\_OTHER (for regular non-real-time scheduling), SCHED\_RR (real-time round-robin policy), and SCHED\_FIFO (real-time FIFO policy).

Here, param is a pointer to a structure representing scheduling priority. It can range from 1 to 99 only for real-time policies. For others (normal non-real-time processes), it is zero.

In Linux, for a known scheduling policy, it is also possible to change only the process priority by using the system call sched\_setparam: int sched\_setparam(pit\_t pid, const struct sched\_param \*param);.

The LinuxThreads library call pthread\_setschedparam is the thread version of sched\_setscheduler and is used to dynamically change the scheduling priority and policy for a running thread: int pthread\_setschedparam(pthread\_t target\_thread, int policy, const struct sched\_param \*param);.

The parameter target\_thread indicates the thread whose priority is to be changed; param indicates the priority.

The LinuxThreads library calls pthread\_attr\_setschedpolicy, and you can use pthread\_attr\_setschedparam to set the scheduling policy and the priority level to the thread attribute object before the thread is created:

|  |
| --- |
| int pthread\_attr\_setschedpolicy(pthread attr\_t \*threadAttr, int policy);  int pthread\_attr\_setschedparam(pthread attr\_t \*threadAttr, const struct sched\_param \*param); |

In Windows, the priority range is from 16 (lower priority) to 31 (higher priority) for the real-time threads. In Linux, the priority range for real-time threads is from 99 (higher) to 1 (lower priority). This has to be mapped before being used.

Examples

The following listings illustrate the concepts in this section.

**Listing 3. Windows thread example**

|  |
| --- |
| **Main Thread**  enum stackSize = 120 \* 1024 ;  // create a thread normal and real time thread  DWORD normalTId, realTID;  HANDLE normalTHandle, realTHandle;  normalTHandle = CreateThread(  NULL, // default security attributes  stackSize, // 120K  NormalThread, // thread function  NULL, // argument to thread function  0, // use default creation flags  &normalTId); // returns the thread identifier  // Set the priority class as "High priority"  SetPriorityClass(pHandle, HIGH\_PRIORITY\_CLASS);  normalTHandle = CreateThread(  NULL, // default security attributes  stackSize, // 120K  NormalThread, // thread function  NULL, // argument to thread function  0, // use default creation flags  &normalTId); // returns the thread identifier  CloseHandle(threadHandle);  ...  ...  // Thread function  DWORD WINAPI NormalThread ( LPVOID lpParam )  {  HANDLE tHandle,pHandle;  pHandle = GetCurrentProcess();  tHandle = GetCurrentThread();  // Set the priority class as "High priority"  SetPriorityClass(pHandle, HIGH\_PRIORITY\_CLASS);  // increase the priority by 2 points above the priority class  SetThreadPriority(tHandle,THREAD\_PRIORITY\_HIGHEST);  // perform job at high priority  ...  ...  ...  // Reset the priority class as "Normal"  SetPriorityClass(pHandle, NORMAL\_PRIORITY\_CLASS);  // set the priority back to normal  SetThreadPriority(tHandle,THREAD\_PRIORITY\_NORMAL);  // Exit thread  ExitThread(0);  }  // Thread function  DWORD WINAPI RealTimeThread ( LPVOID lpParam )  {  HANDLE tHandle, pHandle ;  pHandle = GetCurrentProcess();  tHandle = GetCurrentThread ();  // Set the priority class as "Real time"  SetPriorityClass(pHandle, REALTIME\_PRIORITY\_CLASS);  // increase the priority by 2 points above the priority class  SetThreadPriority(tHandle,THREAD\_PRIORITY\_HIGHEST);  // do time critical work  ...  ...  ...  // Reset the priority class as "Normal"  SetPriorityClass(pHandle, NORMAL\_PRIORITY\_CLASS);  // Reset the priority back to normal  SetThreadPriority(tHandle,THREAD\_PRIORITY\_NORMAL);  ExitThread(0);  } |

**Listing 4. Equivalent thread code in Linux**

|  |
| --- |
| static void \* RegularThread (void \*);  static void \* CriticalThread (void \*);  **// Main Thread**  pthread\_t thread1, thread2; // thread identifiers  pthread\_attr\_t threadAttr;  struct sched\_param param; // scheduling priority  // initialize the thread attribute  pthread\_attr\_init(&threadAttr);  // Set the stack size of the thread  pthread\_attr\_setstacksize(&threadAttr, 120\*1024);  // Set thread to detached state. No need for pthread\_join  pthread\_attr\_setdetachstate(&threadAttr, PTHREAD\_CREATE\_DETACHED);  // Create the threads  pthread\_create(&thread1, &threadAttr, RegularThread, NULL);  pthread\_create(&thread2, &threadAttr, CriticalThread,NULL);  // Destroy the thread attributes  pthread\_attr\_destroy(&threadAttr);  ...  ...  // Regular non-realtime Thread function  static void \* RegularThread (void \*d) {  int priority = -18;  // Increase the priority  setpriority(PRIO\_PROCESS, 0, priority);  // perform high priority job  ...  ...  // set the priority back to normal  setpriority(PRIO\_PROCESS, 0, 0);  pthread\_exit(NULL);  }  // Time Critical Realtime Thread function  static void \* CriticalThread (void \*d) {  // Increase the priority  struct sched\_param param; // scheduling priority  int policy = SCHED\_RR; // scheduling policy  // Get the current thread id  pthread\_t thread\_id = pthread\_self();  // To set the scheduling priority of the thread  param.sched\_priority = 90;  pthread\_setschedparam(thread\_id, policy, &param);  // Perform time critical task  ...  ...  // set the priority back to normal  param.sched\_priority = 0;  policy = 0; // for normal threads  pthread\_setschedparam(thread\_id, policy, &param);  ....  ....  pthread\_exit(NULL);  } |

[Back to top](http://www.ibm.com/developerworks/linux/library/l-ipc2lin1.html#ibm-pcon)

Next in the series

This first part of the series has given you a guide to help map Windows processes and threads to their functional counterparts in Linux. [Part 2](http://www-128.ibm.com/developerworks/linux/library/l-ipc2lin2.html) in the series covers 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).