UNIX Code Migration Guide

UNIX Application Migration Guide

patterns & practices
proven practices for predictable results

Chapter 9: Win32 Code Conversion

Larry Twork, Larry Mead, Bill Howison, JD Hicks, Lew Brodnax, Jim McMicking, Raju Sakthivel, David Holder, Jon Collins, Bill Loeffler
Microsoft Corporation

October 2002

Applies to:
Microsoft® Windows®
UNIX applications

Summary: Chapter 9: Win32 Code Conversion covers the fundamentals of converting code from UNIX to Windows. Functional areas that surface when migrating to Win32 such as processes, threads, signal handling, memory management, networking and a host of related subjects are covered. In general, the material is presented first with background on the migration issue followed by code samples to illustrate before and after migration. (212 printed pages)

Contents

Introduction

This chapter describes how you can modify the source code for your UNIX application so that it will compile on the Microsoft® Windows® operating system. You need to modify your code due to the differences between the UNIX and Windows application and coding environments described earlier in this guide.

The potential coding differences that need to be addressed are described in the following categories:

- Processes
- Signals and signal handling
- Threads
- Memory management
- Users, groups, and security
- File and data access
- Interprocess communication
- Sockets and networking
- Process environment
- Multiprocessor considerations
- Daemons and services

For each of these categories, this chapter:

- Describes the coding differences.
- Outlines options for converting the code.
- Illustrates with source code examples.

You can then choose the solution appropriate to your application and use these examples as a basis for constructing your Windows code.

This guide gives you sufficient information so that you can choose the best method of converting the code. Once you have made your choice, you can refer to the standard documentation to ensure that you understand the details of the Microsoft Win32® application programming interface functions and application program interfaces (APIs). Throughout this chapter, there are references for further information on the recommended coding changes. In particular, references to the detail of the function calls and libraries are given.

Processes

The UNIX and Windows process models are very different, and the major difference lies in the creation of processes. UNIX uses **fork** to create a new copy of a running process and **exec** to replace a process's executable file with a new one. Windows does not have a fork function. Instead, Windows creates processes in one step by using **CreateProcess**. While there is no need in Win32 to execute the process after its creation (as it will already by executing the new code), the standard **exec** functions are still available in Win32.

These differences (and others) result in the need to convert the UNIX code before it can run on a Win32 platform.

The areas that you need to consider that are covered in this section are:

- Process creation
- Replacing a process's executable code
- Spawning child processes and the process hierarchy
- Waiting for a child process
- Setting process resource limits

The concept of Windows Jobs is also introduced, which allows you to group processes together for management purposes. This functionality is not available in UNIX.

> **Note:** There are a number of process management functions in the Win32 API. For details of these functions, consult the Win32 API reference.

Creating a New Process

In UNIX, a developer creates a new process by using **fork**. The **fork** function creates a child process that is an almost exact copy of the parent process. The fact that the child is a copy of the parent ensures that the process environment is the same for the child as it is for the parent.

In Windows, the **CreateProcess** function enables the parent process to create an operating environment for a new process. The environment includes the working directory, window attributes, environment variables, execution priority and command line arguments. A handle is returned by the **CreateProcess** function, which enables the parent application to perform operations on the process and its environment while it is executing. Unlike UNIX, the executable file run by **CreateProcess** is not a copy of the parent process, and it has to be explicitly specified in the call to **CreateProcess**.

An alternative to **CreateProcess** is to use one of the **spawn** functions that is present in the standard C runtime. There are 16 variations of the **spawn** function. Each **spawn** function creates and executes a new process. Many of these have the same functionality as the similarly named **exec** functions. The **spawn** functions include an additional argument that permits the new process to replace the current process, suspend the current process until the spawned process terminates, run asynchronously with the calling process or run simultaneously and detach as a background process.

For a UNIX application to change the executable file run in the child process, the child process must explicitly call an **exec** function to overwrite the executable file with a new application. The combination of **fork** and **exec** is similar to, but not the same as, **CreateProcess**.

The example below shows a UNIX application that forks to create a child process and then runs the UNIX **ps** command by using **execlp**.

Creating a process in UNIX using fork and exec

```
#include <unistd.h>
#include <stdio.h>
#include <sys/types.h>

int main()
```

```
{
    pid_t pid;

    printf("Running ps with fork and execlp\n");
    pid = fork();
    switch(pid)
    {
    case -1:
        perror("fork failed");
        exit(1);
    case 0:
        if (execlp("ps", NULL) < 0) {
            perror("execlp failed");
            exit(1);
        }
        break;
    default:
        break;
    }
    printf("Done.\n");
    exit(0);
}
```

You can port this code to Windows using the Win32 **CreateProcess** function discussed earlier, or by using a **spawn** function from the standard C runtime library. In both cases, the old and new processes run parallel, asynchronously.

Creating a process in Windows using CreateProcess

```
#include <windows.h>
#include <process.h>
#include <stdio.h>

void main()
{
    STARTUPINFO    si;
    PROCESS_INFORMATION  pi;

    GetStartupInfo(&si);

    printf("Running Notepad with CreateProcess\n");
    CreateProcess(NULL, "notepad",  // Name of app to launch
  NULL,      // Default process security attributes
  NULL,      // Default thread security attributes
  FALSE,      // Don't inherit handles from the parent
  0,      // Normal priority
  NULL,      // Use the same environment as the parent
  NULL,      // Launch in the current directory
  &si,      // Startup Information
  &pi);      // Process information stored upon return

    printf("Done.\n");
    exit(0);
}
```

The arguments supported by **CreateProcess** (shown in the preceding example) give you a considerable degree of control over the newly created process. This contrasts with the **spawn** functions, which do not provide options to set process priority, security attributes, or the debug status.

The next example shows a port of the same code using the **_spawnlp** function.

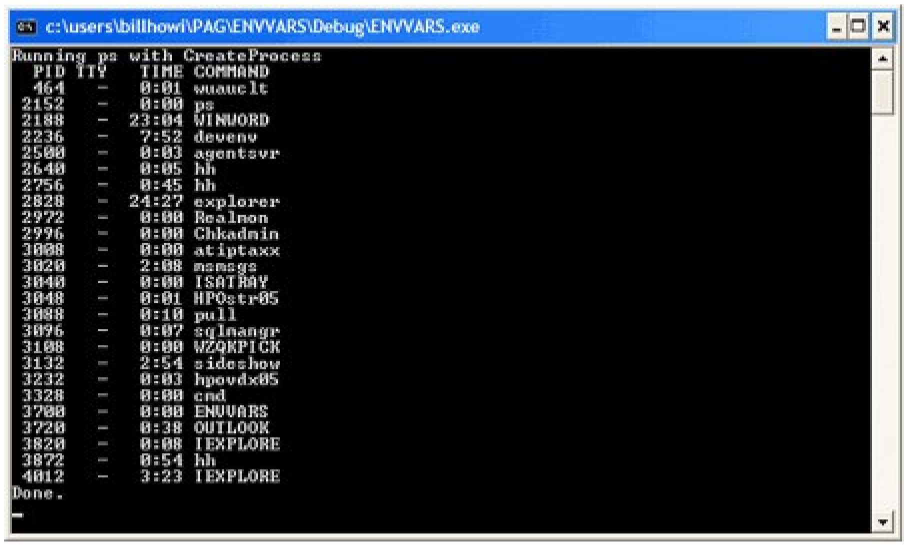
Creating a process in Windows using spawn

```
#include <windows.h>
#include <process.h>
#include <stdio.h>

void main()
{
    printf("Running Notepad with spawnlp\n");
    _spawnlp( _P_NOWAIT, "notepad", "notepad", NULL );

    printf("Done.\n");
    exit(0);
}
```

Running either of the above examples yields a console window similar to that shown here:



```
Running ps with CreateProcess
   PID TTY    TIME COMMAND
   464   -    0:01 wuauclt
  2152   -    0:00 ps
  2188   -   23:04 WINWORD
  2236   -    7:52 devenv
  2500   -    0:03 agentsvr
  2640   -    0:05 hh
  2756   -    0:45 hh
  2828   -   24:27 explorer
  2972   -    0:00 Realmon
  2996   -    0:00 Chkadmin
  3008   -    0:00 atiptaxx
  3020   -    2:08 msmsgs
  3040   -    0:00 ISATRAY
  3048   -    0:01 HPOstr05
  3088   -    0:10 pull
  3096   -    0:07 sqlmangr
  3108   -    0:00 WZQKPICK
  3132   -    2:54 sideshow
  3232   -    0:03 hpovdx05
  3328   -    0:00 cmd
  3700   -    0:00 ENVVARS
  3720   -    0:38 OUTLOOK
  3820   -    0:08 IEXPLORE
  3872   -    0:54 hh
  4012   -    3:23 IEXPLORE
Done.
_
```

Figure 1. Output from spawn example code

Replacing a Process Image (exec)

A UNIX application replaces the executing image with that of another application by using one of the **exec** functions. As mentioned previously, a **fork** followed by an **exec** is similar to **CreateProcess**.

Windows supports the six POSIX variants of the **exec** function plus two additional ones (**execlpe** and **execvpe**). The function signatures are identical, and come as part of the standard C runtime. Porting UNIX code that uses **exec** to Win32 is easy to understand. The following is a simple UNIX example showing the use of the **execlp** function.

> **Note** For more information on **exec** support on Win32, see the standard C runtime library documentation that comes with the Microsoft Visual Studio® development system.

Replacing a process image in UNIX using exec

```
#include <unistd.h>
#include <stdio.h>

int main()
{
    printf("Running ps with execlp\n");
    execlp("ps", "ps", "-l", 0);
    printf("Done.\n");
    exit(0);
}
```

The preceding example compiles and runs on Windows with only minor modifications. It does, however, require an executable file called ps.exe to be available (one is included with the Interix product).

The <unistd.h> include file is not a valid header file when using Windows. To use this example when using Windows, you need to change the header file to <process.h>. Doing so allows you to compile, link, and run this simple application.

Waiting for a Spawned Process

In the preceding section, the example showed how you can create an asynchronous process where the parent and child processes execute simultaneously. No synchronization was performed. This section describes how to modify the previous example to include functionality that enables the parent process to wait for the child process to complete or terminate before continuing.

To accomplish this in UNIX, a developer would use one of the **wait** functions to suspend the parent process until the child process terminates. The same semantics are available when using Windows. The functions used are different, but the results are the same.

When you view the examples, keep in mind that this is not an exhaustive comparison between the two platforms. A very simple scenario is described, but if you need to expand the scenario to include waiting for multiple child processes, the **spawn** example does not map adequately as it does not include support for this functionality. In this case, you need to consider the **CreateProcess** approach and **WaitForMultipleObjects**.

To see the code for this example, see Appendix G: Waiting for a Spawned Process.

Process vs. Threads

In the next example, the UNIX code is forking a process, but not executing a separate runtime image. This creates a separate execution path within the application. When using Windows, this is achieved by using threads rather than processes. If your UNIX application creates separate threads of execution in this manner, you should use the Win32 API **CreateThread**.

The process of creating threads is covered in the next section, Threads.

UNIX code with forking executable

```
#include <unistd.h>
#include <stdio.h>
#include <sys/types.h>
```

```
int main()
{
    pid_t pid;
    int n;

    printf("fork program started\n");
    pid = fork();
    switch(pid)
    {
    case -1:
        perror("fork failed");
        exit(1);
    case 0:
        puts("I'm the child");
        break;
    default:
        puts("I'm the parent");
        break;
    }
    exit(0);
}
```

Managing Process Resource Limits

Developers often want to create processes that run with a specific set of resource restrictions. In some cases, they may impose limitations for the purposes of stress testing or forced failure condition testing. In other cases, however, the limitations may be imposed to restrict runaway processes from using up all available memory, CPU cycles, or disk space.

In UNIX, the **getrlimit** function retrieves resource limits for a process, the **getrusage** function retrieves current usage, and **setrlimit** function sets new limits. The common limit names and their meanings are described in Table 1:

Table 1. Common limit names and definitions

Limit	Description
RLIMIT_CORE	The maximum size, in bytes, of a core file created by this process. If the core file is larger than RLIMIT_CORE, the write is terminated at this value. If the limit is set to 0, then no core files are created.
RLIMIT_CPU	The maximum time, in seconds, of CPU time a process can use. If the process exceeds this time, the system generates SIGXCPU for the process.
RLIMIT_DATA	Maximum size, in bytes, of a process's data segment. If the data segment exceeds this value, the functions **brk**, **malloc**, and **sbrk** will fail.
RLIMIT_FSIZE	The maximum size, in bytes, of a file created by a process. If the limit is 0, the process cannot create a file. If a write or truncation call exceeds the limit, further attempts will fail.
RLIMIT_NOFILE	The highest possible value for a file descriptor, plus one. This limits the number of file descriptors a process may

	allocate. If more than RLIMIT_NOFILE files are allocated, functions allocating new file descriptors may fail with the error EMFILE.
RLIMIT_STACK	The maximum size, in bytes, of a process's stack. The stack won't automatically exceed this limit; if a process tries to exceed the limit, the system generates SIGSEGV for the process.
RLIMIT_AS	Maximum size, in bytes, of a process's total available memory. If this limit is exceeded, the memory functions **brk**, **malloc**, **mmap**, and **sbrk** will fail with errno set to ENOMEM, and automatic stack growth will fail as described for RLIMIT_STACK.

Windows uses job objects to set job limits (rather than process limits). Unlike UNIX, Windows job objects do not have File input/output (I/O) source restrictions. If you require File I/O limits in your application, you need to create your own code to handle this.

Windows job objects

Windows supports the concept of job objects, which allows you to group one or more processes into a single entity. Once a job object has been populated with the desired processes, the entire group can be manipulated for various purposes ranging from termination to imposing resource restrictions.

The restrictions that job objects allow you to enforce are described in Table 2:

Table 2. Job objects

Member	Description	Notes
PerProcessUser-TimeLimit	Specifies the maximum user-mode time allotted to each process (in 100 ns intervals).	The system automatically terminates any process that uses more than its allotted time. To set this limit, specify the JOB_OBJECT_LIMIT_ PROCESS_TIME flag in the **LimitFlags** member.
PerJobUser-TimeLimit	Specifies how much more user-mode time the processes in this job can use (in 100 ns intervals).	By default, the system automatically terminates all processes when this time limit is reached. You can change this value periodically as the job runs. To set this limit, specify the JOB_OBJECT_LIMIT_JOB_TIME flag in the **LimitFlags** member.
LimitFlags	Specifies which restrictions to apply to the job.	See the job objects API reference for more information.
MinimumWorkingSetSize/ MaximumWorkingSetSize	Specifies the minimum and maximum working set size for each process (not for all processes within the job).	Normally, a process's working set can grow above its maximum; setting **MaximumWorkingSetSize** forces a hard limit. Once the process's working set reaches this limit, the process pages against itself. Calls to **SetProcessWorkingSetSize** by an individual process are ignored unless the process is just trying to empty its working set. To set this limit,

		specify the JOB_OBJECT_ LIMIT_WORKINGSET flag in the **LimitFlags** member.
ActiveProcessLimit	Specifies the maximum number of processes that can run concurrently in the job.	Any attempt to go over this limit causes the new process to be terminated with a "not enough quota" error. To set this limit, specify the JOB_OBJECT_ LIMIT_ACTIVE_PROCESS flag in the **LimitFlags** member.
Affinity	Specifies the subset of the CPU(s) that can run the processes.	Individual processes can limit this even further. To set this limit, specify the JOB_OBJECT_ LIMIT_AFFINITY flag in the **LimitFlags** member.
PriorityClass	Specifies the priority class that all processes use.	If a process calls **SetPriorityClass**, the call will return successfully even though it actually fails. If the process calls **GetPriorityClass**, the function returns what the process has set the priority class to even though this might not be process's actual priority class. In addition, **SetThreadPriority** fails to raise threads above normal priority but can be used to lower a thread's priority. To set this limit, specify the JOB_OBJECT_LIMIT_PRIORITY_CLASS flag in the **LimitFlags** member.
SchedulingClass	Specifies a relative time quantum difference assigned to threads in the job.	Value can be from 0 to 9 inclusive; 5 is the default. See the text after this table for more information. To set this limit, specify the JOB_OBJECT_LIMIT_SCHEDULING_CLASS flag in the **LimitFlags** member.

As you may have observed by reviewing the table for setrlimit and job objects, the restrictions offered by job objects are comparable except in one major area: File I/O.

Limiting file I/O when using Windows

When a process is created in UNIX, the Process Control Block (PCB) in kernel space contains an array of limits that is initialized with default values. In the case of the RLIMIT_FSIZE limit, the write procedures in the kernel are aware of the limit structure in the PCB, and these functions make checks to enforce the limits. The Windows operating system does not implement similar limits on files. To solve this problem, you must write your own solution and build it into your application.

This section presents a solution that you could use in your application. This solution emulates the UNIX file resource limits with:

- An array of limits held as a static variable.

  This is similar to how some of the C runtime functions use static variables.

- Our own versions of the UNIX functions **getrlimit()** and **setrlimit()**.

  These functions manipulate the limit array.

- Wrappers for each of the disk write functions.

  These wrappers are resource limit aware.

This solution is implemented as three files. Two of the files, resource.h and resource.c, implement the **getrlimit()**, **setrlimit()**, **rfwrite()** and **_rwrite()** functions. Only **fwrite()** and **_write()** are wrapped since they are the most common disk write functions encountered in the UNIX world. The third file is rlimit.c which is a very simple test program used to confirm that **rfwrite()** will fail when the limit was reached.

For more information, see Appendix B: Limiting File I/O.

Process Accounting

The Win32 API has several functions for gathering process accounting information:

- **GetProcessShutdownParameters**
- **GetProcessTimes**
- **GetProcessWorkingSetSize**
- **SetPriorityClass**
- **SetProcessShutdownParameters**
- **SetProcessWorkingSetSize**

Alternatively, a better method of obtaining process information is through the Windows Management Instrumentation (WMI) API.

For more information on WMI, see Windows Management Instrumentation (WMI) Tools.

Signals and Signal Handling

The UNIX operating system supports a wide range of signals. UNIX signals are software interrupts that catch or indicate different types of events. Windows on the other hand supports only a small set of signals that is restricted to exception events only. Consequently, converting UNIX code to Win32 requires the use of new techniques replacing the use of some UNIX signals.

The Windows signal implementation is limited to the following signals (Table 3):

Table 3. Windows signals

Signal	Meaning
SIGABRT	Abnormal termination
SIGFPE	Floating-point error
SIGILL	Illegal instruction
SIGINT	CTRL+C signal
SIGSEGV	Illegal storage access
SIGTERM	Termination request

> **Note** When a Ctrl+C interrupt occurs, Win32 operating systems generate a new thread to handle the interrupt. This can cause a single-thread application, such as one ported from UNIX, to become multithreaded, potentially resulting in unexpected behavior.

When an application uses other signals not supported in Windows, you have two choices:

- Use additional libraries that provide required signals, such as those provided by Microsoft Windows Services for UNIX.
- Use a comparable Windows mechanism, such as Windows Messages.

This section focuses on the Windows mechanisms that you can use to replace the use of some UNIX signals. Table 4 shows the recommended mechanisms that you can use to replace common UNIX signals. There are three main mechanisms:

- Native signals
- Event objects
- Messages

Table 4. UNIX signals and replacement mechanisms

Signal name	Description	Link to reference material
SIGABRT	Abnormal termination	SIGABRT
SIGALRM	Time-out alarm	SetTimer–WM_TIMER - CreateWaitableTimer
SIGCHLD	Change in status of child	WaitForSingleObject
SIGCONT	Continue stopped process	WaitForSingleObject
SIGFPE	Floating point exception	SIGFPE
SIGHUP	Hangup	NA
SIGILL	Illegal hardware instruction	SIGILL
SIGINT	Terminal interrupt character	WM_CHAR
SIGKILL	Termination	WM_QUIT
SIGPIPE	Write to pipe with no readers	WaitForSingleObject
SIGQUIT	Terminal Quit character	WM_CHAR
SIGSEGV	Invalid memory reference	SIGSEGV
SIGSTOP	Stop process	WaitForSingleObject
SIGTERM	Termination	SIGTERM
SIGTSTP	Terminal Stop character	WM_CHAR
SIGTTIN	Background read from control tty	NA
SIGTTOU	Background write to control tty	NA
SIGUSR1	User defined signal	SendMessage–WM_APP
SIGUSR2	User defined signal	SendMessage–WM_APP

Note Only POSIX signals are considered in this table (that is, Seventh Edition, System V, and BSD signals are not).

This section discusses how you can use the three mechanisms in Table 4 to convert the parts of your code that use signals into the Windows environment.

Another mechanism that can be useful when converting some UNIX uses of signals to Windows is event kernel objects. For more information on these objects, see the **CreateEvent** example in the Logging System Messages section later in this chapter.

Using Native Signals in Windows

In the following example, the simple case of catching SIGINT to detect Ctrl-C is demonstrated. As you can see from the two source listings, support for handling native signals in UNIX and Win32 is very similar.

Managing signals in UNIX

```
#include <unistd.h>
#include <stdio.h>
#include <signal.h>

/*  The intrpt function reacts to the signal passed in the parameter
signum.
    This function is called when a signal occurs.
    A message is output, then the signal handling for SIGINT is reset
    (by default generated by pressing CTRL-C) back to the default
behavior.
*/
void intrpt(int signum)
{
    printf("I got signal %d\n", signum);
    (void) signal(SIGINT, SIG_DFL);
}

/*  main intercepts the SIGINT signal generated when Ctrl-C is input.
    Otherwise, sits in an infinite loop, printing a message once a second.
*/
int main()
{
    (void) signal(SIGINT, intrpt);

    while(1) {
        printf("Hello World!\n");
        sleep(1);
    }
}
```

Managing signals in Windows

```
#include <windows.h>
#include <signal.h>
#include <stdio.h>

void intrpt(int signum)
{
    printf("I got signal %d\n", signum);
    (void) signal(SIGINT, SIG_DFL);
}

/*  main intercepts the SIGINT signal generated when Ctrl-C is input.
    Otherwise, sits in an infinite loop, printing a message once a second.
*/

void main()
{
    (void) signal(SIGINT, intrpt);
    while(1) {
        printf("Hello World!\n");
        Sleep(1000);
    }
}
```

> **Note** By default, signal terminates the calling program with exit code 3, regardless of the value of sig. For more information, see the signal topic in the Visual C++ Run Time Library Reference.

With the exception of requiring an additional header file, and the different signature of the **sleep** function, these two examples are identical. Unfortunately, this is the extent of the similarities in signal handling between the two platforms.

Replacing UNIX Signals Within Windows

UNIX uses signals to send alerts to processes when specific actions occur. A UNIX application would use the **kill** function to activate signals internally. As discussed earlier, Win32 provides only limited support for signals. As a result, you have to rewrite your code to use another form of event notification in Win32.

The following example illustrates how you would convert UNIX code to Windows Messages or Event Objects. It shows a simple main that forks a child process, which issues the SIGALRM signal. The parent process catches the alarm and outputs a message when it is received.

Using the SIGALRM signal in UNIX

```
#include <unistd.h>
#include <stdio.h>
#include <signal.h>

static int alarm_fired = 0;

/*  The alrm_bell function simulates an alarm clock.  */
void alrm_bell(int sig)
{
    alarm_fired = 1;
}

int main()
{
    int pid;

/*  Child process waits for 5 sec's before sending SIGALRM to its parent. */
    printf("alarm application starting\n");
    if((pid = fork()) == 0) {
        sleep(5);
        kill(getppid(), SIGALRM);
        exit(0);
    }

/*  Parent process arranges to catch SIGALRM with a call to signal
    and then waits for the child process to send SIGALRM. */
    printf("waiting for alarm\n");
    (void) signal(SIGALRM, alrm_bell);
    pause();
    if (alarm_fired)
        printf("Ring...Ring!\n");

    printf("alarm application done\n");
    exit(0);
}
```

Replacing UNIX Signals with Windows Messages

In the first Win32 example below, a form of Microsoft Windows Messages is used to signal the parent process. In the example, the **SetTimer** function is used to signal the parent process that an alarm has been activated. While code could have been created to do the timing, using the **SetTimer** function greatly simplifies this example.

Another advantage of using **SetTimer** is that the **callback** function is invoked in the same thread that calls **SetTimer**. No synchronization is necessary.

If the requirements are simple, consider using a thread to act as a timer thread, which simply calls **Sleep** to create the desired delay. At the end of the delay, a call is made to a timer **callback** function. The problem with this approach is that the **callback** function is called from a different thread than your primary thread. If the **callback** function requires resources that are thread specific, you will need to use one of the appropriate synchronization mechanisms discussed later in the "Threads" section.

Additional code has been added to the example so that an application using this code can catch any standard Windows message as well as application and user defined messages. You can use these messages to engineer solutions to other signals that are not directly supported by the native signal implementation in Win32.

Replacing SIGALRM using Windows messages

```
#include <windows.h>
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>

static int alarm_fired = 0;

/*  The alrm_bell function simulates an alarm clock. */
VOID CALLBACK alrm_bell(HWND hwnd, UINT uMsg, UINT idEvent, DWORD
dwTime )
{
    alarm_fired = 1;
    printf("Ring...Ring!\n");
}

void main()
{
    printf("alarm application starting\n");

/* Set up a 5 second timer which calls alrm_bell */
    SetTimer(0, 0, 5000, (TIMERPROC)alrm_bell);

    printf("waiting for alarm\n");
    MSG msg = { 0, 0, 0, 0 };

/*  Get the message, & dispatch.  This causes alrm_bell to be
invoked. */
    while(!alarm_fired)
        if (GetMessage(&msg, 0, 0, 0) ) {
            if (msg.message == WM_TIMER)
                printf("WM_TIMER\n");
            DispatchMessage(&msg);
        }
    printf("alarm application done\n");
    exit(0);
}
```

Notice in this example that the WM_TIMER message is issued and captured by the **GetMessage** function. If you remove the call to **DispatchMessage**, the **alrm_bell** function would never be called, but the WM_TIMER message would be received. With this simple application, you can capture a variety of Windows messages. Moreover, if you want to trigger the **callback** function before the specified time, you can use the **PostMessage**(WM_TIMER) call. This is analogous to using the **kill** function to send a signal in UNIX.

Replacing UNIX Signals with Windows Event Objects

Some events that UNIX handles through signals are represented in Win32 as objects. Functions are available to integrate these event objects. An example of these functions is **WaitForSingleObject**.

In the example code below, a timer object is used to signal when a timed interval has elapsed. Again, this example provides the same functionality as the UNIX SIGALRM example above.

> **Note** While this illustration encompasses the process in a single thread, this is not a requirement. The timer object can be tested and waited for in other threads if necessary.

Replacing SIGALRM using event objects

```
#define _WIN32_WINNT 0X0500

#include <windows.h>
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>

void main()
{
    HANDLE hTimer = NULL;
    LARGE_INTEGER liDueTime;

    liDueTime.QuadPart = -50000000;

    printf("alarm application starting\n");

// Set up a 5 second timer object
    hTimer = CreateWaitableTimer(NULL, TRUE, "WaitableTimer");
    SetWaitableTimer(hTimer, &liDueTime, 0, NULL, NULL, 0);

// Now wait for the alarm
    printf("waiting for alarm\n");
// Wait for the timer object
    WaitForSingleObject(hTimer, INFINITE);
    printf("Ring...Ring!\n");
    printf("alarm application done\n");
    exit(0);
}
```

Porting the Sigaction Call

Win32 does not support **sigaction**. The UNIX example below shows how **sigaction** is typically used in a UNIX application. In this example, the handler for the SIGALRM signal has been set. How this code can be converted to use Windows Messages was shown earlier. You could also use Windows Messages here if you prefer.

Note To terminate this application from the keyboard, press CTRL+\.

```
#include <unistd.h>
#include <stdio.h>
#include <signal.h>

void intrpt(int signum)
{
    printf("I got signal %d\n", signum);
}

int main()
{
    struct sigaction act;

    act.sa_handler = intrpt;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;

    sigaction(SIGINT, &act, 0);

  while(1) {
    printf("Hello World!\n");
    sleep(1);
  }
}
```

Threads

A thread is an independent path of execution in a process that shares the address space, code, and global data of the process. Time slices are allocated to each thread based on priority, and consist of an independent set of registers, stack, I/O handles, and message queue. Threads can usually run on separate processors on a multiprocessor computer. Win32 enables you to assign threads to a specific processor on a multiprocessor hardware platform.

An application using multiple processes usually has to implement some form of interprocess communication (IPC). This can result in significant overhead, and possibly a communication bottleneck. In contrast, threads share the process data between them, and interthread communication can be much faster. The problem with threads sharing data is that this can lead to data access conflicts between multiple threads. You can address these conflicts using synchronization techniques, such as semaphores and mutexes.

In UNIX, developers implement threads by using the POSIX **pthread** functions. In Win32, developers can implement UNIX threading by using the Win32 API thread management functions. The functionality and operation of threads in UNIX and Win32 is very similar; however, the function calls and syntax are very different.

The following are some similarities between UNIX and Windows:

- Every thread must have an entry point. The name of the entry point is entirely up to you so long as the signature is unique and the linker can adequately resolve any ambiguity.
- Each thread is passed a single parameter when it is created. The contents of this parameter are entirely up to the developer and have no meaning to the operating system.
- A thread function must return a value.
- A thread function needs to use local parameters and variables as much as possible. When you use global variables or shared resources, threads must use some form of synchronization to avoid potentially clobbering and corrupting data.

This section looks at how you should go about converting UNIX threaded applications into Win32 thread applications. As you know from the preceding section about processes, you may also have decided to convert some of your application's use of UNIX processes into threads.

> **Note** More information about programming with threads in Win32 can be found on the MSDN Web site at Multithreading: Programming Tips_core_multithreading.3a_.programming_tips
>
> For details on thread management functions in the Win32 API, see the Win32 API reference in Visual Studio or MSDN.

Creating a Thread

When creating a thread in UNIX, use the **pthread_create** function. This function has three arguments: a pointer to a data structure that describes the thread, an argument specifying the thread's attributes (usually set to NULL indicating default settings) and the function the thread will run. The thread finishes execution with a **pthread_exit**, where in this case, it returns a string. The process can wait for the thread to complete using the function **pthread_join**.

This simple UNIX example below creates a thread and waits for it to finish.

Creating a single thread in UNIX

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

char message[] = "Hello World";

void *thread_function(void *arg) {
    printf("thread_function started. Arg was %s\n", (char *)arg);
    sleep(3);
    strcpy(message, "Bye!");
    pthread_exit("See Ya");
}

int main() {
    int res;
    pthread_t a_thread;
    void *thread_result;
    res = pthread_create(&a_thread, NULL, thread_function, (void
*)message);
    if (res != 0) {
        perror("Thread creation failed");
        exit(EXIT_FAILURE);
    }
    printf("Waiting for thread to finish...\n");
    res = pthread_join(a_thread, &thread_result);
    if (res != 0) {
        perror("Thread join failed");
        exit(EXIT_FAILURE);
    }
    printf("Thread joined, it returned %s\n", (char *)thread_result);
    printf("Message is now %s\n", message);
    exit(EXIT_SUCCESS);
}
```

In Win32, threads are created using the **CreateThread** function. **CreateThread** requires:

- The size of the thread's stack
- The security attributes of the thread
- The address at which to begin execution of a procedure
- An optional 32 bit value that is passed to the thread's procedure
- Flags that permit the thread priority to be set
- An address to store the system-wide unique thread identifier

Once a thread is created, the thread identifier can be used to manage the thread until it has terminated. The next example demonstrates how you should use **CreateThread** to create a single thread.

Creating a single thread in Windows

```
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>

char message[] = "Hello World";

DWORD WINAPI thread_function(PVOID arg) {
    printf("thread_function started. Arg was %s\n", (char *)arg);
    Sleep(3000);
    strcpy(message, "Bye!");
    return 100;
}

void main() {
    HANDLE a_thread;
    DWORD a_threadId;
    DWORD thread_result;

  // Create a new thread.
```

```
    a_thread = CreateThread(NULL, 0, thread_function, (PVOID)message,
0, &a_threadId);

    if (a_thread == NULL) {
        perror("Thread creation failed");
        exit(EXIT_FAILURE);
    }
    printf("Waiting for thread to finish...\n");
    if (WaitForSingleObject(a_thread, INFINITE) != WAIT_OBJECT_0) {
        perror("Thread join failed");
        exit(EXIT_FAILURE);
    }

    // Retrieve the code returned by the thread.
    GetExitCodeThread(a_thread, &thread_result);

    printf("Thread joined, it returned %d\n", thread_result);
    printf("Message is now %s\n", message);
    exit(EXIT_SUCCESS);
}
```

The UNIX and Win32 examples have roughly equivalent semantics. There are only two notable differences:

- The thread function in the Win32 code cannot return a string value. Developers must use some other means to convey the string message back to the parent (for example, returning an index into a string array).
- The Win32 version of the thread function simply returns a DWORD value rather than calling a function to terminate the thread. **ExitThread** could have been called, but this is not necessary because **ExitThread** is called automatically upon the return from the thread procedure. **TerminateThread** could also be called, but this isn't necessary, nor is it recommended. This is because TerminateThread causes the thread to exit unexpectedly. The thread then has no chance to execute any user-mode code and its initial stack in not deallocated. Furthermore, any DLLs attached to the thread are not notified that the thread is terminating. For more information, see Process and Thread Functions.

The two solutions have vastly different syntaxes. Win32 uses a different set of API calls to manage threads. As a result, the relevant data elements and arguments are considerably different.

Canceling a Thread

The details of terminating threads differ significantly between UNIX and Win32. While both environments allow threads to block termination entirely, UNIX offers additional facilities that allow a thread to specify if it is to be terminated immediately or deferred until it reaches a safe recovery point. Moreover, UNIX provides a facility known as cancellation cleanup handlers, which a thread can push and pop from a stack that is invoked in a last-in-first-out order when the thread is terminated. These cleanup handlers are coded to clean up and restore any resources before the thread is actually terminated.

The Win32 API allows you to terminate a thread asynchronously. Unlike UNIX, in Win32 code you cannot create cleanup handlers and it is not possible for a thread to defer from being terminated. Therefore, it is recommended that you design your code so that threads terminate by returning an exit code and so that threads cannot be terminated forcibly. To do this, you should design your thread code to accept some form of message or event to signal that they should be terminated. Based on this notification, the thread logic can elect to execute cleanup-handling code and return normally.

To prevent a thread from being terminated, you should remove the security attributes for THREAD_TERMINATE from the thread object.

While forcing a thread to end by using **TerminateThread** is not recommended, for completeness, the following example shows how you could convert UNIX code that cancels a thread into Win32 code that cancels a thread using this method.

Canceling a thread in UNIX

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void *thread_function(void *arg) {
    int i, res;
    res = pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
    if (res != 0) {
        perror("Thread pthread_setcancelstate failed");
        exit(EXIT_FAILURE);
    }
    res = pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, NULL);
    if (res != 0) {
        perror("Thread pthread_setcanceltype failed");
        exit(EXIT_FAILURE);
    }
    printf("thread_function is running\n");
    for(i = 0; i < 10; i++) {
        printf("Thread is running (%d)...\n", i);
        sleep(1);
    }
    pthread_exit(0);
}

int main() {
    int res;
    pthread_t a_thread;
    void *thread_result;

    res = pthread_create(&a_thread, NULL, thread_function, NULL);
    if (res != 0) {
        perror("Thread creation failed");
        exit(EXIT_FAILURE);
    }
    sleep(3);
    printf("Cancelling thread...\n");
    res = pthread_cancel(a_thread);
    if (res != 0) {
        perror("Thread cancellation failed");
        exit(EXIT_FAILURE);
    }
    printf("Waiting for thread to finish...\n");
    res = pthread_join(a_thread, &thread_result);
    if (res != 0) {
        perror("Thread join failed");
        exit(EXIT_FAILURE);
    }
    exit(EXIT_SUCCESS);
}
```

Canceling a thread in Windows

```
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>

DWORD WINAPI thread_function(PVOID arg) {
    printf("thread_function is running. Argument was %s\n", (char
*)arg);
    for(int i = 0; i < 10; i++) {
        printf("Thread is running (%d)...\n", i);
        Sleep(1000);
    }
    return 100;
}

void main() {
    HANDLE a_thread;
    DWORD thread_result;


    // Create a new thread.
    a_thread = CreateThread(NULL, 0, thread_function, (PVOID)NULL, 0,
NULL);

    if (a_thread == NULL) {
        perror("Thread creation failed");
        exit(EXIT_FAILURE);
    }

  Sleep(3000);
  printf("Cancelling thread...\n");
  if (!TerminateThread(a_thread, 0)) {
    perror("Thread cancellation failed");
    exit(EXIT_FAILURE);
  }

    printf("Waiting for thread to finish...\n");
    WaitForSingleObject(a_thread, INFINITE);
    GetExitCodeThread(a_thread, &thread_result);

    exit(EXIT_SUCCESS);
}
```

When you compare the UNIX and Win32 examples, you can see that in the Win32 implementation the setting for the deferred termination is absent. This is because deferring termination is not supported in Win32. **TerminateThread** is not immediate and it is not predictable. The termination resulting from a **TerminateThread** call can occur at any point during the thread execution. In contrast, UNIX threads tagged as deferred can terminate when a safe cancellation point is reached.

If you need to match the UNIX behavior in your Win32 application exactly you must create your own cancellation code, and thereby prevent the thread from being forcibly terminated.

Thread Synchronization

When you have more than one thread executing simultaneously, you have to take the initiative to protect shared resources. For example, if your thread increments a variable, you cannot predict the result as the variable may have been modified by another thread before or after the increment. The reason that you cannot predict the result is that the order in which threads have access to a shared resource is indeterminate.

The following example illustrates code that is, in principle, indeterminate.

> **Note** This is a very simple example and on most computers the result would always be the same, but the important point to note is that this is not guaranteed.

The main thread in the below example is represented by the parent. It generates a "P", and the child or secondary thread outputs a "T". A UNIX example and a Windows example are shown.

Multiple non-synchronized threads in UNIX

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void *thread_function(void *arg) {
    int count2;

    printf("thread_function is running. Argument was: %s\n", (char
*)arg);
    for (count2 = 0; count2 < 10; count2++) {
        sleep(1);
        printf("T");
    }
    sleep(3);
}

char message[] = "Hello I'm a Thread";

int main() {
    int count1, res;
    pthread_t a_thread;
    void *thread_result;

    res = pthread_create(&a_thread, NULL, thread_function, (void
*)message);
    if (res != 0) {
        perror("Thread creation failed");
        exit(EXIT_FAILURE);
    }
    printf("entering loop\n");
    for (count1 = 0; count1 < 10; count1++) {
        sleep(1);
        printf("P");
    }

    printf("\nWaiting for thread to finish...\n");
    res = pthread_join(a_thread, &thread_result);
    if (res != 0) {
        perror("Thread join failed");
        exit(EXIT_FAILURE);
    }
    printf("\nThread joined\n");
    exit(EXIT_SUCCESS);
}
```

Multiple non-synchronized threads in Windows

```
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>

DWORD WINAPI thread_function(PVOID arg) {
    int count2;

    printf("thread_function is running. Argument was: %s\n", (char
*)arg);
    for (count2 = 0; count2 < 10; count2++) {
        Sleep(1000);
        printf("T");
    }

    Sleep(3000);
    return 0;
}

char message[] = "Hello I'm a Thread";

void main() {
    HANDLE a_thread;
  DWORD a_threadId;
    DWORD thread_result;
    int count1;

  // Create a new thread.
    a_thread = CreateThread(NULL, 0, thread_function, (PVOID)message,
0, &a_threadId);

    if (a_thread == NULL) {
        perror("Thread creation failed");
        exit(EXIT_FAILURE);
    }

    printf("entering loop\n");
    for (count1 = 0; count1 < 10; count1++) {
        Sleep(1000);
        printf("P");
    }

    printf("\nWaiting for thread to finish...\n");
  if (WaitForSingleObject(a_thread, INFINITE) != WAIT_OBJECT_0) {
        perror("Thread join failed");
        exit(EXIT_FAILURE);
    }

  // Retrieve the code returned by the thread.
  GetExitCodeThread(a_thread, &thread_result);
```

```
    printf("\nThread joined\n");
    exit(EXIT_SUCCESS);
}
```

No actual synchronization between these two threads is being performed, and each thread uses the same, shared variable. If the threads were running serially, you'd see output like the following:

```
MOV EAX, 2           ; Thread 1: Move 2 into a register.
MOV [run_now], EAX   ; Thread 1: Store 2 in run_now.

MOV EAX, 1           ; Thread 2: Move 1 into a register.
MOV [run_now], EAX   ; Thread 2: Store 1 in run_now.
```

However, since there is no guarantee of the order that the threads will be executed in, you could have the following output:

```
MOV EAX, 2           ; Thread 1: Move 2 into a register.
MOV EAX, 1           ; Thread 2: Move 1 into a register.
MOV [run_now], EAX   ; Thread 1: Store 2 in run_now.

MOV [run_now], EAX   ; Thread 2: Store 1 in run_now.
```

It is not possible to predict the output that you will see from these examples. In most applications, unpredictable results are an undesirable feature. Consequently, it is important that you take great care in controlling access to shared resources in threaded code. UNIX and Windows provide mechanisms for controlling resource access. These mechanisms are referred to as synchronization techniques, which are discussed in the next few sections.

Interlocked exchange

A simple form of synchronization is to use what is known as an *interlocked exchange*. An interlocked exchange performs a single operation that cannot be preempted. The threads of different processes can only use this mechanism if the variable is in shared memory. The variable pointed to by the target parameter must be aligned on a 32-bit boundary; otherwise, this function will fail on multiprocessor x86 systems. Since this is not the case in the example, it does not help much, but it does illustrate the use of the **InterlockedExchange** functions.

Rewriting the previous Win32 example by using **InterlockedExchange** results in the following code:

Thread synchronization using interlocked exchange in Windows

```
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>

LONG new_value = 1;
char message[] = "Hello I'm a Thread";

DWORD WINAPI thread_function(PVOID arg) {
    int count2;

    printf("thread_function is running. Argument was: %s\n", (char
*)arg);
```

```
    for (count2 = 0; count2 < 10; count2++) {
        Sleep(1000);
        printf("(T-%d)", new_value);
        InterlockedExchange(&new_value, 1);
    }

    Sleep(3000);
    return 0;
}

void main() {
    HANDLE a_thread;
    DWORD a_threadId;
    DWORD thread_result;
    int count1;

// Create a new thread.
    a_thread = CreateThread(NULL, 0, thread_function, (PVOID)message,
0, &a_threadId);

    if (a_thread == NULL) {
        perror("Thread creation failed");
        exit(EXIT_FAILURE);
    }

    printf("entering loop\n");
    for (count1 = 0; count1 < 10; count1++) {
        Sleep(1000);
        printf("(P-%d)", new_value);
        InterlockedExchange(&new_value, 2);
    }

    printf("\nWaiting for thread to finish...\n");
  if (WaitForSingleObject(a_thread, INFINITE) != WAIT_OBJECT_0) {
        perror("Thread join failed");
        exit(EXIT_FAILURE);
    }

// Retrieve the code returned by the thread.
    GetExitCodeThread(a_thread, &thread_result);

    printf("\nThread joined\n");
    exit(EXIT_SUCCESS);
}
```

Synchronization with SpinLocks

In the previous example, as noted, you still have no synchronization between the two threads. The output may still be out of order. One simple mechanism that offers synchronization would be to implement a spin lock. To accomplish this, a variant of the Interlocked function called **InterlockedCompareExchange** is used.

```
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>

LONG run_now = 1;
char message[] = "Hello I'm a Thread";
```

```
DWORD WINAPI thread_function(PVOID arg) {
    int count2;

    printf("thread_function is running. Argument was: %s\n", (char
*)arg);
    for (count2 = 0; count2 < 10; count2++) {
        if (InterlockedCompareExchange(&run_now, 1, 2) == 2)
            printf("T-2");
        else
            Sleep(1000);
    }
    Sleep(3000);
    return 0;
}

void main() {
    HANDLE a_thread;
  DWORD a_threadId;
    DWORD thread_result;
    int count1;

// Create a new thread.
    a_thread = CreateThread(NULL, 0, thread_function, (PVOID)message,
0, &a_threadId);

    if (a_thread == NULL) {
        perror("Thread creation failed");
        exit(EXIT_FAILURE);
    }

    printf("entering loop\n");
    for (count1 = 0; count1 < 10; count1++) {
        if (InterlockedCompareExchange(&run_now, 2, 1) == 1)
            printf("P-1");
        else
            Sleep(1000);
    }

    printf("\nWaiting for thread to finish...\n");
    if (WaitForSingleObject(a_thread, INFINITE) != WAIT_OBJECT_0) {
        perror("Thread join failed");
        exit(EXIT_FAILURE);
    }

// Retrieve the code returned by the thread.
    GetExitCodeThread(a_thread, &thread_result);
    printf("\nThread joined\n");
    exit(EXIT_SUCCESS);
}
```

Spinlocks work well for synchronizing access to a single object, but most applications are not this simple. Moreover, using spinlocks is not the most efficient means to control access to a shared resource. Running a While loop in user mode while waiting for a global value to change wastes CPU cycles unnecessarily. A mechanism is needed that allows the thread to not waste CPU time while waiting to access a shared resource.

When a thread requires access to a shared resource (for example a shared memory object), it must either be notified or scheduled to resume execution. To accomplish this, a thread must call an operating system function, passing it parameters that indicate what the thread is waiting for. If the operating system detects that the resource is available, the function returns and the thread resumes.

If the resource is unavailable, the system places the thread in a wait state, making the thread not schedulable. This prevents the thread from wasting any CPU time. When a thread is waiting, the system permits the exchange of information between the thread and the resource. The operating system tracks the resources that a thread needs and automatically resumes the thread when the resource becomes available. The thread's execution is synchronized with the availability of the resource.

Mechanisms that prevent the thread from wasting CPU time include critical sections (for example, the **EnterCriticalSection** function waits for ownership of the specified critical section object, and returns when the calling thread has been granted ownership), semaphores and mutexes. Windows includes all three of these mechanisms, and UNIX provides both semaphores and mutexes. These three mechanisms are described in the following sections.

Synchronization with critical sections

Another mechanism for solving this simple scenario is to use a critical section. A critical section is similar to **InterlockedExchange** except that you have the ability to define the logic that takes place as an atomic operation.

What follows is the simple example from the previous section with the **InterlockedExchange** replaced with critical sections. On multiprocessor systems, it's best to use **InitializeCriticalSectionAndSpinCount**, instead of **InitializeCriticalSection,** which provides an optimized version of critical sections by employing spin counting. A critical section with spin locking allows the **EnterCriticalSection** to be tried up to spin count times before transitioning into kernel mode to wait for the resource. The advantage to this is that the transition into kernel mode requires approximately 1,000 CPU cycles.

Moreover, there is a slight chance that entering a critical section may fail due to memory limitations. The **InitializeCriticalSectionAndSpinCount** form of the critical section function then returns a status of STATUS_NO_MEMORY. This is an improvement over the **InitializeCriticalSection** function, which does not return any status as can be determined by its **void** return type.

Critical section code is highlighted in bold.

```
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>

CRITICAL_SECTION g_cs;
char message[] = "Hello I'm a Thread";

DWORD WINAPI thread_function(PVOID arg) {
    int count2;

    printf("\nthread_function is running. Argument was: %s\n", (char
*)arg);
    for (count2 = 0; count2 < 10; count2++) {
        EnterCriticalSection(&g_cs);
        printf("T");
        LeaveCriticalSection(&g_cs);
    }
    Sleep(3000);
    return 0;
}

void main() {
    HANDLE a_thread;
    DWORD a_threadId;
    DWORD thread_result;
    int count1;

    InitializeCriticalSection(&g_cs);

// Create a new thread.
    a_thread = CreateThread(NULL, 0, thread_function, (PVOID)message,
0, &a_threadId);

    if (a_thread == NULL) {
        perror("Thread creation failed");
        exit(EXIT_FAILURE);
    }

    printf("entering loop\n");
    for (count1 = 0; count1 < 10; count1++) {
        EnterCriticalSection(&g_cs);
        printf("P");
        LeaveCriticalSection(&g_cs);
    }

    printf("\nWaiting for thread to finish...\n");
  if (WaitForSingleObject(a_thread, INFINITE) != WAIT_OBJECT_0) {
        perror("Thread join failed");
        exit(EXIT_FAILURE);
    }

// Retrieve the code returned by the thread.
    GetExitCodeThread(a_thread, &thread_result);
    printf("\nThread joined\n");
    DeleteCriticalSection(&g_cs);
    exit(EXIT_SUCCESS);
}
```

Synchronization using semaphores

In the following example, two threads are created that use a shared memory buffer. Access to the shared memory is synchronized using a semaphore. The primary thread (**main**) creates a semaphore object and uses this object to handshake with the secondary thread (**thread_function**). The primary thread instantiates the semaphore in a state that prevents the secondary thread from acquiring the semaphore while it is initiated.

The primary thread relinquishes the semaphore once the user types in some text at the console and presses return. Once this is done, the secondary thread acquires the semaphore and processes the shared memory area. At this point, the main thread is blocked waiting for the semaphore, and will not resume until the secondary thread has relinquished control by calling **ReleaseSemaphore**.

These two examples are somewhat different. In UNIX, the semaphore object functions of **sem_post** and **sem_wait** are all that are required to perform handshaking. With Win32, you must use a combination of

WaitForSingleObject and **ReleaseSemaphore** in both the primary and the secondary threads in order to facilitate handshaking. The two solutions are also very different from a syntactic standpoint. The primary difference between their implementations is with the API calls that are used to manage the semaphore objects.

One aspect of **CreateSemaphore** that you need to be aware of is the last argument in its parameter list. This is a string parameter specifying the name of the semaphore. You should not pass a NULL for this parameter. Most (but not all) of the kernel objects, including semaphores, are named. All kernel object names are stored in a common namespace except if it is a server running Microsoft Terminal Server, in which case there will also be a namespace for each session. If the namespace is global, one or more unassociated applications could attempt to use the same name for a semaphore. To avoid namespace contention, applications should use some unique naming convention. One solution you could use would be to base your semaphore names on globally unique identifiers (GUIDs).

Terminal server and naming semaphore objects

As mentioned earlier, Terminal Servers have multiple namespaces for kernel objects. There is one global namespace, which is used by kernel objects that are accessible by any and all client sessions and is usually populated by services. Additionally, each client session has its own namespace to prevent namespace collisions between multiple instances of the same application running in different sessions.

In addition to the session and global namespaces, Terminal Servers also have a local namespace. By default, an application's named kernel objects reside in the session namespace. It is possible, however, to override what namespace will be used. This is accomplished by prefixing the name with Global\ or Local\. These prefix names are reserved by Microsoft, are case sensitive and are ignored if the computer is not operating as a Terminal Server.

UNIX example: synchronization using semaphores

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>

#define SHARED_SIZE 1024
char shared_area[SHARED_SIZE];
sem_t bin_sem;

void *thread_function(void *arg) {
    sem_wait(&bin_sem);
    while(strncmp("done", shared_area, 4) != 0) {
        printf("You input %d characters\n", strlen(shared_area) -1);
        sem_wait(&bin_sem);
    }
    pthread_exit(NULL);
}

int main() {
    int res;
    pthread_t a_thread;
    void *thread_result;

    res = sem_init(&bin_sem, 0, 0);
    if (res != 0) {
        perror("Semaphore initialization failed");
        exit(EXIT_FAILURE);
    }
    res = pthread_create(&a_thread, NULL, thread_function, NULL);
    if (res != 0) {
        perror("Thread creation failed");
        exit(EXIT_FAILURE);
    }
    printf("Input some text. Enter 'done' to finish\n");
    while(strncmp("done", shared_area, 4) != 0) {
        fgets(shared_area, SHARED_SIZE, stdin);
        sem_post(&bin_sem);
    }
    printf("\nWaiting for thread to finish...\n");
    res = pthread_join(a_thread, &thread_result);
    if (res != 0) {
        perror("Thread join failed");
        exit(EXIT_FAILURE);
    }
    printf("\nThread joined\n");
    sem_destroy(&bin_sem);
    exit(EXIT_SUCCESS);
}
```

Win32 example: synchronization using semaphores

```
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>

#define SHARED_SIZE 1024
char shared_area[SHARED_SIZE];
LPCTSTR lpszSemaphore = "SEMAPHORE-EXAMPLE";
HANDLE sem_t;

DWORD WINAPI thread_function(PVOID arg) {
   LONG dwSemCount;
   HANDLE hSemaphore = OpenSemaphore( SYNCHRONIZE |
SEMAPHORE_MODIFY_STATE, FALSE, lpszSemaphore );

   WaitForSingleObject( hSemaphore, INFINITE );
   while(strncmp("done", shared_area, 4) != 0) {
       printf("You input %d characters\n", strlen(shared_area) -1);
       ReleaseSemaphore(hSemaphore, 1, &dwSemCount);
       WaitForSingleObject( hSemaphore, INFINITE );
   }
   ReleaseSemaphore(hSemaphore, 1, &dwSemCount);
   CloseHandle( hSemaphore );
   return 0;
}

void main() {
    HANDLE a_thread;
    DWORD a_threadId;
    DWORD thread_result;
    LONG dwSemCount;

// Initialize Semaphore object.
  sem_t = CreateSemaphore( NULL, 0, 1, lpszSemaphore );
  if (sem_t == NULL) {
        perror("Semaphore initialization failed");
        exit(EXIT_FAILURE);
    }

// Create a new thread.
    a_thread = CreateThread(NULL, 0, thread_function, (PVOID)NULL, 0,
&a_threadId);

    if (a_thread == NULL) {
        perror("Thread creation failed");
        exit(EXIT_FAILURE);
    }

    printf("Input some text. Enter 'done' to finish\n");
    while(strncmp("done", shared_area, 4) != 0) {
        fgets(shared_area, SHARED_SIZE, stdin);
        ReleaseSemaphore(sem_t, 1, &dwSemCount);
    WaitForSingleObject(sem_t, INFINITE);
    }

    printf("\nWaiting for thread to finish...\n");
    if (WaitForSingleObject(a_thread, INFINITE) != WAIT_OBJECT_0) {
        perror("Thread join failed");
        exit(EXIT_FAILURE);
    }

// Retrieve the code returned by the thread.
    GetExitCodeThread(a_thread, &thread_result);

    printf("\nThread joined\n");
    exit(EXIT_SUCCESS);
}
```

Synchronization using mutexes

A *mutex* is a kernel object that provides a thread with mutually exclusive access to a single resource. Any thread of the calling process can specify the mutex-object handle in a call to one of the **wait** functions. The single-object wait functions return when the state of the specified object is signaled. The state of a mutex object is signaled when it is not owned by any thread. When the mutex's state is signaled, one waiting thread is granted ownership, the mutex's state changes to nonsignaled, and the wait function returns. Only one thread can own a mutex at any given time. The owning thread uses the **ReleaseMutex** function to release its ownership.

The next example looks at the use of mutexes to coordinate access to a shared resource, and to handshake between two threads. The logic is virtually identical to the semaphore example in the previous section. The only real difference is that this example uses a mutex instead of a semaphore.

UNIX example: thread synchronization using mutexes

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
```

```
#define SHARED_SIZE 1024
char shared_area[SHARED_SIZE];
pthread_mutex_t shared_mutex; /* protects shared_area */

void *thread_function(void *arg) {
    pthread_mutex_lock(&shared_mutex);
    while(strncmp("done", shared_area, 4) != 0) {
        printf("You input %d characters\n", strlen(shared_area) -1);
        pthread_mutex_unlock(&shared_mutex);
        pthread_mutex_lock(&shared_mutex);
    }
    pthread_mutex_unlock(&shared_mutex);
    pthread_exit(0);
}

int main() {
    int res;
    pthread_t a_thread;
    void *thread_result;
    res = pthread_mutex_init(&shared_mutex, NULL);
    if (res != 0) {
        perror("Mutex initialization failed");
        exit(EXIT_FAILURE);
    }
    res = pthread_create(&a_thread, NULL, thread_function, NULL);
    if (res != 0) {
        perror("Thread creation failed");
        exit(EXIT_FAILURE);
    }
    pthread_mutex_lock(&shared_mutex);
    printf("Input some text. Enter 'done' to finish\n");
    while (strncmp("done", shared_area, 4) != 0) {
        fgets(shared_area, SHARED_SIZE, stdin);
        pthread_mutex_unlock(&shared_mutex);
        pthread_mutex_lock(&shared_mutex);
    }

    pthread_mutex_unlock(&shared_mutex);
    printf("\nWaiting for thread to finish...\n");
    res = pthread_join(a_thread, &thread_result);
    if (res != 0) {
        perror("Thread join failed");
        exit(EXIT_FAILURE);
    }
    printf("\nThread joined\n");
    pthread_mutex_destroy(&shared_mutex);
    exit(EXIT_SUCCESS);
}
```

Win32 example: thread synchronization using mutexes

```
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>

#define SHARED_SIZE 1024
char shared_area[SHARED_SIZE];
LPCTSTR lpszMutex = "MUTEX-EXAMPLE";
HANDLE shared_mutex;
```

```
DWORD WINAPI thread_function(PVOID arg) {
   HANDLE hMutex = OpenMutex(MUTEX_ALL_ACCESS, FALSE, lpszMutex);

   WaitForSingleObject( hMutex, INFINITE );
   while(strncmp("done", shared_area, 4) != 0) {
       printf("You input %d characters\n", strlen(shared_area) -1);
       ReleaseMutex(hMutex);
       WaitForSingleObject(hMutex, INFINITE);
   }
   ReleaseMutex(hMutex);
   CloseHandle(hMutex);
   return 0;
}

void main() {
    HANDLE a_thread;
    DWORD a_threadId;
    DWORD thread_result;

// Initialize Semaphore object.
  shared_mutex = CreateMutex( NULL, TRUE, lpszMutex );

  if (shared_mutex == NULL) {
        perror("Mutex initialization failed");
        exit(EXIT_FAILURE);
    }

// Create a new thread.
    a_thread = CreateThread(NULL, 0, thread_function, (PVOID)NULL, 0,
&a_threadId);

    if (a_thread == NULL) {
        perror("Thread creation failed");
        exit(EXIT_FAILURE);
    }

    printf("Input some text. Enter 'done' to finish\n");
    while(strncmp("done", shared_area, 4) != 0) {
        fgets(shared_area, SHARED_SIZE, stdin);
        ReleaseMutex(shared_mutex);
        WaitForSingleObject(shared_mutex, INFINITE);
    }

    ReleaseMutex(shared_mutex);
    printf("Waiting for thread to finish...\n");
    if (WaitForSingleObject(a_thread, INFINITE) != WAIT_OBJECT_0) {
        perror("Thread join failed");
        exit(EXIT_FAILURE);
    }

// Retrieve the code returned by the thread.
    GetExitCodeThread(a_thread, &thread_result);
    CloseHandle(shared_mutex);

    printf("Thread joined\n");
    exit(EXIT_SUCCESS);
}
```

Thread Attributes

There are a number of attributes associated with threads in UNIX that you need to convert into equivalent attributes in Win32. This section contrasts the UNIX and Win32 thread attributes and describes how you should convert your code. Table 5 lists the relevant UNIX thread attributes, and then each attribute is discussed individually:

Table 5. UNIX thread attributes

Attribute	Default values	Description
detachstate	PTHREAD_CREATE_JOINABLE PTHREAD_CREATE_DETACHED	Thread may be joined by other threads. Threads may not be waited on for termination.
inheritsched	PTHREAD_INHERIT_SCHED PTHREAD_EXPLICIT_SCHED	Scheduling parameters, policy, and scope are inherited from creating thread. Scheduling parameters for the newly created thread are specified in the thread attribute.
schedparam	—	Priority set to default for scheduling policy.
schedpolicy	SCHED_OTHER SCHED_FIFO SCHED_RR	Scheduling policy is determined by the system. Threads are scheduled in a first-in-first-out order. Threads are scheduled in a round-robin fashion.
Scope	PTHREAD_SCOPE_SYSTEM PTHREAD_SCOPE_PROCESS	Threads are scheduled system-wide. Threads are scheduled based on other threads in the owning process.
Stackaddr	N/A	Attribute not supported; address selected by the operating system.
Stacksize	0	Stack size inherited from process stack size attribute.

Detachstate

Detachstate indicates whether a thread can be waited on for termination. Within Win32, the same effect is achieved by closing any handles that exist for a given thread. Since a handle is required for one of the wait and thread management functions, without a handle, you are effectively stopped from acting on a thread. You can also control thread objects based on a security descriptor that is provided at the time the thread is created.

> **Note** For more information on Access-Control, see Platform SDK: Security: Access Control.

The handle returned by the **CreateThread** function has THREAD_ALL_ACCESS access to the thread object. When you call the **GetCurrentThread** function, the system returns a pseudohandle with the maximum access that the thread's security descriptor allows the caller.

The valid access rights for thread objects include the DELETE, READ_CONTROL, SYNCHRONIZE, WRITE_DAC, and WRITE_OWNER standard access rights, in addition to the thread-specific access rights shown in Table 6.

Table 6. Thread-specific access rights

Value	Meaning
SYNCHRONIZE	A standard right required to wait for the thread to exit.
THREAD_ALL_ACCESS	Specifies all possible access rights for a thread object.
THREAD_DIRECT_IMPERSONATION	Required for a server thread that impersonates a client.
THREAD_GET_CONTEXT	Required to read the context of a thread by using **GetThreadContext**.
THREAD_IMPERSONATE	Required to use a thread's security information directly without calling it by using a communication mechanism that provides impersonation services.
THREAD_QUERY_INFORMATION	Required to read certain information from the thread object.
THREAD_SET_CONTEXT	Required to write the context of a thread.
THREAD_SET_INFORMATION	Required to set certain information in the thread object.
THREAD_SET_THREAD_TOKEN	Required to set the impersonation token for a thread.
THREAD_SUSPEND_RESUME	Required to suspend or resume a thread.
THREAD_TERMINATE	Required to terminate a thread.

Inheritsched/schedparam/schedpolicy/scope

Inheritsched/schedparam/schedpolicy/scope indicates that the scheduling is either inherited from the thread that created the new thread, or is explicitly set. It also defines the policy and scope applied to scheduling threads. In Win32, by default, the priority class of a process is NORMAL_PRIORITY_CLASS. Use the **CreateProcess** function to specify the priority class of a child process when you create it.

If the calling process is IDLE_PRIORITY_CLASS or BELOW_NORMAL_PRIORITY_CLASS, the new process inherits this class. You use the **GetPriorityClass** function to determine the current priority class of a process and the **SetPriorityClass** function to change the priority class of a process.

Stacksize

The stack size applied to a thread is controlled at the time the thread is created by using **CreateThread**. The initial size of the stack is specified in bytes. The system rounds this value to the nearest page. If this parameter is of zero value, the new thread uses the default size for the executable.

Setting thread attributes

Now that the thread attributes have been described, let's take a look at a simple example of how the attributes of a thread can be set.

The UNIX example below makes some basic use of thread attributes. The corresponding Win32 example doesn't even need to use attributes to accomplish the same functionality. All that is required with Win32 is to create a thread that can't be acted upon by a wait. This can be accomplished by passing NULL as the **dwThreadId** parameter to **CreateThread**, and by closing the handle that is returned by the call.

The net effect of these combined activities effectively hinders an application's ability to manage the thread. This issue is addressed in the "Thread Scheduling and Priorities" section later in this chapter.

UNIX example setting thread attributes

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

char message[] = "Hello I'm a Thread";
int thread_finished = 0;

void *thread_function(void *arg) {
    printf("thread_function running. Arg was: %s\n", (char *)arg);
    sleep(4);
    printf("Second thread setting finished flag, and exiting now\n");
    thread_finished = 1;
    pthread_exit(NULL);
}

int main() {
    int count=0, res;
    pthread_t a_thread;
    void *thread_result;
    pthread_attr_t thread_attr;

    res = pthread_attr_init(&thread_attr);
    if (res != 0) {
        perror("Attribute creation failed");
        exit(EXIT_FAILURE);
    }
    res = pthread_attr_setdetachstate(&thread_attr,
PTHREAD_CREATE_DETACHED);
    if (res != 0) {
        perror("Setting detached attribute failed");
        exit(EXIT_FAILURE);
    }
    res = pthread_create(&a_thread, &thread_attr, thread_function,
 (void *)message);
    if (res != 0) {
        perror("Thread creation failed");
        exit(EXIT_FAILURE);
    }
    (void)pthread_attr_destroy(&thread_attr);
    while(!thread_finished) {
        printf("Waiting for thread to finish (%d)\n", ++count);
        sleep(1);
    }
    printf("Other thread finished, See Ya!\n");
    exit(EXIT_SUCCESS);
}
```

Win32 example: setting thread attributes

```
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>

char message[] = "Hello I'm a Thread";
int thread_finished = 0;

DWORD WINAPI thread_function(PVOID arg) {
    printf("\nthread_function running. Arg was: %s\n", (char *)arg);
    Sleep(4000);
    printf("Second thread setting finished flag, and exiting now\n");
    thread_finished = 1;
    return 100;
}

void main() {
    int count=0;
    HANDLE a_thread;

// Create a new thread.
    a_thread = CreateThread(NULL, 0, thread_function, (PVOID)message,
0, NULL);

    if (a_thread == NULL) {
        perror("Thread creation failed");
        exit(EXIT_FAILURE);
    }

    CloseHandle(a_thread);

    while(!thread_finished) {
        printf("Waiting for thread to finish (%d)\n", ++count);
        Sleep(1000);
    }
    printf("Other thread finished, See Ya!\n");

    exit(EXIT_SUCCESS);
}
```

Win32 security and thread objects

Threads are kernel objects. As such, they are protected by Windows security, and therefore a process must request permission to manipulate an object before attempts are made. The creator of the object can prevent an unauthorized user from doing anything with the object by denying access to it.

Object flags are covered as part of the thread discussion here, but this information also pertains to other kernel objects that are obtained by using one of the Win32 **Create** functions.

Until now, threads have been created in these solutions with a NULL security attribute. This indicated that the thread should be created using the default security, and that the returned handle should be inheritable. If you want to change the behavior of the previous example to prevent the thread handle from being inherited and or closed, you could use the **SetHandleInformation** function to accomplish this. The following is an example of this:

```
#define HANDLE_FLAG_INHERIT    0x00000001
#define HANDLE_FLAG_PROTECT_FROM_CLOSE 0x00000002

SetHandleInformation(hThread, HANDLE_FLAG_INHERIT,
HANDLE_FLAG_INHERIT);
SetHandleInformation(hThread, HANDLE_FLAG_PROTECT_FROM_CLOSE,
HANDLE_FLAG_PROTECT_FROM_CLOSE);
```

To change both flags in a single call you should bitwise OR the flags together. After this call, attempting to close the handle by using **CloseHandle** would result in an exception being raised.

Thread Scheduling and Priorities

This section looks at how you can change the scheduling priority of a thread in UNIX and Win32.

Ideally, you want to map Win32 priority classes to UNIX scheduling policies, and Win32 thread priority levels to UNIX priority levels. Unfortunately, it isn't this simple.

The priority level of a Win32 thread is determined by both the priority class of its process and its priority level. The priority class and priority level are combined to form the *base priority* of each thread.

Every thread in Windows has a base priority level determined by the thread's priority value and the priority class of its owning process. The operating system uses the base priority level of all executable threads to determine which thread gets the next slice of CPU time. Threads are scheduled in a round-robin fashion at each priority level, and only when there are no executable threads at a higher level will scheduling of threads at a lower level take place.

UNIX offers both round robin and FIFO scheduling algorithms, whereas Windows uses only round robin. This does not mean that Windows is less flexible; it simply means that any fine tuning that was performed on thread scheduling in UNIX has to be implemented differently when using Windows.

Table 7 shows the base priority levels for combinations of priority class and priority value.

Table 7. Process and thread priority

	Process priority class	Thread priority level
1	IDLE_PRIORITY_CLASS	THREAD_PRIORITY_IDLE
1	BELOW_NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_IDLE
1	NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_IDLE
1	ABOVE_NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_IDLE
1	HIGH_PRIORITY_CLASS	THREAD_PRIORITY_IDLE
2	IDLE_PRIORITY_CLASS	THREAD_PRIORITY_LOWEST
3	IDLE_PRIORITY_CLASS	THREAD_PRIORITY_BELOW_NORMAL
4	IDLE_PRIORITY_CLASS	THREAD_PRIORITY_NORMAL
4	BELOW_NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_LOWEST

5	IDLE_PRIORITY_CLASS	THREAD_PRIORITY_ABOVE_NORMAL
5	BELOW_NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_BELOW_NORMAL
5	Background NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_LOWEST
6	IDLE_PRIORITY_CLASS	THREAD_PRIORITY_HIGHEST
6	BELOW_NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_NORMAL
6	Background NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_BELOW_NORMAL
7	BELOW_NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_ABOVE_NORMAL
7	Background NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_NORMAL
7	Foreground NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_LOWEST
8	BELOW_NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_HIGHEST
8	NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_ABOVE_NORMAL
8	Foreground NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_BELOW_NORMAL
8	ABOVE_NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_LOWEST
9	NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_HIGHEST
9	Foreground NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_NORMAL
9	ABOVE_NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_BELOW_NORMAL
10	Foreground NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_ABOVE_NORMAL
10	ABOVE_NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_NORMAL
11	Foreground NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_HIGHEST
11	ABOVE_NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_ABOVE_NORMAL
11	HIGH_PRIORITY_CLASS	THREAD_PRIORITY_LOWEST
12	ABOVE_NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_HIGHEST
12	HIGH_PRIORITY_CLASS	THREAD_PRIORITY_BELOW_NORMAL
13	HIGH_PRIORITY_CLASS	THREAD_PRIORITY_NORMAL
14	HIGH_PRIORITY_CLASS	THREAD_PRIORITY_ABOVE_NORMAL
15	HIGH_PRIORITY_CLASS	THREAD_PRIORITY_HIGHEST
15	HIGH_PRIORITY_CLASS	THREAD_PRIORITY_TIME_CRITICAL
15	IDLE_PRIORITY_CLASS	THREAD_PRIORITY_TIME_CRITICAL
15	BELOW_NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_TIME_CRITICAL
15	NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_TIME_CRITICAL
15	ABOVE_NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_TIME_CRITICAL
16	REALTIME_PRIORITY_CLASS	THREAD_PRIORITY_IDLE
17	REALTIME_PRIORITY_CLASS	-7
18	REALTIME_PRIORITY_CLASS	-6
19	REALTIME_PRIORITY_CLASS	-5
20	REALTIME_PRIORITY_CLASS	-4
21	REALTIME_PRIORITY_CLASS	-3
22	REALTIME_PRIORITY_CLASS	THREAD_PRIORITY_LOWEST
23	REALTIME_PRIORITY_CLASS	THREAD_PRIORITY_BELOW_NORMAL
24	REALTIME_PRIORITY_CLASS	THREAD_PRIORITY_NORMAL
25	REALTIME_PRIORITY_CLASS	THREAD_PRIORITY_ABOVE_NORMAL
26	REALTIME_PRIORITY_CLASS	THREAD_PRIORITY_HIGHEST
27	REALTIME_PRIORITY_CLASS	3
28	REALTIME_PRIORITY_CLASS	4
29	REALTIME_PRIORITY_CLASS	5
30	REALTIME_PRIORITY_CLASS	6

THREAD_PRIORITY_TIME_CRITICAL

Managing thread priorities in Windows

The Win32 API provides a number of functions for managing thread priorities:

- **GetThreadContext**

  Returns the execution context of the specified thread. The following is an example showing the thread context:

  ```
  CONTEXT context;
  TCHAR szBuffer[128];
  Context.ContextFlags = CONTEXT_FULL | CONTEXT_DEBUG_REGISTERS;
  GetThreadContext( GetCurrentThread(), &context);
  printf("CS=%X, EIP=%X, FLAGS=%X, DR1=%X\n",
   context.SegCs, context.Eip, context.EFlags, context.Dr1);
  ```

- **GetThreadPriority**

  Returns the assigned thread priority level for the specified thread. The priority for the thread and the process class determine the thread's base priority level.

  To see how thread priority affects the system, a simple test like the one below could be added to a simple Windows application:

  ```
  SetThreadPriority(GetCurrentThread(), THREAD_PRIORITY_LOWEST);
  DWORD dwTicks = GetTickCount();
  for(long i = 0; i < 200000; i ++)
     for(long j = 0; j < 2000; j ++)
  printf("Test time=%ld\n", GetTickCount()-dwTicks);
  ```

  Adjusting the thread priority should yield different time deltas.

- **GetThreadPriorityBoost**

  Retrieves the priority boost control state of the specified thread.

  Threads have *dynamic priority*, meaning the priority that the scheduler uses to identify which thread will execute. Initially, a thread's priority is the same as its base priority, but the system may increase or decrease the priority to maintain thread responsiveness. Only threads with a priority between 0 and 15 are eligible for dynamic priority boost.

  The system boosts the dynamic priority of a thread to enhance its responsiveness as follows:
  - When a process that uses NORMAL_PRIORITY_CLASS is brought to the foreground, the scheduler boosts the priority class of the process associated with the foreground window so that it is greater than or equal to the priority class of any background processes. The priority class returns to its original setting when the process is no longer in the foreground.

    In the Microsoft Windows NT® operating system, as well as in Windows 2000 or later, the user can control the boosting of processes that use NORMAL_PRIORITY_CLASS through the system Control Panel.

    - When a window receives input, such as timer messages, mouse messages or keyboard input, the scheduler boosts the priority of the thread that owns the window.
    - When the wait conditions for a blocked thread are satisfied, the scheduler boosts the priority of the thread. For example, when a wait operation associated with disk or keyboard I/O finishes, the thread receives a priority boost.
- **SetPriorityClass**

  Adjusts the priority class of a given process.

- **SetThreadIdealProcessor**

  Specifies the preferred processor for a specific thread. The system schedules threads on the preferred processor when possible.

- **SetThreadPriority**

  Changes the priority level for a thread. Consult the Win32 API reference for details on the different priority levels.

- **SetThreadPriorityBoost**

  Enables or disables dynamic priority boost by the system.

An example of converting UNIX thread scheduling into Windows

In this example, the thread priority level is set to the lowest level within the given policy or class for UNIX and Windows respectively. For UNIX, lowering the thread priority level requires creating an attribute object prior to instantiating the thread, and then setting the policy of the attribute object. Once this activity is complete, the thread is created with the modified attribute. Upon successfully instantiating the thread, the priority level is adjusted to the lowest level within the designated policy and class. In UNIX, this is accomplished by a call to **pthread_attr_setschedparam,** and when using Win32 by a call to **SetThreadPriority**.

UNIX example: thread scheduling

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

char message[] = "Hello I'm a Thread";
int thread_finished = 0;

void *thread_function(void *arg) {
    printf("thread_function running. Arg was %s\n", (char *)arg);
    sleep(4);
    printf("Second thread setting finished flag, and exiting now\n");
    thread_finished = 1;
    pthread_exit(NULL);
}

int main() {
    int count=0, res, min_priority, max_priority;
    struct sched_param scheduling_params;
    pthread_t a_thread;
    void *thread_result;
    pthread_attr_t thread_attr;

    res = pthread_attr_init(&thread_attr);
    if (res != 0) {
        perror("Attribute creation failed");
        exit(EXIT_FAILURE);
    }
    res = pthread_attr_setschedpolicy(&thread_attr, SCHED_OTHER);
    if (res != 0) {
        perror("Setting schedpolicy failed");
        exit(EXIT_FAILURE);
    }
    res = pthread_attr_setdetachstate(&thread_attr,
PTHREAD_CREATE_DETACHED);
    if (res != 0) {
        perror("Setting detached attribute failed");
        exit(EXIT_FAILURE);
    }
    res = pthread_create(&a_thread, &thread_attr, thread_function,
      (void *)message);
    if (res != 0) {
        perror("Thread creation failed");
        exit(EXIT_FAILURE);
    }
    max_priority = sched_get_priority_max(SCHED_OTHER);
    min_priority = sched_get_priority_min(SCHED_OTHER);
    scheduling_params.sched_priority = min_priority;
    res = pthread_attr_setschedparam(&thread_attr,
      &scheduling_params);
    if (res != 0) {
        perror("Setting schedparam failed");
        exit(EXIT_FAILURE);
    }
    (void)pthread_attr_destroy(&thread_attr);
    while(!thread_finished) {
        printf("Waiting for thread to finish (%d)\n", ++count);
        sleep(1);
    }
    printf("Other thread finished, See Ya!\n");
    exit(EXIT_SUCCESS);
}
```

Win32 example: thread scheduling

```
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>

DWORD WINAPI thread_function(PVOID arg);

char message[] = "Hello I'm a Thread";
int thread_finished = 0;

void main() {
    int count=0;
    HANDLE a_thread;
// Create a new thread.
    a_thread = CreateThread(NULL, 0, thread_function, (PVOID)message,
0, NULL);

    if (a_thread == NULL) {
        perror("Thread creation failed");
        exit(EXIT_FAILURE);
    }

  if (!SetThreadPriority(a_thread, THREAD_PRIORITY_LOWEST)) {
        perror("Setting sched priority failed");
        exit(EXIT_FAILURE);
    }

    CloseHandle(a_thread);

    while(!thread_finished) {
        printf("Waiting for thread to finished (%d)\n", ++count);
        Sleep(1000);
    }
    printf("Other thread finished, bye!\n");

    exit(EXIT_SUCCESS);
}

DWORD WINAPI thread_function(PVOID arg) {
    printf("thread_function running. Arg was %s\n", (char *)arg);
    Sleep(4000);
    printf("Second thread setting finished flag, and exiting now\n");
    thread_finished = 1;
    return 100;
}
```

In the preceding Win32 example, the priority level of the thread is adjusted to the lowest level within the priority class of the owning process. If you want to change the priority class as well as the priority level, insert the following code just before the **SetThreadPriority** call

```
SetPriorityClass(GetCurrentProcess(), PriorityClass)
```

where **PriorityClass** would have been one of the following values Table 8 summarizes how to change the scheduling priority for a thread and priority class for the owning process.

Table 8. PriorityClass values

PriorityClass	Meaning
ABOVE_NORMAL_PRIORITY_CLASS	Windows 2000 and XP: Indicates a process that has priority above NORMAL_PRIORITY_CLASS but below HIGH_PRIORITY_CLASS.
BELOW_NORMAL_PRIORITY_CLASS	Windows 2000 and XP: Indicates a process that has priority above IDLE_PRIORITY_CLASS but below NORMAL_PRIORITY_CLASS.
HIGH_PRIORITY_CLASS	Specifies this class for a process that performs time-critical tasks that must be executed immediately. The threads of the process preempt the threads of normal

	or idle priority class processes. An example is the Task List, which must respond quickly when called by the user, regardless of the load on the operating system. Use extreme care when using the high-priority class, because a high-priority class application can use nearly all available CPU time.
IDLE_PRIORITY_CLASS	Specifies this class for a process whose threads run only when the system is idle. The threads of the process are preempted by the threads of any process running in a higher priority class. An example is a screen saver. The idle-priority class is inherited by child processes.
NORMAL_PRIORITY_CLASS	Specifies this class for a process with no special scheduling needs.
REALTIME_PRIORITY_CLASS	Specifies this class for a process that has the highest possible priority. The threads of the process preempt the threads of all other processes, including the operating system processes, which may be performing important tasks. For example, a real-time process that executes for more than a very brief interval can prevent disk caches from flushing, or can cause the mouse to be unresponsive.

Managing Multiple Threads

In the next two examples, numerous threads are created that terminate at random times. Their termination and display messages are then caught to indicate their termination status.

Although this example is contrived, it does illustrate one key point: the semantics of creating multiple threads and waiting for their completion are similar in both platforms.

UNIX example: multiple threads in

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#define NUM_THREADS 5

void *thread_function(void *arg) {
    int t_number = *(int *)arg;
    int rand_delay;

    printf("thread_function running. Arg was %d\n", t_number);
//  Seed the random-number generator with current time so that
//  the numbers will be different each time function is run.
    srand( (unsigned)time(NULL));
//  random time delay from 1 to 10
    rand_delay = 1+ 9.0*(float)rand()/(float)RAND_MAX;
    sleep(rand_delay);
    printf("See Ya from thread #%d\n", t_number);
    pthread_exit(NULL);
}

int main() {
    int res;
    pthread_t a_thread[NUM_THREADS];
    void *thread_result;
    int multiple_threads;

    for(multiple_threads = 0; multiple_threads < NUM_THREADS;
      multiple_threads++) {
        res = pthread_create(&(a_thread[multiple_threads]), NULL,
          thread_function, (void *)&multiple_threads);
        if (res != 0) {
            perror("Thread creation failed");
            exit(EXIT_FAILURE);
        }
        sleep(1);
    }
    printf("Waiting for threads to finish…\n");
    for(multiple_threads = NUM_THREADS - 1; multiple_threads >= 0;
      multiple_threads--) {
        res = pthread_join(a_thread[multiple_threads],
          &thread_result);
        if (res == 0) {
            printf("Another thread\n");
        }
        else {
            perror("pthread_join failed");
        }
    }
    printf("All done\n");
    exit(EXIT_SUCCESS);
}
```

Win32 example: multiple threads in

```
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define NUM_THREADS 5

DWORD WINAPI thread_function(PVOID arg) {
    int t_number = *(int *)arg;
    int rand_delay;

    printf("thread_function running. Arg was %d\n", t_number);
//  Seed the random-number generator with current time so that
//  the numbers will be different each time function is run.
    srand((unsigned)time(NULL));
//  random time delay from 1 to 10
    rand_delay = 1 + (rand() % 10);
    Sleep(rand_delay*1000);
    printf("See Ya from thread #%d\n", t_number);
    return 100;
}

void main() {
    HANDLE a_thread[NUM_THREADS];
    int multiple_threads;

    for(multiple_threads = 0; multiple_threads < NUM_THREADS;
      multiple_threads++) {
        // Create a new thread.
        a_thread[multiple_threads] =
      CreateThread(NULL, 0, thread_function,
(PVOID)&multiple_threads, 0, NULL);

    if (a_thread[multiple_threads] == NULL) {
    perror("Thread creation failed");
    exit(EXIT_FAILURE);
    }

    Sleep(1000);
  }

    printf("Waiting for threads to finish...\n");
    for(multiple_threads = NUM_THREADS - 1; multiple_threads >= 0;
      multiple_threads--) {
    if (WaitForSingleObject(a_thread[multiple_threads], INFINITE) ==
      WAIT_OBJECT_0) {
            printf("Another thread\n");
        }
        else {
            perror("WaitForSingleObject failed");
        }
    }
    printf("All done\n");

    exit(EXIT_SUCCESS);
}
```

Fibers

A *fiber* is a lightweight thread that must be scheduled by the owning thread. Fibers exist within the context of the thread that schedules them and operate with the identity of the thread. Fibers should not be considered a replacement for a well-designed, multithreaded application. Instead, fibers should be used in situations where a design requires finely tuned scheduling, and are typically used when porting applications that require proprietary task-switching algorithms.

The primary difference between fibers and threads is that fibers are not preemptively scheduled. One key point, however, is that fibers are owned by a thread, and threads can be preempted by the task switcher. When a thread is suspended, so is the current fiber, and when a thread is resumed, so is the fiber that was active before being preempted.

Memory Management

Like UNIX, Windows has the standard heap management functions. Windows also sports functions for managing memory on a thread basis. Like many of the functional comparisons between UNIX and Windows, you can be best served by consulting guides for UNIX and Win32 programming. The basic functional mapping is covered in the next few sections.

Heap

Windows provides services similar to UNIX with respect to heap management functionality. The standard C runtime includes comparable functions for **calloc**, **malloc**, **free** and so on. It also has additional functions that may or may not be available in UNIX. The more significant added functionality is covered briefly in the following section.

Thread Local Storage

This section is a brief introduction to Thread Local Storage (TLS). For complete details, you should consult the Win32 API reference. The purpose of TLS is to define memory on a per-thread basis. The typical scenario where TLS would be used is within a dynamic-linked library (DLL), but this is not the only possible use. In the case of the DLL scenario, here are some of the details of its use:

- When a DLL attaches to a process, the DLL uses **TlsAlloc** to allocate a TLS index. The DLL then allocates some dynamic storage and uses the TLS index in a call to **TlsSetValue** to store the address in the TLS slot. This concludes the per-thread initialization for the initial thread of the process. The TLS index is stored in a global or static variable of the DLL.
- Each time the DLL attaches to a new thread of the process, the DLL allocates some dynamic storage for the new thread and uses the TLS index in a call to **TlsSetValue** to store the address in the TLS slot. This concludes the per-thread initialization for the new thread.
- Each time an initialized thread makes a DLL call requiring the data in its dynamic storage, the DLL uses the TLS index in a call to **TlsGetValue** to retrieve the address of the dynamic storage for that thread.

The functions used to manage Thread Local Storage are described below:

- **TlsAlloc**

  Allocates a thread local storage (TLS) index. A TLS index is used by a thread to store and retrieve values that are local to the thread. The minimum number of indices available to each process is defined by TLS_MINIMUM_AVAILABLE. TLS indices are not valid across process boundaries.

- **TlsFree**

  Releases a thread local storage index. This, however, does not release the data allocated and set in the TLS index slot.

- **TlsSetValue**

  Stores memory in a thread local storage index.

- **TlsGetValue**

  Returns a memory element stored in a specified thread local storage index.

Thread local storage example

The following section shows a portion of an example application. It illustrates allocation and access to a memory space on a per-thread basis. First, there is the main thread of the process that allocates a memory slot. The memory slot is then accessed and modified by a child thread. If several instances of the thread are active, each thread procedure would have a unique TLSIndex value to ensure the separation and isolation of data and state.

```
DWORD TLSIndex = 0;
DWORD WINAPI ThreadProc( LPVOID lpData)
{
  HWND hWnd = (HWND) lpData;
LPVOID lpVoid = HeapAlloc( GetProcessHeap(), 0, 128 );

  TlsSetValue( TLSIndex, lpVoid );

// Do your processing on the memory within the thread here. . .
  HeapFree( GetProcessHeap(), 0, lpVoid );

  Return(0);
}

LRESULT CALLBACK WndProc( HWND …
{
  switch( uMsg )
  {
    case WM_CREATE:
      TLSIndex = TlsAlloc();

      // Start your threads using CreateThread…
      Break;
    Case WM_DESTROY:
      TlsFree( TLSIndex );
      Break;
    Case WM_COMMAND:
    Switch( LWORD( wParam ))
    {
      case IDM_TEST:
// Do something with the TLS value by a call to TlsGetValue(DWORD)
      break;
    }
  }
}
```

Memory-Mapped Files

Windows supports memory-mapped files and memory-mapped page files. Memory-mapped page files are covered in the "Shared Memory" section as part of an exercise to port System V IPC shared memory to Windows using memory-mapped files.

Creating and using shared memory in UNIX and in Windows are conceptually the same, but syntactically different. A simple example of creating a shared memory area and mapping it in UNIX follows:

```
if ( (fd = open("/dev/zero", O_RDWR)) < 0)
  err_sys("open error");
if ( (area = mmap(0, SIZE, PROT_READ | PROT_WRITE,
MAP_SHARED, fd, 0)) == (caddr_t) -1)
  err_sys("mmap error");

close(fd);  // can close /dev/zero now that it's mapped
```

In Win32, it is coded as follows:

```
hMapObject = CreateFileMapping(
           INVALID_HANDLE_VALUE, // use paging file
           NULL,                 // no security attributes
           PAGE_READWRITE,       // read/write access
           0,                    // size: high 32-bits
           SHMEMSIZE,            // size: low 32-bits
           "dllmemfilemap");     // name of map object
if (hMapObject != NULL) {
// Get a pointer to the file-mapped shared memory.
  lpvMem = MapViewOfFile(
    hMapObject,     // object to map view of
    FILE_MAP_WRITE, // read/write access
    0,              // high offset:  map from
    0,              // low offset:   beginning
    0);             // default: map entire file
    if (lpvMem == NULL) {
      CloseHandle(hMapObject);
    }
  }
```

For details on the **CreateFileMapping** and **MapViewOfFile** functions, see the Win32 API documentation.

Shared Memory

Shared memory permits two or more threads or processes to share a region of memory. It is generally considered the most performant method of IPC since data is not copied as part of the communication process. Instead, the same physical area of memory is accessed by both the client and the server.

Windows does not support the standard System V interprocess communications mechanisms for shared memory (the shm*() APIs). It does, however, support memory-mapped files and memory-mapped page files, which you can use as an alternative to the shm*() APIs.

In Appendix A: Shared Memory, there is an example of how to port a simple UNIX application based on System V IPC to Windows, based on memory-mapped page file.

Synchronizing Access to Shared Resources

The technical challenge of using shared memory is to ensure that the server and client are not attempting to access the shared resource simultaneously. This is particularly troublesome if one or both are writing to the same-shared memory area. For example, if the server is writing to the shared memory, the client should not try to access the data until the server has completed the write operation.

To address this, several forms of synchronization are available for use in Windows. In Appendix C: Creating a Thread in Windows, the different forms of synchronization available in Windows are shown. These are:

- Semaphore
- Mutex
- Event and critical section

UNIX has two of these mechanisms, the semaphore and the semaphore, as well as an additional mechanism: file locking.

The first three mechanisms have two states: signaled and non-signaled. The synchronization object is considered busy when it is in a non-signaled state. When the object is busy, a waiting thread will block until the object switches to a signaled state. At this time, the pending thread continues executing.

The last form of synchronization is the critical section object. The critical section object is only for synchronizing threads within a single process. This synchronization mechanism only works for a single instance of the example application. While this is true, you can still consider its use as an IPC synchronization mechanism. This form of synchronization is appropriate for cases where you want to migrate your existing application from a multiprocess architecture to a single process with multithreaded architecture.

A complete Windows example of using threads, shared memory, semaphore, mutexes, and critical sections and events can be found in Appendix C: Creating a Thread in Windows.

> **Note** For applications that consume large amounts of memory and that are constrained by a lack of virtual address space, large memory support is available on Windows 2000 Advanced Server, Windows 2000 Datacenter Server, and Windows XP. A process running when using Windows normally has 2 GB of memory available in both user and system space. If the /3GB switch is inserted into Boot.ini file, Windows changes the split to give user space 3 GB and system space 1 GB. This change is a system wide option and applies to all applications run on the computer so using the /3GB switch should be analyzed for undesired side effects. See Knowledge Base article Q295443 for a sample of how to modify Boot.ini.
>
> Applications that need to control the amount of stack or heap space can use linker switches for this purpose. The default size for both the stack and heap is 1 MB. Use the /STACK option to set the size of the stack and the /HEAP option to set the heap size. Both options take the size in bytes.

Further Reading on Memory Management

A few references on memory management that you may want to acquire are:

- Solomon, David A., and Russinovich, Mark E. *Inside Microsoft Windows 2000, Third Edition*. Redmond, WA: Microsoft Press, 2000. (See Chapters 7 and 10.)
- Richter, Jeffrey. *Programming Applications for Microsoft Windows, Fourth Edition*. Redmond, WA: Microsoft Press, 1999. (See Part III, Chapters 13-18.)
- Stevens, W. Richard. *Advanced Programming in the UNIX Environment.* Reading, MA: Addison-Wesley Publishing Co., 1992.

Users, Groups and Security

The UNIX and Windows security models are quite different. Win32 uses the underlying Windows security model. This results in some key differences between the way Win32 security works and the way UNIX security works. Some of these differences have already been covered in Chapter 2 in the Comparison of Windows and UNIX Architectures section. This section covers the differences in the security model and how you should modify your code to operate in Win32.

The key areas that are addressed here are:

- A comparison of the UNIX and Win32 user and group APIs
- Adding a new group
- Adding a user to a group
- Listing groups
- Adding a user account
- Changing a user's password
- Removing a user account
- Getting user information about all users
- Getting information about a specific user
- Retrieving the current user's user name
- Security functions

UNIX and Win32 User and Group Functions

This section describes the different user and group functions for UNIX and Win32.

The UNIX user and group functions

Table 9 shows user and group management functions that control user and group accounts in a security database. To link some of the code examples that use these functions, you must add **-lcrypt** to the **gcc** option list.

Table 9. UNIX User and group functions—security

Function	Description
Group database functions	
endgrent	Close database
fgetgrent, fgetgrent_r	Get next Group database entry from FILE Stream
getgrent, getgrent_r	Get next Group database entry
getgrgid, getgrgid_r	Get Group database entry with Group ID
getgrnam, getgrnam_r	Get Group database entry with Name
setgrent	Rewind database
Supplementary group access list functions	
getgroups	Get
initgroups	Initialize
setgroups	Set
User "shadow" database functions	
endspent	Close database
fgetspent, fgetspent_r	Get next User "Shadow" database entry from FILE Stream
getspent, getspent_r	Get next User "Shadow" database entry
getspnam, getspnam_r	Get User "Shadow" database entry with Name
setspent	Rewind database
User database functions	
endpwent	Close database
fgetpwent, fgetpwent_r	Get next User database entry from FILE Stream
getpw	User database "get" function to get passwd entry from UID

getpwent, getpwent_r	Get next User database entry
getpwnam, getpwnam_r	Get User database entry with Name
getpwuid, getpwuid_r	Get User database entry with User ID
setpwent	Rewind database
User database "Lock/UnLock" functions	
lckpwdf	"Lock" function
ulckpwdf	"UnLock" function
User database "Write" functions	
putgrent	Write group file entry
putpwent	Write password file entry
putspent	Write shadow password file entry

The Win32 user functions

Table 10 shows Win32 user management functions that control a user's account in a security database. To link these functions in the example code later in this section, you must add **Netapi32.lib** to the Visual Studio project link-library list.

Table 10. Win32 User and Group functions—security

Function	Description
NetUserAdd	Adds a user account and assigns a password and privilege level.
NetUserChangePassword	Changes a user's password for a specified network server or domain.
NetUserDel	Deletes a user account from the server.
NetUserEnum	Lists all user accounts on a server.
NetUserGetGroups	Returns a list of global group names to which a user belongs.
NetUserGetInfo	Returns information about a particular user account on a server.
NetUserGetLocalGroups	Returns a list of local group names to which a user belongs.
NetUserSetGroups	Sets global group memberships for a specified user account.
NetUserSetInfo	Sets the password and other elements of a user account.

User account information is available at the following levels:

- USER_INFO_0
- USER_INFO_1
- USER_INFO_2
- USER_INFO_3
- USER_INFO_4
- USER_INFO_10
- USER_INFO_11
- USER_INFO_20

- USER_INFO_21
- USER_INFO_22
- USER_INFO_23

In addition, the following information levels are valid when you call the NetUserSetInfo (..\..\..\Originals\mk:@MSITStore:\\dimension\cdrive\Program Files\Microsoft Visual Studio\MSDN\2001OCT\1033\NETMGMT.CHM::\hh\network\ntlmapi2_0zfz.htm) function:

- USER_INFO_1003
- USER_INFO_1005
- USER_INFO_1006
- USER_INFO_1007
- USER_INFO_1008
- USER_INFO_1009
- USER_INFO_1010
- USER_INFO_1011
- USER_INFO_1012
- USER_INFO_1014
- USER_INFO_1017
- USER_INFO_1020
- USER_INFO_1024
- USER_INFO_1051
- USER_INFO_1052
- USER_INFO_1053

The Win32 group functions

The network management provides functions for both local groups and global groups.

Local group

A *local group* can contain user accounts or global group accounts from one or more domains. (Global groups can contain users from only one domain.) A local group shares common privileges and rights only within its own domain.

The network management local group functions control members of local groups in a way that the functions can only be called locally on the system on which the local group is defined. On a Windows NT, Windows 2000 or Windows XP workstation, or on a server that is not a domain controller, you can use only a local group defined on that system. A local group defined on the primary domain controller is replicated to all other domain controllers in the domain. Therefore, a local group is available on all domain controllers within the domain in which it was created.

The local group functions create or delete local groups, and review or adjust the memberships of local groups. These functions are shown in Table 11.

Table 11. Win32 local group functions

Function	Description
NetLocalGroupAdd	Creates a local group.
NetLocalGroupAddMembers	Adds one or more users or global groups to an

	existing local group.
NetLocalGroupDel	Deletes a local group, removing all existing members from the group.
NetLocalGroupDelMembers	Removes one or more members from an existing local group
NetLocalGroupEnum	Returns information about each local group account on a server.
NetLocalGroupGetInfo	Returns information about a particular local group account on a server.
NetLocalGroupGetMembers	Lists all members of a specified local group.
NetLocalGroupSetInfo	Sets general information about a local group.
NetLocalGroupSetMembers	Assigns members to a local group.

Global group

A *global group* contains user accounts from one domain that are grouped together under one group account name. A global group can contain only members (users) from the domain where the global group is created; it cannot contain local groups or other global groups. A global group is available within its own domain and within any trusting domain.

The network management group functions control global groups. Table 12 shows these group functions.

Table 12. Win32 network management (global group) functions

Function	Description
NetGroupAdd	Creates a global group.
NetGroupAddUser	Adds one user to an existing global group.
NetGroupDel	Removes a global group whether or not the group has any members.
NetGroupDelUser	Removes one user name from a global group.
NetGroupEnum	Lists all global groups on a server.
NetGroupGetInfo	Returns information about a particular global group.
NetGroupGetUsers	Lists all members of a particular global group.
NetGroupSetInfo	Sets general information about a global group.
NetGroupSetUsers	Assigns members to a new global group. Replaces the members of an existing group.

Global group account information is available at the following levels:

- GROUP_INFO_0
- GROUP_INFO_1
- GROUP_INFO_2
- GROUP_INFO_3
- GROUP_INFO_1002
- GROUP_INFO_1005

Adding a New Group

The following examples illustrate migrating code from UNIX to Windows to add a new group.

UNIX example: adding a new group

This example uses the **getgrnam** and **getgruid** functions to verify that a group account and gid does not already exists before adding a new group account with the **putgrent** function.

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <grp.h>

int main(int argc, char *argv[])
{
    struct group a_group;
    struct group *ptr_group;

    char gr_name[20];
    char gr_passwd[] = "";
    char *gr_mem[] = {0};

    int i, gid;

    a_group.gr_name = gr_name;
    a_group.gr_passwd = gr_passwd;
    a_group.gr_mem = gr_mem;

    if (argc != 3)
    {
       printf("Usage: %s GroupName GID\n", argv[0]);
       exit(1);
    }

// Call the getpwnam function to check if user exists.
    if(ptr_group = getgrnam (argv[1])) {
        printf("Group already exists\n");
        exit(1);
    }

// Call the getpwuid function to check if User ID exists
    if(ptr_group = getgrgid (gid=(gid_t) atoi(argv[2]))) {
        printf("Group ID already exists\n");
        exit(1);
    }

    ptr_group = &a_group;

    strcpy (ptr_group->gr_name, argv[1]);
    printf ("Group: %s\n", ptr_group->gr_name);

    ptr_group->gr_gid = gid;
    printf ("gid: %d\n", ptr_group->gr_gid);


// Call the putgrent function
    putgrent(ptr_group, fopen ("/etc/group", "a+"));

    exit(0);
}
```

Win32 example: adding a new group

This example uses the **NetLocalGroupAdd()** function to create a local group in the security database.

```
#ifndef UNICODE
#define UNICODE
#endif

#include <stdio.h>
#include <windows.h>
#include <lm.h>

int wmain(int argc, wchar_t *argv[])
{
   LOCALGROUP_INFO_1 lgi;
   DWORD dwLevel = 1;
   DWORD dwError = 0;
   NET_API_STATUS nStatus;

   if (argc != 3)
   {
      fwprintf(stderr, L"Usage: %s \\\\ServerName GroupName\n",
argv[0]);
      exit(1);
   }

   // Set up the LOCALGROUP_INFO_1 structure.

   lgi.lgrpi1_name = argv[2];
   lgi.lgrpi1_comment = NULL;

   nStatus = NetLocalGroupAdd(argv[1],
                              dwLevel,
                              (LPBYTE)&lgi,
                              &dwError);
   //
   // If the call succeeds, inform the user.
   //
   if (nStatus == NERR_Success)
      fwprintf(stderr, L"Local Group %s has been successfully added
        on %s\n",
               argv[2], argv[1]);
   //
   // Otherwise, print the system error.
   //
   else
      fprintf(stderr, "A system error has occurred: %d\n", nStatus);

   return 0;
}
```

Adding a User to a Group

The following examples illustrate migrating code from UNIX to Windows to add a user to a group.

UNIX example: adding a user to a group

This example uses the **getgrnam** and **getpwnam** functions to verify that the specified group account exists, and that the user account to be added exists, before creating a new /etc/group file (named: /etc/groupx) with the added member to the specified group using a combination of the **fgetgrent** and **fprintf** functions. This program assumes that the super user will be using this program, and will "manually" copy /etc/groupx to /etc/group after verifying proper update to the group entry.

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <grp.h>
#include <pwd.h>

int main(int argc, char *argv[])
{
    struct group a_group;
    struct group *ptr_group;
    struct passwd *ptr_pswd;

    int i;
    FILE *stream_in, *stream_out;

    if (argc != 3)
    {
       printf("Usage: %s GroupName NewMember\n", argv[0]);
       exit(1);
    }

// Call the getgrnam function to check if group exists.
    if(!(ptr_group = getgrnam (argv[1]))) {
        printf("group does not exist\n");
        exit(1);
    }
// Call the getpwnam function to check if group exists.
    if(!(ptr_pswd = getpwnam (argv[2]))) {
        printf("member to be added does not exist\n");
        exit(1);
    }

    ptr_group = &a_group;

// Scan /etc/group file, create /etc/groupx, with updated group entry
    stream_in = fopen ("/etc/group", "r");
    stream_out = fopen ("/etc/groupx", "w");

    while(ptr_group = fgetgrent(stream_in)) {
  i=0;
        fprintf(stream_out, "%s:%s:%d:", ptr_group->gr_name,\
                ptr_group->gr_passwd, ptr_group->gr_gid);
        while(ptr_group->gr_mem[i] != 0) {
            fprintf(stream_out, "%s,", ptr_group->gr_mem[i]);
            i++;
        }
        if(strcmp(ptr_group->gr_name, argv[1]) == 0)
            fprintf(stream_out, "%s\n", argv[2]);
        else
            fprintf(stream_out, "\n");
        printf("Next Group: %s\n", ptr_group->gr_name);
    }

    fclose (stream_in);
    fclose (stream_out);

    exit(0);
}
```

Win32 example: adding a user to a group

This example uses the **NetLocalGroupAddMembers ()** function to add members to a local group in the security database.

```
#ifndef UNICODE
#define UNICODE
#endif

#include <stdio.h>
#include <windows.h>
#include <lm.h>

int wmain(int argc, wchar_t *argv[])
{
   LOCALGROUP_MEMBERS_INFO_3    lgmi;
   DWORD dwLevel = 3;
   DWORD dwCount = 1;
   NET_API_STATUS nStatus;

   if (argc != 4)
   {
      fwprintf(stderr, L"Usage: %s \\\\ServerName GroupName
        UserName\n", argv[0]);
      exit(1);
   }

   // Set up the LOCALGROUP_MEMBERS_INFO_3 structure.

   lgmi.lgrmi3_domainandname = argv[3];


   nStatus = NetLocalGroupAddMembers(argv[1],
                                     argv[2],
                                     dwLevel,
                                     (LPBYTE)&lgmi,
                                     dwCount);
   //
   // If the call succeeds, inform the user.
   //
   if (nStatus == NERR_Success)
      fwprintf(stderr, L"User %s has been successfully added to Local
        Group %s on %s\n",
               argv[3], argv[2], argv[1]);
   //
   // Otherwise, print the system error.
   //
   else
      fprintf(stderr, "A system error has occurred: %d\n", nStatus);
   return 0;
}
```

Listing Groups

The following examples illustrate migrating code to list groups from UNIX to Windows.

UNIX example: listing groups

This example uses the **getgroups** function to retrieve a list of global groups to which the calling user belongs.

```
#include <sys/types.h>
#include <unistd.h>
#include <grp.h>

int main()
{
   struct group *ptr_group;
   gid_t grouplist[100];
   int num;

   num = getgroups (99, grouplist);

   printf("group\tID:\n---------------\n");
   if (num > 0)
      while(num--) {
         ptr_group = getgrgid(grouplist[num]);
         printf("%s\t%d\n", ptr_group->gr_name, grouplist[num]);
      }
   else
      printf("num=%d\n", num);

   exit(0);
}
```

Win32 example: listing groups

This example uses the **NetUserGetGroups()** function to retrieve a list of global groups to which a specified user belongs. You use the **NetUserGetLocalGroups()** function to get a list of local groups to which a user belongs.

```
#ifndef UNICODE
#define UNICODE
#endif

#include <stdio.h>
#include <assert.h>
#include <windows.h>
#include <lm.h>

int wmain(int argc, wchar_t *argv[])
{
   LPGROUP_USERS_INFO_0 pBuf = NULL;
   DWORD dwLevel = 0;
   DWORD dwPrefMaxLen = -1;
   DWORD dwEntriesRead = 0;
   DWORD dwTotalEntries = 0;
   NET_API_STATUS nStatus;

   if (argc != 3)
   {
      fwprintf(stderr, L"Usage: %s \\\\ServerName UserName\n",
argv[0]);
      exit(1);
   }
   //
   // Call the NetUserGetGroups function, specifying level 0.
   //
   nStatus = NetUserGetGroups(argv[1],
                              argv[2],
                              dwLevel,
                              (LPBYTE*)&pBuf,
                              dwPrefMaxLen,
                              &dwEntriesRead,
                              &dwTotalEntries);
   //
   // If the call succeeds,
   //
   if (nStatus == NERR_Success)
   {
      LPGROUP_USERS_INFO_0 pTmpBuf;
      DWORD i;
      DWORD dwTotalCount = 0;

      if ((pTmpBuf = pBuf) != NULL)
      {
         fprintf(stderr, "\nGlobal group(s):\n");
         //
         // Loop through the entries;
         //  print the name of the global groups
         //  to which the user belongs.
         //
         for (i = 0; i < dwEntriesRead; i++)
         {
            assert(pTmpBuf != NULL);

            if (pTmpBuf == NULL)
            {
               fprintf(stderr, "An access violation has occurred\n");
               break;
            }

            wprintf(L"\t-- %s\n", pTmpBuf->grui0_name);

            pTmpBuf++;
            dwTotalCount++;
         }
      }
      //
      // If all available entries were
      //  not enumerated, print the number actually
      //  enumerated and the total number available.
      //
      if (dwEntriesRead < dwTotalEntries)
```

```
         fprintf(stderr, "\nTotal entries: %d", dwTotalEntries);
      //
      // Otherwise, just print the total.
      //
      printf("\nEntries enumerated: %d\n", dwTotalCount);
   }
   else
      fprintf(stderr, "A system error has occurred: %d\n", nStatus);
   //
   // Free the allocated buffer.
   //
   if (pBuf != NULL)
      NetApiBufferFree(pBuf);

   return 0;
}
```

Adding a User Account

The following examples illustrate migrating code to add user accounts from UNIX to Windows.

UNIX example: adding a user account

This example uses the **getpwnam** and **getpwuid** functions to verify that a user account and **uid** does not already exist before adding a new user account with the **putpwent** function. This program assumes that a user home directory with the same name has already been created under /home, and does not verify that this directory exists.

```
#define _XOPEN_SOURCE
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <pwd.h>

int main(int argc, char *argv[])
{
    struct passwd pswd;
    struct passwd *ptr_pswd;

    char pw_name[20];
    char pw_passwd[30];
    char pw_dir[26];
    char pw_shell[15];

    int uid;

    pswd.pw_name = pw_name;
    pswd.pw_passwd = pw_passwd;
    pswd.pw_dir = pw_dir;
    pswd.pw_shell = pw_shell;

    if (argc != 5)
    {
       printf("Usage: %s UserName Password UID GID\n", argv[0]);
       exit(1);
    }
// Call the getpwnam function to check if user exists.
    if(ptr_pswd = getpwnam (argv[1])) {
        printf("user already exists\n");
        exit(1);
    }

// Call the getpwuid function to check if User ID exists
    if(ptr_pswd = getpwuid (uid=(uid_t) atoi(argv[3]))) {
        printf("user ID already exists\n");
        exit(1);
    }

    ptr_pswd = &pswd;

    strcpy (ptr_pswd->pw_name, argv[1]);
    printf ("Name: %s\n", ptr_pswd->pw_name);

    strcpy (ptr_pswd->pw_passwd, crypt(argv[2],"az"));

    ptr_pswd->pw_uid = uid;
    printf ("uid: %d\n", ptr_pswd->pw_uid);

    ptr_pswd->pw_gid = (gid_t) atoi(argv[4]);
    printf ("gid: %d\n", ptr_pswd->pw_gid);

    strcpy (ptr_pswd->pw_dir, "/home/");
    strcat (ptr_pswd->pw_dir, argv[1]);
    printf ("Home Dir: %s\n", ptr_pswd->pw_dir);

    strcpy (ptr_pswd->pw_shell, "/bin/bash");
    printf ("Shell: %s\n", ptr_pswd->pw_shell);

// Call the putpwent function
    putpwent(ptr_pswd, fopen ("/etc/passwd", "a+"));

    exit(0);
}
```

Win32 example: adding a user account

This example uses **NetUserAdd()** function to add a user account and assigns a password and privilege level.

```
#ifndef UNICODE
#define UNICODE
#endif

#include <stdio.h>
#include <windows.h>
#include <lm.h>

int wmain(int argc, wchar_t *argv[])
{
   USER_INFO_1 ui;
   DWORD dwLevel = 1;
   DWORD dwError = 0;
   NET_API_STATUS nStatus;

   if (argc != 3)
   {
      fwprintf(stderr, L"Usage: %s \\\\ServerName UserName\n",
argv[0]);
      exit(1);
   }
   //
   // Set up the USER_INFO_1 structure.
   //  USER_PRIV_USER: name identifies a user,
   //    rather than an administrator or a guest.
   //  UF_SCRIPT: required for LAN Manager 2.0 and
   //    Windows NT and later.
   //
   ui.usri1_name = argv[2];
   ui.usri1_password = argv[2];
   ui.usri1_priv = USER_PRIV_USER;
   ui.usri1_home_dir = NULL;
   ui.usri1_comment = NULL;
   ui.usri1_flags = UF_SCRIPT;
   ui.usri1_script_path = NULL;
   //
   // Call the NetUserAdd function, specifying level 1.
   //
   nStatus = NetUserAdd(argv[1],
                        dwLevel,
                        (LPBYTE)&ui,
                        &dwError);
   //
   // If the call succeeds, inform the user.
   //
   if (nStatus == NERR_Success)
      fwprintf(stderr, L"User %s has been successfully added on
%s\n",
               argv[2], argv[1]);
   //
   // Otherwise, print the system error.
   //
   else
      fprintf(stderr, "A system error has occurred: %d\n", nStatus);

   return 0;
}
```

Changing a User's Password

The following examples illustrate migrating code to change a user's password from UNIX to Windows.

UNIX example: changing a user's password

This example uses the **getpwnam** function to verify that the user account exists before creating a new /etc/passwd file (named /etc/passwdx) with the entered password using a combination of the **fgetpwent** and **fprintf** functions. This program assumes that the super user will be using this program, and will "manually" copy /etc/passwdx to /etc/passwd after verifying proper update to the user entries.

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <pwd.h>
```

```
int main(int argc, char *argv[])
{
    struct passwd pswd;
    struct passwd *ptr_pswd;

    FILE *stream_in, *stream_out;

    if (argc != 3)
    {
       printf("Usage: %s UserName Password\n", argv[0]);
       exit(1);
    }

// Call the getpwnam function to check if user exists.
    if(!(ptr_pswd = getpwnam (argv[1]))) {
        printf("user does not exist\n");
        exit(1);
    }

    ptr_pswd = &pswd;

// Scan /etc/passwd file, create /etc/passwdx, with updated user
  entry
    stream_in = fopen ("/etc/passwd", "r");
    stream_out = fopen ("/etc/passwdx", "w");

    while(ptr_pswd = fgetpwent(stream_in)) {
        if(strcmp(ptr_pswd->pw_name, argv[1]) == 0)
            strcpy (ptr_pswd->pw_passwd, crypt(argv[2],"az"));
            fprintf(stream_out, "%s:%s:%d:%d:%s:%s:%s\n", ptr_pswd-
              >pw_name,\
                  ptr_pswd->pw_passwd, ptr_pswd->pw_uid, ptr_pswd-
                    >pw_gid,\
                  ptr_pswd->pw_gecos, ptr_pswd->pw_dir, ptr_pswd-
                    >pw_shell);
        printf("Next Name: %s\n", ptr_pswd->pw_name);
    }

    fclose (stream_in);
    fclose (stream_out);

    exit(0);
}
```

Win32 example: changing a user's password

This example uses the **NetUserChangePassword()** function to change a user's password for a specified network server or domain.

```
#ifndef UNICODE
#define UNICODE
#endif

#include <stdio.h>
#include <windows.h>
#include <lm.h>

int wmain(int argc, wchar_t *argv[])
```

```
{
   DWORD dwError = 0;
   NET_API_STATUS nStatus;
   //
   // All parameters are required.
   //
   if (argc != 5)
   {
      fwprintf(stderr, L"Usage: %s \\\\ServerName UserName
        OldPassword NewPassword\n", argv[0]);
      exit(1);
   }
   //
   // Call the NetUserChangePassword function.
   //
   nStatus = NetUserChangePassword(argv[1], argv[2], argv[3],
argv[4]);
   //
   // If the call succeeds, inform the user.
   //
   if (nStatus == NERR_Success)
      fwprintf(stderr, L"User password has been changed
        successfully\n");
   //
   // Otherwise, print the system error.
   //
   else
      fprintf(stderr, "A system error has occurred: %d\n", nStatus);

   return 0;
}
```

Removing a User Account

The following examples illustrate migrating code to remove user accounts from UNIX to Windows.

UNIX example: removing a user account

This example uses the **getpwnam** function to verify that the user account exists before creating a new /etc/passwd file (named /etc/passwdx). It does this with the specified user account removed, using a combination of the **fgetpwent** and **fprintf** functions. This program assumes that the super user will be using this program, and will "manually" copy /etc/passwdx to /etc/passwd after verifying proper update to the user entries.

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <pwd.h>

int main(int argc, char *argv[])
{
    struct passwd pswd;
    struct passwd *ptr_pswd;

    FILE *stream_in, *stream_out;
    if (argc != 2)
    {
       printf("Usage: %s UserName\n", argv[0]);
       exit(1);
    }

// Call the getpwnam function to check if user exists.
    if(!(ptr_pswd = getpwnam (argv[1]))) {
        printf("user does not exist\n");
        exit(1);
    }

    ptr_pswd = &pswd;

// Scan /etc/passwd file, create /etc/passwdx, with updated user
entry
    stream_in = fopen ("/etc/passwd", "r");
    stream_out = fopen ("/etc/passwdx", "w");

    while(ptr_pswd = fgetpwent(stream_in)) {
        if(strcmp(ptr_pswd->pw_name, argv[1]) != 0)
            fprintf(stream_out, "%s:%s:%d:%d:%s:%s:%s\n", ptr_pswd-
              >pw_name,\
                    ptr_pswd->pw_passwd, ptr_pswd->pw_uid, ptr_pswd-
                      >pw_gid,\
                    ptr_pswd->pw_gecos, ptr_pswd->pw_dir, ptr_pswd-
                      >pw_shell);
        printf("Next Name: %s\n", ptr_pswd->pw_name);
    }

    fclose (stream_in);
    fclose (stream_out);

    exit(0);
}
```

Win32 example: removing a user account

This example uses the **NetUserDel()** function to delete a user account from a server.

```
#ifndef UNICODE
#define UNICODE
#endif

#include <stdio.h>
#include <windows.h>
#include <lm.h>

int wmain(int argc, wchar_t *argv[])
{
   DWORD dwError = 0;
   NET_API_STATUS nStatus;
   //
   // All parameters are required.
   //
   if (argc != 3)
   {
      fwprintf(stderr, L"Usage: %s \\\\ServerName UserName\n",
argv[0]);
```

```
      exit(1);
   }
   //
   // Call the NetUserDel function to delete the share.
   //
   nStatus = NetUserDel(argv[1], argv[2]);
   //
   // Display the result of the call.
   //
   if (nStatus == NERR_Success)
      fwprintf(stderr, L"User %s has been successfully deleted on 
%s\n",
                argv[2], argv[1]);
   else
      fprintf(stderr, "A system error has occurred: %d\n", nStatus);

   return 0;
}
```

Getting User Information About All Users

The following examples illustrate migrating code to get user information from UNIX to Windows.

UNIX example: getting user information about all users

This example uses the **getpwent** function to provide information about all available user accounts.

```
#include <stdio.h>
#include <unistd.h>
#include <pwd.h>

int main()
{
    struct passwd pswd;
    struct passwd *ptr_pswd;

// Scan /etc/passwd file
    while(ptr_pswd = getpwent())
        printf("Account Name: %s\n", ptr_pswd->pw_name);

    exit(0);
}
```

Win32 example: getting user information about all users

This example uses the **NetUserEnum()** function to provide information about all the users on a server.

```
#ifndef UNICODE
#define UNICODE
#endif

#include <stdio.h>
#include <assert.h>
#include <windows.h>
#include <lm.h>

int wmain(int argc, wchar_t *argv[])
```

```
{
   LPUSER_INFO_0 pBuf = NULL;
   LPUSER_INFO_0 pTmpBuf;
   DWORD dwLevel = 0;
   DWORD dwPrefMaxLen = -1;
   DWORD dwEntriesRead = 0;
   DWORD dwTotalEntries = 0;
   DWORD dwResumeHandle = 0;
   DWORD i;
   DWORD dwTotalCount = 0;
   NET_API_STATUS nStatus;
   LPTSTR pszServerName = NULL;

   if (argc > 2)
   {
      fwprintf(stderr, L"Usage: %s [\\\\ServerName]\n", argv[0]);
      exit(1);
   }
   // The server is not the default local computer.
   //
   if (argc == 2)
      pszServerName = argv[1];
   wprintf(L"\nUser account on %s: \n", pszServerName);
   //
   // Call the NetUserEnum function, specifying level 0;
   //   enumerate global user account types only.
   //
   do // begin do
   {
      nStatus = NetUserEnum(pszServerName,
                            dwLevel,
                            FILTER_NORMAL_ACCOUNT, // global users
                            (LPBYTE*)&pBuf,
                            dwPrefMaxLen,
                            &dwEntriesRead,
                            &dwTotalEntries,
                            &dwResumeHandle);
      //
      // If the call succeeds,
      //
      if ((nStatus == NERR_Success) || (nStatus == ERROR_MORE_DATA))
      {
         if ((pTmpBuf = pBuf) != NULL)
         {
            //
            // Loop through the entries.
            //
            for (i = 0; (i < dwEntriesRead); i++)
            {
               assert(pTmpBuf != NULL);

               if (pTmpBuf == NULL)
               {
                  fprintf(stderr, "An access violation has
                    occurred\n");
                  break;
               }
               //
               //  Print the name of the user account.
               //
               wprintf(L"\t-- %s\n", pTmpBuf->usri0_name);

               pTmpBuf++;
               dwTotalCount++;
            }
         }
      }
      //
      // Otherwise, print the system error.
      //
      else
         fprintf(stderr, "A system error has occurred: %d\n",
nStatus);
      //
      // Free the allocated buffer.
      //
      if (pBuf != NULL)
      {
         NetApiBufferFree(pBuf);
         pBuf = NULL;
      }
   }
   // Continue to call NetUserEnum while
   //  there are more entries.
   //
   while (nStatus == ERROR_MORE_DATA); // end do
   //
   // Check again for allocated memory.
   //
   if (pBuf != NULL)
      NetApiBufferFree(pBuf);
   //
   // Print the final count of users enumerated.
   //
   fprintf(stderr, "\nTotal of %d entries enumerated\n",
dwTotalCount);

   return 0;
}
```

Getting Information About a Specific User

The following examples illustrate migrating code from UNIX to Windows to get information about a specific user account.

UNIX example: getting information about a specific user

This example uses the **getpwnam** function to obtain the information and output the account information of a specified user.

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <pwd.h>

int main(int argc, char *argv[])
{
    struct passwd *ptr_pswd;

    int uid;
    if (argc != 2)
    {
       printf("Usage: %s UserName\n", argv[0]);
       exit(1);
    }

// Call the getpwnam function to check if user exists.
    if(!(ptr_pswd = getpwnam (argv[1]))) {
        printf("user does not exist\n");
        exit(1);
    }

    printf ("Name: %s\n", ptr_pswd->pw_name);
    printf ("uid: %d\n", ptr_pswd->pw_uid);
    printf ("gid: %d\n", ptr_pswd->pw_gid);
    printf ("Home Dir: %s\n", ptr_pswd->pw_dir);

    printf ("Shell: %s\n", ptr_pswd->pw_shell);

    exit(0);
}
```

Win32 example: getting information about a specific user

This example uses **NetUserGetInfo()** function to retrieve information about a particular user account on a server. This function returns specific information in USER INFO structure based on different levels. Please refer to the platform SDK documentation for more details.

```
#ifndef UNICODE
#define UNICODE
#endif

#include <stdio.h>
#include <windows.h>
#include <lm.h>

int wmain(int argc, wchar_t *argv[])
{
   DWORD dwLevel = 10;
   LPUSER_INFO_10 pBuf = NULL;
   NET_API_STATUS nStatus;

   if (argc != 3)
   {
      fwprintf(stderr, L"Usage: %s \\\\ServerName UserName\n",
argv[0]);
      exit(1);
   }
   //
   // Call the NetUserGetInfo function; specify level 10.
   //
   nStatus = NetUserGetInfo(argv[1],
                            argv[2],
                            dwLevel,
                            (LPBYTE *)&pBuf);
   //
   // If the call succeeds, print the user information.
   //
   if (nStatus == NERR_Success)
   {
    if (pBuf != NULL)
      {
         wprintf(L"\n\tAccount:      %s\n", pBuf->usri10_name);
         wprintf(L"\tComment:      %s\n", pBuf->usri10_comment);
         wprintf(L"\tUser comment: %s\n", pBuf->usri10_usr_comment);
         wprintf(L"\tFull name:    %s\n", pBuf->usri10_full_name);
      }
   }
   // Otherwise, print the system error.
   //
   else
      fprintf(stderr, "A system error has occurred: %d\n", nStatus);
   //
   // Free the allocated memory.
   //
   if (pBuf != NULL)
      NetApiBufferFree(pBuf);

   return 0;
}
```

Retrieving the Current User's User Name

The following examples illustrate migrating code from UNIX to Windows to retrieve the current user's user name.

UNIX example: retrieving the current user's username

This example uses the **getlogin()** function to retrieve the user name of the user currently logged onto the system.

```
#include <stdio.h>
#include <unistd.h>

main()
{

// Get and display the user name.
    printf("User name: %s\n", getlogin());

}
```

Win32 example: retrieving the current user's user name

This example uses **GetUserName()** Function to retrieve the user name of the current thread. This is the name of the user currently logged on to the system. The **GetUserNameEx()** function can be used to retrieve the user name in a specified format.

```
#include <windows.h>
#include <stdio.h>
#include <lmcons.h>
```

```
void main()
{
    LPTSTR lpszSystemInfo;          // pointer to system information
string
    DWORD cchBuff = 256;            // size of user name
    TCHAR tchBuffer[UNLEN + 1];     // buffer for expanded string

    lpszSystemInfo = tchBuffer;

    // Get and display the user name.
    GetUserName(lpszSystemInfo, &cchBuff);

    printf("User name: %s\n", lpszSystemInfo);
}
```

Security Functions

Windows security enables you to control the ability of a process to access securable objects or to perform various system administration tasks. Application developers use access control to control access to securable objects such as files, registry keys and directory service objects.

Table 13 shows the two basic components of the Windows access-control model.

Table 13. Windows access-control model components

Access-control component	Description
Access tokens	Access tokens contain information about a logged-on user.
Security descriptors	Security descriptors contain the security information that protects a securable object.

When a user logs on to a Windows system, the system authenticates the user's account name and password. If the log on is successful, the system creates an access token. Every process executed on behalf of this user will have a copy of this access token. The access token contains security identifiers (SIDs) that identify the user's account and any group accounts to which the user belongs. The token also contains a list of the privileges held by the user or the user's groups. The system uses this token to identify the associated user when a process tries to access a securable object or perform a system administration task that requires privileges.

An access-control list (ACL) is a list of access-control entries (ACEs). Each ACE in an ACL identifies a trustee and specifies the access rights allowed, denied or audited for that trustee. The security descriptor for a securable object can contain two ACLs: a DACL and a SACL.

A DACL (discretionary access-control list) identifies the trustees that are allowed or denied access to a securable object. When a process tries to access a securable object, the system checks the ACEs in the object's DACL to determine whether to grant access to it. If the object does not have a DACL, the system grants full access to everyone. If the object's DACL has no ACEs, the system denies all attempts to access the object because the DACL does not allow any access rights. The system checks the ACEs in sequence until it finds one or more ACEs that allow all the requested access rights, or until any of the requested access rights are denied.

A system access-control list (SACL) enables administrators to log attempts to access a secured object. Each ACE specifies the types of access attempts by a specified trustee that cause the system to generate a record in the security event log. An ACE in a SACL can generate audit records when an access attempt fails, when it succeeds or both. In future releases, a SACL can also raise an alarm when an unauthorized user attempts to gain access to an object.

Table 14 shows the functions that are used with access tokens.

Table 14. Access token functions

Access token	**Description**
TBD	

UNIX example: using security functions

The following examples illustrate migrating code from UNIX to Windows that uses basic security functions.

```
#include <stdio.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <err.h>
#include <pwd.h>

int main()
{
    struct passwd pswd;
    struct stat stat_info;
    uid_t uid;
    int rtn;
    struct passwd * ptr_pswd = &pswd;

// Get the owner UID of the file.
// and Check error code.
    if (stat ("myfile.txt", &stat_info) == -1)
        err(1, NULL);

    uid = stat_info.st_uid;

// Lookup Account UID to get the owner's name.
// and Check error code.
    if(ptr_pswd = getpwuid (uid))
// Print the account name.
        printf("owner: %s\n", ptr_pswd->pw_name);
    else
        err(1, NULL);
}
```

Win32 example: using security functions

This example uses the **GetSecurityInfo()** and **LookupAccountSid()** functions to find and print the name of the owner of a file. The file exists in the current working directory on the local server.

```
#include <stdio.h>
```

```
#include <windows.h>
#include <tchar.h>
#include "accctrl.h"
#include "aclapi.h"

int main(int argc, char **argv)
{
    DWORD dwRtnCode = 0;
    PSID pSidOwner;
    BOOL bRtnBool = TRUE;
    LPTSTR AcctName, DomainName;
    DWORD dwAcctName = 1, dwDomainName = 1;
    SID_NAME_USE eUse = SidTypeUnknown;
    HANDLE hFile;
    PSECURITY_DESCRIPTOR pSD;
    LPVOID lpMsgBuf;

    // Get the handle of the file object.
    hFile = CreateFile(
                    "myfile.txt",
                    GENERIC_READ,
                    FILE_SHARE_READ,
                    NULL,
                    OPEN_EXISTING,
                    FILE_ATTRIBUTE_NORMAL,
                    NULL);

    // Check GetLastError for CreateFile error code.
    if (hFile == INVALID_HANDLE_VALUE) {
        FormatMessage(
            FORMAT_MESSAGE_ALLOCATE_BUFFER |
            FORMAT_MESSAGE_FROM_SYSTEM |
            FORMAT_MESSAGE_IGNORE_INSERTS,
            NULL,
            GetLastError(),
            MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), // Default language
            (LPTSTR) &lpMsgBuf,
            0,
            NULL
        );

        _tprintf(TEXT("Error message:%s\n"), lpMsgBuf);
        LocalFree(lpMsgBuf);

        return -1;
    }

    // Allocate memory for the SID structure.
    pSidOwner = (PSID)GlobalAlloc(
              GMEM_FIXED,
              sizeof(PSID));

    // Allocate memory for the security descriptor structure.
    pSD = (PSECURITY_DESCRIPTOR)GlobalAlloc(
              GMEM_FIXED,
              sizeof(PSECURITY_DESCRIPTOR));

    // Get the owner SID of the file.
    dwRtnCode = GetSecurityInfo(
                    hFile,
                      SE_FILE_OBJECT,
                      OWNER_SECURITY_INFORMATION,
                      &pSidOwner,
                      NULL,
                      NULL,
                      NULL,
                      &pSD);

    // Check GetLastError for GetSecurityInfo error condition.
    if (dwRtnCode != ERROR_SUCCESS) {
              DWORD dwErrorCode = 0;

              dwErrorCode = GetLastError();
              _tprintf(TEXT("GetSecurityInfo error = %d\n"),
                dwErrorCode);
              return -1;
    }

    // First call to LookupAccountSid to get the buffer sizes.
    DomainName = NULL;
    AcctName = NULL;
    bRtnBool = LookupAccountSid(
                      NULL,           // local computer
                      pSidOwner,
                      AcctName,
                      (LPDWORD)&dwAcctName,
                      DomainName,
                      (LPDWORD)&dwDomainName,
                      &eUse);

    // Reallocate memory for the buffers.
    AcctName = (char *)GlobalAlloc(
              GMEM_FIXED,
              dwAcctName);

    // Check GetLastError for GlobalAlloc error condition.
    if (AcctName == NULL) {
              DWORD dwErrorCode = 0;

              dwErrorCode = GetLastError();
              _tprintf(TEXT("GlobalAlloc error = %d\n"),
                dwErrorCode);
              return -1;
    }

    DomainName = (char *)GlobalAlloc(
           GMEM_FIXED,
           dwDomainName);

    // Check GetLastError for GlobalAlloc error condition.
    if (DomainName == NULL) {
          DWORD dwErrorCode = 0;

          dwErrorCode = GetLastError();
          _tprintf(TEXT("GlobalAlloc error = %d\n"), dwErrorCode);
          return -1;

    }

    // Second call to LookupAccountSid to get the account name.
```

```
    bRtnBool = LookupAccountSid(
          NULL,                          // name of local or remote computer
          pSidOwner,                     // security identifier
          AcctName,                      // account name buffer
          (LPDWORD)&dwAcctName,          // size of account name buffer
          DomainName,                    // domain name
          (LPDWORD)&dwDomainName,        // size of domain name buffer
          &eUse);                        // SID type

    // Check GetLastError for LookupAccountSid error condition.
    if (bRtnBool == FALSE) {
          DWORD dwErrorCode = 0;

          dwErrorCode = GetLastError();

          if (dwErrorCode == ERROR_NONE_MAPPED)
              _tprintf(TEXT("Account owner not found for specified SID.\n"));
          else
              _tprintf(TEXT("Error in LookupAccountSid.\n"));
          return -1;

    } else if (bRtnBool == TRUE)

    // Print the account name.
    _tprintf(TEXT("Account owner = %s\n"), AcctName);

    return 0;
}
```

File and Data Access

Every program that runs from the UNIX shell opens three standard files. These files have integer file descriptors and provide the primary means of communication between the programs; they also exist for as long as the process runs. You associate other file descriptors with files and devices by using the **open** system call. (See Table 15.)

Table 15. UNIX standard file descriptors

File	File descriptor	Description
Standard input	0	Standard input file provides a way to send data to a process. By default, the standard input is read from the keyboard.
Standard output	1	Standard output file provides a means for the program to output data. By default, the standard output goes to the display.
Standard error	2	Standard error is where the program reports any errors occurred during program execution. By default, the standard error goes to the display.

In Windows, in a similar manner to UNIX, when a program begins execution, the startup code automatically opens three streams:

- Standard input (pointed to by **stdin**)
- Standard output (pointed to by **stdout**)
- Standard error (pointed to by **stderr**)

These streams are directed to the console (keyboard and screen) by default. Use **freopen** to redirect **stdin**, **stdout**, or **stderr** to a disk file or a device.

The **stdout** and **stderr** streams are flushed whenever they are full or, if you are writing to a character device, after each library call. If a program terminates abnormally, output buffers may not be flushed, resulting in loss of data. Use **fflush** or **_flushall** to ensure that the buffer associated with a specified file or all open buffers are flushed to the operating system, which can cache data before writing it to disk. The commit-to-disk feature ensures that the flushed buffer contents are not lost in the event of a system failure.

Low-Level File Access

The low-level I/O functions invoke the operating system directly for lower-level operation than that provided by standard (or stream) I/O. Function calls relating to low-level input and output do not buffer or format data.

Low-level I/O functions can access the standard streams opened at program startup using the standard file descriptors. They deal with bytes of information, and this means you are using binary files, not text files. Instead of file pointers, you use low-level file handles or file descriptors, which give a unique integer number to identify each file.

UNIX example: writing to standard output

The following sample simply writes to the standard output file descriptor, and if any errors occur, it writes an error message to the standard error file descriptor.

```
#include <unistd.h>
int main()
{
    if ((write(1, "Here is some data\n", 18)) != 18)
        write(2, "A write error has occurred on file descriptor 1\n",46);
    exit(0);
}
```

Win32 example: writing to standard output

The following sample simply writes to the standard output file descriptor, and if any errors occur, it writes an error message to the standard error file descriptor.

> **Note** Just like in UNIX, the Win32 **cmd.exe** shell redirects standard error from the command line with the "2>" operator (without the quotes).

```
#include <io.h>

void main()
{
    if (write(1, "Here is some data\n", 18) != 18)
        write(2, "A write error has occurred on file descriptor 1\n",
46);
}
```

UNIX example 1: using standard input and standard output

The following sample simply reads characters from the standard input file descriptor and writes that information to the standard output file descriptor. If any input/output errors occur, an error message is output to the standard error file descriptor.

```
#include <unistd.h>
int main()
{
    char buffer[129];
    int num_read;

    num_read = read(0, buffer, 128);
    if (num_read == -1)2
        write(2, "A read error has occurred\n", 26);
    if ((write(1,buffer,num_read)) != num_read)
        write(2, "A write error has occurred\n",27);
    exit(0);
}
```

Win32 example 1: using standard input and standard output

The following sample simply reads characters from the standard input file descriptor and writes that information to the standard output file descriptor. If any input/output errors occur, an error message is output to the standard error file descriptor.

```
#include <io.h>

void main()
{
    char buffer[129];
    int num_read;

    num_read = read(0, buffer, 128);
    if (num_read == -1)
        write(2, "A read error has occurred\n", 26);

    if ((write(1, buffer, num_read)) != num_read)
        write(2, "A write error has occurred\n", 27);
}
```

UNIX example 2: using standard input and standard output

The following sample simply reads up to 1 KB of characters from the input file and writes that information to the output file. If any input/output errors occur, an error message is output to the standard error file descriptor for any input or output errors.

```
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>
```

```
int main()
{
    char block[1024];
    int in, out;
    int num_read;

    in = open("input_file", O_RDONLY);
    if (in == -1) {
        write(2, "An error has occurred opening the file:
          'input_file'\n", 52);
        exit(1);
    }
    out = open("output_file", O_WRONLY|O_CREAT, S_IRUSR|S_IWUSR);
    if (out == -1) {
        write(2, "An error has occurred opening the file:
          'output_file'\n", 53);
        exit(1);
    }
    while((num_read = read(in,block,sizeof(block))) > 0)
        write(out, block, num_read);

    exit(0);
}
```

Win32 example 2: using standard input and standard output

The following sample simply reads up to 1 KB of characters from the input file and writes that information to the output file. If any input/output errors occur, an error message is output to the standard error file descriptor for any input or output errors.

```
#include <windows.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <io.h>

void main()
{
    char block[1024];
    int in, out;
    int num_read;

    in = open("input_file", O_RDONLY);
    if (in == -1)
    {
        write(2, "An error has occurred opening the file:
          'input_file'\n", 52);
        exit (1);
    }

    out = open("output_file", O_WRONLY|O_CREAT, S_IREAD|S_IWRITE);
    if (out == -1)
    {
        write(2, "An error has occurred opening the file:
          'output_file'\n", 53);
        exit (1);
    }

    while ((num_read = read(in, block, sizeof(block))) > 0)
```

```
        write(out, block, num_read);
}
```

Standard (Stream) File Access

The standard or stream I/O functions process data in different sizes and formats, from single characters to large data structures. They also provide buffering, which can improve performance. These routines affect only buffers created by the run-time library routines, and have no effect on buffers created by the operating system.

The standard I/O library and its header file *stdio.h* provide low-level file I/O system calls. This library is part of ANSI standard C, so they port directly to Windows.

UNIX example 1: using stream file access

The following sample simply reads characters from the input file opened with the standard file I/O library function **fopen()**, and writes that information to the output file also opened with **fopen()**. Then it uses a loop of **fgetc** and **fputc** calls to transfer the contents of "input_file" to "output_file".

```
#include <stdio.h>

int main()
{
    int c;
    FILE *in, *out;

    in = fopen("input_file","r");
    if (in == NULL) {
        write(2, "An error has occurred opening the file:
          'input_file'\n", 52);
        exit(1);
    }
    out = fopen("output_file","w");
    if (out == NULL) {
        write(2, "An error has occurred opening the file:
          'output_file'\n", 53);
        exit(1);
    }

    while((c = fgetc(in)) != EOF)
        fputc(c,out);

    fclose(in);
    fclose(out);

    exit(0);
}
```

Win32 example 1: using stream file access

The following sample simply reads characters from the input file opened with the standard file I/O library function **fopen()**, and writes that information to the output file also opened with **fopen()**. Then it uses a loop of **fgetc()** and **fputc()** calls to transfer the contents of "input_file" to "output_file".

```
#include <windows.h>
```

```
#include <stdio.h>
#include <io.h>

void main()
{
    int c;
    FILE *in, *out;

    in = fopen("input_file", "r");
    if (in == NULL)
    {
        write(2, "An error has occurred opening the file:
          'input_file'\n", 52);
        exit (1);
    }

    out = fopen("output_file", "w");
    if (out == NULL)
    {
        write(2, "An error has occurred opening the file:
          'output_file'\n", 53);
        exit (1);
    }

    while ((c = fgetc(in)) != EOF)
        fputc(c, out);

    fclose(in);
    fclose(out);
}
```

UNIX example 2: using stream file access

The following sample uses **fprintf()** to format various data and print it to the file named fprintf.out.

```
#include <stdio.h>

int main()
{
    FILE *stream;
    int i = 10;
    double fp = 1.5;
    char s[] = "this is a string";
    char c = '\n';

    stream = fopen("fprintf.out", "w");
    fprintf(stream, "%s%c", s, c);
    fprintf(stream, "%d\n", i);
    fprintf(stream, "%f\n", fp);
    fclose(stream);
}
```

Win32 example 2: using stream file access

The following sample uses **fprintf** to format various data and print it to the file named fprintf.out.

```
#include <stdio.h>
```

```
void main()
{
    FILE *stream;
    int i = 10;
    double fp = 1.5;
    char s[] = "this is a string";
    char c = '\n';

    stream = fopen("fprintf.out", "w");
    fprintf(stream, "%s%c", s, c);
    fprintf(stream, "%d\n", i);
    fprintf(stream, "%f\n", fp);
    fclose(stream);
}
```

ioctl() Calls

The **ioctl()** function performs a variety of control operations on devices and streams. For non-stream files, the operations performed by this call are device-specific control operations.

```
#include <sys/ioctl.h>
#include <stropts.h>

int ioctl(int fildes, int request, /* arg */ …);
```

In Windows, a subset of the operations on a socket is provided via the **ioctlsocket()** function. The **ioctlsocket()** function supports only the SIOCATMARK command and does not have command parameter equivalent to the FIOASYNC of **ioctl()**.

Windows ioctlsocket()

The **ioctlsocket** function controls the I/O mode of a socket.

```
int ioctlsocket(SOCKET s, long cmd, u_long FAR *argp);
```

The **ioctlsocket** function can be used on any socket in any state. It is used to set or retrieve operating parameters associated with the socket, independent of the protocol and communications subsystem. Table 16 shows the supported commands to use in the command parameter and their semantics.

Table 16. Ioctlsocket parameters and semantics

Command	Description
FIONBIO	Use with a nonzero **argp** parameter to enable the non-blocking mode of socket **s**. The **argp** parameter is zero if non-blocking is to be disabled. The **argp** parameter points to an unsigned long value. When a socket is created, it operates in blocking mode by default (non-blocking mode is disabled). This is consistent with BSD sockets. The **WSAAsyncSelect** and **WSAEventSelect** functions automatically set a socket to non-blocking mode. If **WSAAsyncSelect** or **WSAEventSelect** has been issued on a socket, then any attempt to use **ioctlsocket** to set the

	socket back to blocking mode will fail with WSAEINVAL. To set the socket back to blocking mode, an application must first disable **WSAAsyncSelect** by calling **WSAAsyncSelect** with the **lEvent** parameter equal to zero, or disable **WSAEventSelect** by calling **WSAEventSelect** with the lNetworkEvents parameter equal to zero.
FIONREAD	Use to determine the amount of data pending in the network's input buffer that can be read from socket s. The **argp** parameter points to an unsigned long value in which ioctlsocket stores the result. FIONREAD returns the amount of data that can be read in a single call to the **recv** function, which may not be the same as the total amount of data queued on the socket. If s is message oriented (for example, type SOCK_DGRAM), FIONREAD still returns the amount of pending data in the network buffer; however, the amount that can actually be read in a single call to the **recv** function is limited to the data size written in the send or **sendto** function call.
SIOCATMARK	Use to determine whether or not all out-of-band (OOB) data has been read. (See the "Windows Sockets 1.1 Blocking Routines" section and EINPROGRESS for a discussion on OOB data.) This applies only to a stream-oriented socket (for example, type SOCK_STREAM) that has been configured for in-line reception of any OOB data (SO_OOBINLINE). If no OOB data is waiting to be read, the operation returns TRUE. Otherwise, it returns FALSE, and the next **recv** or **recvfrom** performed on the socket will retrieve some or all of the data preceding the mark. The application should use the SIOCATMARK operation to determine whether any data remains. If there is any normal data preceding the urgent (out-of-band) data, it will be received in order. (A recv or recvfrom never mixex OOB and normal data in the same call.) The **argp** parameter points to an unsigned long value in which **ioctlsocket** stores the Boolean result.

File Control

File Control in UNIX is implemented using the fcntl() function.

```
#include <unistd.h>
#include <fcntl.h>

int fcntl(int fd, int cmd);
int fcntl(int fd, int cmd, long arg);
int fcntl(int fd, int cmd, struct flock *lock);
```

The **fcntl()** function performs one of a number of miscellaneous operations on file descriptors. Table 17 contains the commands and semantics:

Table 17. File control commands and semantics

Command	Description
F_DUPFD	Find the lowest numbered available file descriptor greater than or equal to **arg** and make it be a copy of **fd**. On success, the new descriptor is returned.
F_GETFD	Read the close-on-exec flag.
F_SETFD	Set the close-on-exec flag to the value specified by the FD_CLOEXEC bit of **arg**.
F_GETFL	Read the descriptor's flags (all flags (as set by open(2)) are returned).
F_SETFL	Set the descriptor's flags to the value specified by arg. O_APPEND, O_NONBLOCK, and O_ASYNC may be set; the other flags are unaffected.
F_GETLK, F_SETLK and F_SETLKW	These commands are used to manage discretionary file locks. The third argument lock is a pointer to a struct flock (that may be overwritten by this call).
F_GETLK	Return the flock structure that prevents us from obtaining the lock, or set the l_type field of the lock to F_UNLCK if there is no obstruction.
F_SETLK	The lock is set (when l_type is F_RDLCK or F_WRLCK) or cleared (when it is F_UNLCK). If the lock is held by someone else, this call returns -1 and sets errno to EACCES or EAGAIN.
F_SETLKW	Like F_SETLK, but instead of returning an error, wait for the lock to be released. If a signal that is to be caught is received while **fcntl** is waiting, it is interrupted and (after the signal handler has returned) returns immediately (with return value -1 and errno set to EINTR).
F_GETOWN, F_SETOWN, F_GETSIG and F_SETSIG	These commands are used to manage I/O availability signals.
F_GETOWN	Get the process ID or process group currently receiving SIGIO and SIGURG signals for events on file descriptor **fd**. Process groups are returned as negative values.
F_SETOWN	Set the process ID or process group that will receive SIGIO and SIGURG signals for events on file descriptor **fd**. Process groups are specified using negative values. (F_SETSIG can be used to specify a different signal instead of SIGIO).
F_GETSIG	Get the signal sent when input or output becomes possible. A value of zero means SIGIO is sent. Any other value (including SIGIO) is the signal sent instead, and in this case additional info is available to the signal handler if installed with SA_SIGINFO.
F_SETSIG	Sets the signal sent when input or output becomes possible. A value of zero means to send the default SIGIO signal. Any other value (including SIGIO) is the signal to send instead, and in this case additional info is available to the signal handler if installed with SASIGINFO.

In Windows, equivalent functions are available for some of the UNIX **fcntl** commands, but not for all.

- Use **_dup()** function for F_DUPFD command.
- Use **LockFile()**, **LockFileEx()** and **UnLockFile()** functions for F_SETLK and F_SETLKW commands.

UNIX example 1: using fcntl()

The following sample opens the input file and duplicates the file descriptor. It reads the content from the input file using the duplicate file descriptor and sends it to the standard output. If any input/output errors occur, an error message is output to the standard error file descriptor for any input or output errors.

```
#include <unistd.h>
#include <fcntl.h>

int main()
{
    char block[1024];
    int fd, fd2;
    int num_read;

    fd = open("input_file", O_RDONLY);
    if (fd == -1) {
        write(2, "An error has occurred opening the file:
          'input_file'\n", 52);
        exit(1);
    }
    fd2 = fcntl(fd, F_DUPFD, 0);

    while((num_read = read(fd2, block, sizeof(block))) > 0)
        write(2, block, num_read);

    close (fd2);
    close (fd);
    exit(0);
}
```

Win32 example 1: using fcntl()

The following sample opens the input file and duplicates the file descriptor. It reads the content from the input file using the duplicate file descriptor and sends it to the standard output. If any input/output errors occur, an error message is output to the standard error file descriptor for any input or output errors.

```
#include <windows.h>
#include <io.h>
#include <stdio.h>
#include <fcntl.h>

void main()
{
    char block[1024];
    int fd, fd2;
    int num_read;

    fd = open("input_file", O_RDONLY);
    if (fd == -1)
    {
        write(2, "An error has occurred opening the file:
          'input_file'\n", 52);
        exit (1);
    }

    fd2 = dup(fd);

    while ((num_read = read(fd2, block, sizeof(block))) > 0)
        write(2, block, num_read);

    close(fd2);
    close(fd);
}
```

UNIX example 2: Using fcntl()

The following sample opens a file, sets the read lock and unlocks it. If any errors occur, an error message is output to the standard error file descriptor.

```
#include <unistd.h>
#include <fcntl.h>

int main()
{
    struct flock l;

    int fd = open("/tmp/locktest", O_RDWR|O_CREAT, 0644);
    if (fd < 0)
    {
        perror("file open error");
        exit(1);
    }

    l.l_type = F_RDLCK;
    l.l_whence = SEEK_SET;
    l.l_start = 0;
    l.l_len = 0;

    if (fcntl(fd, F_SETLK, &l) == -1)
    {
        perror("fcntl error-F_RDLCK");
        exit(1);
    }

    l.l_type = F_UNLCK;

    if (fcntl(fd, F_SETLK, &l) == -1)
    {
        perror("fcntl error-F_UNLCK");
        exit(1);
    }

    exit(0);
}
```

Win32 example 2: using fnctl()

The following sample opens two files, reads the contents of the first file, locks the second file and writes the contents of the first file, and unlocks the second file. If any errors occur, an error message is output to the console.

```
#include <windows.h>
#include <stdio.h>

void main()
{
    HANDLE hFile;
    HANDLE hAppend;
    DWORD  dwBytesRead, dwBytesWritten, dwPos;
    char   buff[4096];

    // Open the existing file.

    hFile = CreateFile("ONE.TXT",      // open ONE.TXT
        GENERIC_READ,                  // open for reading
        0,                             // do not share
        NULL,                          // no security
        OPEN_EXISTING,                 // existing file only
        FILE_ATTRIBUTE_NORMAL,         // normal file
        NULL);                         // no attr. template

    if (hFile == INVALID_HANDLE_VALUE)
    {
        printf("Could not open ONE.TXT\n");  // process error
    }

    // Open the existing file, or if the file does not exist,
    // create a new file.

    hAppend = CreateFile("TWO.TXT",    // open TWO.TXT
        GENERIC_WRITE,                 // open for writing
        0,                             // do not share
        NULL,                          // no security
        OPEN_ALWAYS,                   // open or create
        FILE_ATTRIBUTE_NORMAL,         // normal file
        NULL);                         // no attr. template

    if (hAppend == INVALID_HANDLE_VALUE)
    {
        printf("Could not open TWO.TXT\n");    // process error
    }

    // Append the first file to the end of the second file.
    // Lock the second file to prevent another process from
    // accessing it while writing to it. Unlock the
    // file when writing is finished.

    do
    {
        if (ReadFile(hFile, buff, 4096, &dwBytesRead, NULL))
        {
            dwPos = SetFilePointer(hAppend, 0, NULL, FILE_END);
            LockFile(hAppend, dwPos, 0, dwPos + dwBytesRead, 0);
            WriteFile(hAppend, buff, dwBytesRead,
                &dwBytesWritten, NULL);
            UnlockFile(hAppend, dwPos, 0, dwPos + dwBytesRead, 0);
        }
    } while (dwBytesRead == 4096);

    // Close both files.

    CloseHandle(hFile);
    CloseHandle(hAppend);
}
```

Directory Operation

Directory operations involve calling the appropriate functions to traverse a directory hierarchy or to list the contents of a directory.

Directory scanning

Directory scanning involves traversing a directory hierarchy.

Working directory

UNIX provides the **_getcwd()**, **get_current_dir_name()**, and **getwd()** functions to get the current working directory.

```
#include <unistd.h>

char *getcwd(char *buf, size_t size);
char *get_current_dir_name(void);
char *getwd(char *buf);
```

UNIX example 1: using directory handling functions

This sample prints out the current directory, and then recurses through subdirectories.

```
#include <unistd.h>
#include <stdio.h>
#include <dirent.h>
#include <string.h>
#include <sys/stat.h>

void ScanDir(char *dir, int indent)
{
    DIR *dp;
    struct dirent *dir_entry;
    struct stat stat_info;

    if((dp = opendir(dir)) == NULL) {
        fprintf(stderr,"cannot open directory: %s\n", dir);
        return;
    }
    chdir(dir);
    while((dir_entry = readdir(dp)) != NULL) {
        lstat(dir_entry->d_name,&stat_info);
        if(S_ISDIR(stat_info.st_mode)) {
            /* Directory, but ignore . and .. */
            if(strcmp(".",dir_entry->d_name) == 0 ||
                strcmp("..",dir_entry->d_name) == 0)
                continue;
            printf("%*s%s/\n",indent,"",dir_entry->d_name);
            /* Recurse at a new indent level */
            ScanDir(dir_entry->d_name,indent+4);
        }
        else printf("%*s%s\n",indent,"",dir_entry->d_name);
    }
    chdir("..");
    closedir(dp);
}

int main(int argc, char* argv[])
{
    char *topdir, defaultdir[2]=".";
    if (argc != 2) {
         printf("Argument not supplied - using current
           directory.\n");
         topdir=defaultdir;
    }
    else
        topdir=argv[1];
    printf("Directory scan of %s\n",topdir);
    ScanDir(topdir,0);
    printf("done.\n");

    exit(0);
}
```

Win32 example 1: using directory handling functions

This sample prints out the current directory, and then recurses through subdirectories. It uses the **FindFirstFile()**, **FindNextFile()**, and **FindClose()** Win32 API functions.

```
#include <windows.h>
#include <stdio.h>

void ScanDir(char *dirname, int indent)
{
    BOOL            fFinished;
    HANDLE          hList;
    TCHAR           szDir[MAX_PATH+1];
    TCHAR           szSubDir[MAX_PATH+1];
    WIN32_FIND_DATA FileData;

    // Get the proper directory path
    sprintf(szDir, "%s\\*", dirname);

    // Get the first file
    hList = FindFirstFile(szDir, &FileData);
    if (hList == INVALID_HANDLE_VALUE)
    {
        printf("No files found\n\n");
    }
    else
    {
        // Traverse through the directory structure
        fFinished = FALSE;
        while (!fFinished)
        {
            // Check the object is a directory or not
            if (FileData.dwFileAttributes & FILE_ATTRIBUTE_DIRECTORY)
            {
                if ((strcmp(FileData.cFileName, ".") != 0) &&
(strcmp(FileData.cFileName, "..") != 0))
                {
                    printf("%*s%s\\\n", indent, "",
                      FileData.cFileName);

                    // Get the full path for sub directory
                    sprintf(szSubDir, "%s\\%s", dirname,
                      FileData.cFileName);

                    ScanDir(szSubDir, indent + 4);
                }
            }
            else
                printf("%*s%s\n", indent, "", FileData.cFileName);


            if (!FindNextFile(hList, &FileData))
            {
                if (GetLastError() == ERROR_NO_MORE_FILES)
                {
                    fFinished = TRUE;
                }
            }
        }
    }

    FindClose(hList);
}

void main(int argc, char *argv[])
{
    char *pszInputPath;
    char pwd[2] = ".";

    if (argc < 2)
    {
        printf("Argument not supplied - using current directory.\n");
        pszInputPath = pwd;
    }
    else
    {
        pszInputPath = argv[1];
        printf("Input Path: %s\n\n", pszInputPath);
    }

    ScanDir(pszInputPath, 0);

    printf("\ndone.\n");
}
```

UNIX example 2: using directory handling functions

This sample prints out the current working directory using the **getcwd()** function.

```
#include <unistd.h>
```

```
#include <stdio.h>

int main()
{
    char *cwd;
    char buffer[129];

    if ((cwd = getcwd(buffer, 128)) == NULL)
    {
        perror("Current working directory-getcwd() failed");
        exit(2);
    }

    printf("Current working directory: %s\n", cwd);

    exit(0);
}
```

Win32 example 2: using directory handling functions

This sample prints out the current working directory using the **_getcwd()** function. This function gets the full path of the current working directory for the default drive. It returns a string that represents the path of the current working directory. If the current working directory is the root, the string ends with a backslash (\). If the current working directory is a directory other than the root, the string ends with the directory name and not with a backslash.

```
#include <direct.h>
#include <stdlib.h>
#include <stdio.h>

void main( void )
{
   char buffer[_MAX_PATH];

   /* Get the current working directory: */
   if( _getcwd( buffer, _MAX_PATH ) == NULL )
      perror( "_getcwd error" );
   else
      printf( "%s\n", buffer );
}
```

UNIX example 3: using directory handling functions

This sample prints out the current working directory using the **get_current_dir_name()** function.

```
#include <unistd.h>
#include <stdio.h>

int main()
{
    char *cwd;

    cwd = (char *)get_current_dir_name();

    printf("Current working directory: %s\n", cwd);

    exit(0);
}
```

Win32 example 3: using directory handling functions

This sample prints out the current working directory using the **GetCurrentDirectory()** Win32 API function.

```
#include <windows.h>
#include <stdio.h>

void main()
{
    DWORD   cchCurDir;
    LPTSTR  lpszCurDir;
    TCHAR   tchBuffer[MAX_PATH + 1];
    DWORD   nSize;


    lpszCurDir = tchBuffer;
    cchCurDir = MAX_PATH;

    nSize = GetCurrentDirectory(cchCurDir, lpszCurDir);

    printf("Current Directory is : %s\n", lpszCurDir);
}
```

Interprocess Communication

Like UNIX, Windows has various forms of interprocess communication (IPC). The forms that are most familiar to UNIX developers are:

- Process pipes
- Named pipes
- Message queues
- Sockets
- Memory-mapped files
- Shared memory

Windows supports implementations of five of these (that is, all except Message Queues). For further details see the "Microsoft Message Queues" section.

In addition to these there are two forms of IPC that are not part of the Win32 API. These are Message Queuing (also known as MSMQ) and COM+.

This section looks at how you convert UNIX code that uses the different forms of IPC. It also introduces new methods of IPC that are not available in UNIX, but may provide you with a better solution for your application's interprocess communication. The subject of sockets has been left until the next section.

Process Pipes

Process pipes are supported in Win32 by using the standard C runtime library. As discussed in the following sections, they are largely equivalent to process pipes in UNIX.

High-level popen call

Three UNIX examples are considered in this section, and a slightly modified Win32 version of each has also been provided. Note that this text assumes the presence of the **Uname.exe** executable program on the Windows-based system. If your system does not contain this executable, these samples will not work and you will need to modify them to use an equivalent utility.

In the first example of process pipes, a process called **uname** is executed and passes the output of this process to standard output. As you review these examples, notice that the differences between the UNIX and Win32 implementations are the header files that are required and the function names for **popen** and **pclose**. The names of these functions in Win32 are preceded by an underscore. The function syntax is the same and the behavior is largely compatible.

UNIX example: process pipes

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main()
{
    FILE *read_fp;
    char buffer[BUFSIZ + 1];
    int chars_read;
    memset(buffer, '\0', sizeof(buffer));
    read_fp = (FILE *) popen("uname -a", "r");
    if (read_fp != NULL) {
        chars_read = fread(buffer, sizeof(char), BUFSIZ, read_fp);
        if (chars_read > 0) {
            printf("Output is:\n%s\n", buffer);
        }
        pclose(read_fp);
        exit(EXIT_SUCCESS);
    }
    exit(EXIT_FAILURE);
}
```

Win32 example: process pipes

```
#include <windows.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

void main()
{
    FILE *read_fp;
    char buffer[BUFSIZ + 1];
    size_t chars_read;
    memset(buffer, '\0', sizeof(buffer));
    read_fp = (FILE *)_popen("uname -a", "r");
    if (read_fp != NULL) {
        chars_read = fread(buffer, sizeof(char), BUFSIZ, read_fp);
        if (chars_read > 0) {
            printf("Output is:\n%s\n", buffer);
        }
        _pclose(read_fp);
        exit(EXIT_SUCCESS);
    }
    exit(EXIT_FAILURE);
}
```

In the second example of process pipes, the code sends data into the **Od.exe** command. You need to have the **Od.exe** command installed for these examples to work. The **Od.exe** command converts binary data into octal format. The output from **Od.exe** goes to the console. Again, notice that the only difference between the two solutions is in the required header files and the underscore preceding the function names.

Second UNIX example: process pipes

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main()
{
    FILE *write_fp;
    char buffer[BUFSIZ + 1];

    sprintf(buffer, "This is counting words");
    write_fp = (FILE *) popen("wc -w", "w");
    if (write_fp != NULL) {
        printf("This string: '%s' has this many words:\n", buffer);
        fwrite(buffer, sizeof(char), strlen(buffer), write_fp);
        pclose(write_fp);
        exit(EXIT_SUCCESS);
    }
    exit(EXIT_FAILURE);
}
```

Second Windows example: process pipes

```
#include <windows.h>
#include <stdlib.h>
#include <stdio.h>

void main()
{
    FILE *write_fp;
    char buffer[BUFSIZ + 1];

    sprintf(buffer, "This is counting words");
    write_fp = _popen("wc -w", "w");
    if (write_fp != NULL) {
        printf("This string: '%s' has this many words:\n", buffer);
        fwrite(buffer, sizeof(char), strlen(buffer), write_fp);
        _pclose(write_fp);
        exit(EXIT_SUCCESS);
    }
    exit(EXIT_FAILURE);
}
```

To clarify the coding differences between UNIX and Win32, an example of process pipes has been included in Appendix D: Process Pipes.

Low-level pipe call

This section demonstrates an example of how you should convert code that uses pipes to communicate between two parts of an application using the pipe function call. The example below demonstrates how you can write to one end of the pipe (fd[1]) and read from the other (fd[0]). As you might expect from the previous examples, the differences between the UNIX and the Win32 implementations are the header files required and the underscore that precedes the pipe function. In this case, however, an additional modification has to be made before the solution will work when using Windows. If you look at the online documentation for pipe, you'll notice that it requires two additional arguments. Providing these arguments specifies the amount of memory to be used as a buffer for the pipe, as well as the mode of the pipe (O_TEXT or O_BINARY).

UNIX example: low-level pipe call

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main()
{
    int data_out, data_in, file_pipes[2];
    const char data[] = "ABCDE";
    char buffer[BUFSIZ + 1];

    memset(buffer, '\0', sizeof(buffer));
    if (pipe(file_pipes) == 0) {
        data_out = write(file_pipes[1], data, strlen(data));
        printf("Wrote %d bytes\n", data_out);
        data_in = read(file_pipes[0], buffer, BUFSIZ);
        printf("Read %d bytes: %s\n", data_in, buffer);
        exit(EXIT_SUCCESS);
    }
    exit(EXIT_FAILURE);
}
```

Windows example: low-level pipe call

```
#include <windows.h>
#include <stdlib.h>
#include <stdio.h>
#include <io.h>
#include <fcntl.h>
#include <process.h>

void main()
{
    size_t data_out, data_in;
    int file_pipes[2];
    const char data[] = "ABCDE";
    char buffer[BUFSIZ + 1];

    memset(buffer, '\0', sizeof(buffer));
    if (_pipe(file_pipes, 32, O_BINARY) == 0) {
        data_out = write(file_pipes[1], data, strlen(data));
        printf("Wrote %d bytes\n", data_out);
        data_in = read(file_pipes[0], buffer, BUFSIZ);
        printf("Read %d bytes: %s\n", data_in, buffer);
        exit(EXIT_SUCCESS);
    }
    exit(EXIT_FAILURE);
}
```

More pipes examples

To help you understand the changes needed in applications using pipes for interprocess communications two more examples are included in the appendixes. These are:

- Appendix E: Pipes Across a Process
- Appendix F: Pipes Used as Standard Input/Output

Named Pipes (FIFOs)

Process pipes were covered in the previous section. In this section, a few examples of named pipes are shown. These are sometimes referred to as first-in-first-out (FIFO).

Interprocess communication with named pipes

In order to show how you can convert code using FIFO from UNIX to Windows, a simple example that creates a named pipe is shown. This example uses minimal security restrictions for simplicity. The example uses the **mkfifo** function in UNIX and the **CreateNamedPipe** function when using Win32. There are considerable differences between these two functions. Both functions have the same purpose, however, in that the **CreateNamedPipe** function offers a greater degree of control over the configuration of the pipe. The details are beyond the scope of this guide, but you can find the links for **CreateNamedPipe** on the MSDN Web site.

The examples can be seen in full in the following appendixes:

- Appendix H: Creating a Named Pipe
- Appendix I: Opening a FIFO
- Appendix J: Interprocess Communication with FIFOs

Message Queues

Microsoft Win32 doesn't support message queues as standard. If you want to use message queuing in your application, you should use Microsoft Message Queuing (also known as MSMQ). Message Queuing is covered comprehensively in other Microsoft documentation and it is therefore only briefly described here.

Note For more information on Message Queuing, see What's new in Windows XP?.

Message Queuing technology enables applications running at different times to communicate across heterogeneous networks and systems that may be temporarily offline. Applications send messages to queues and read messages from queues. Message Queuing provides guaranteed message delivery, efficient routing, security and priority-based messaging. It can be used to implement solutions for both asynchronous and synchronous scenarios requiring high performance.

Sockets and Networking

Windows Sockets 2 uses the sockets paradigm that was first popularized by Berkeley Software Distribution (BSD) UNIX. It was later adapted for Microsoft Windows in Windows Sockets 1.1.

One of the primary goals of Winsock has been to provide a protocol-independent interface fully capable of supporting emerging networking capabilities, such as real-time multimedia communications.

Winsock is an interface, not a protocol. As an interface, it is used to discover and utilize the communications capabilities of any number of underlying transport protocols. Because it is not a protocol, it does not in any way affect the *bits on the wire*, and does not need to be utilized on both ends of a communications link.

Windows Sockets programming previously centered on TCP/IP. Some of the programming practices that worked with TCP/IP do not work with every protocol. As a result, the Winsock API added new functions where necessary to handle several protocols.

Winsock has changed its architecture to provide easier access to multiple transport protocols. Following the Windows Open System Architecture (WOSA) model, Winsock now defines a standard service provider interface (SPI) between the application-programming interface (API), with its functions exported from Ws2_32.dll and the protocol stacks. Consequently, Winsock support is not limited to TCP/IP protocol stacks as is the case for Windows Sockets 1.1. For more information, see the discussion of "Windows Sockets 2 Architecture" in Microsoft Visual Studio.

There are new challenges in developing Windows Sockets 2 applications. When sockets only supported TCP/IP, a developer could create an application that supported only two socket types: connectionless and connection-oriented. Connectionless protocols used SOCK_DGRAM sockets and connection-oriented protocols used SOCK_STREAM sockets. These are just two of the many new socket types. Additionally, developers can no longer rely on socket type to describe all the essential attributes of a transport protocol.

Sockets are not part of the Win32 library. You'll need to consult the platform SDK for detailed information about the WinSock API's. The online help for the platform SDK contains complete samples that demonstrate how to implement socket-based client and server applications. For an in-depth comparison between UNIX sockets and WinSock, refer to the discussion of Socket Programming Considerations in the Microsoft Platform SDK.

The Process Environment

The process environment includes several key elements, which are explained here. The notable differences between these elements in Windows are also described briefly. This section only discusses POSIX.

Environment Variables

Every process has an environment block associated with it. An environment block is a block of memory allocated within the process's address space. Each block contains a set of name value pairs. Both UNIX and Windows support process environment blocks. The particular differences may vary depending on what supplier and version of UNIX you are dealing with. For information you can use to conduct your own comparison, see the MSDN article, Changing Environment Variables.

A summary of the notable differences between environment variables in Win32 and POSIX is also provided.

Differences between POSIX and Win32 environment variables

Win32 supports an ANSI version of the environment functions as well as a Unicode variant. The Unicode variants are preceded by a **_w** prefix. Using the **_w** prefix in your application helps to ensure that the application is linked with the correct variant when compiled with _UNICODE or _MBCS preprocessor strings. In addition to the ANSI functions **putenv** and **getenv**, Win32 also supports the **GetEnvironmentVariable**, **ExpandEnvironmentStrings**, and **SetEnvironmentVariable** functions. This example only shows the ANSI functions. Using the ANSI functions provides you with the simplest method of converting your code from UNIX to Win32.

A simple example of accessing the environment block is included below. This is an example that works equally well in POSIX and Win32.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[])
{
    char *var, *value;

    if(argc == 1 || argc > 3) {
        fprintf(stderr,"usage: environ var [value]\n");
        exit(1);
    }
    var = argv[1];
    value = getenv(var);
    if(value)
        printf("Variable %s has value %s\n", var, value);
    else
        printf("Variable %s has no value\n", var);

    if(argc == 3) {
        char *string;
        value = argv[2];
        string = (char *)malloc(strlen(var)+strlen(value)+2);
        if(!string) {
            fprintf(stderr,"out of memory\n");
            exit(1);
        }
        strcpy(string,var);
        strcat(string,"=");
        strcat(string,value);
        printf("Calling putenv with: %s\n",string);
        if(putenv(string) != 0) {
            fprintf(stderr,"putenv failed\n");
            free(string);
            exit(1);
        }

        value = getenv(var);
        if(value)
            printf("New value of %s is %s\n", var, value);
        else
            printf("New value of %s is null??\n", var);
    }
    exit(0);
 }
```

Temporary Files

Both UNIX and Win32 support functions that create temporary files. The example below works equally well in UNIX and Win32; no modifications are required.

The **tmpnam()** function returns a pointer to a temporary file name. **_tempname()** does this as well, but you can also use it to specify the directory and file name prefix.

The following is an example:

```
#include <stdlib.h>
#include <stdio.h>

int main()
{
    char tmpname[L_tmpnam];
    char *filename;
    FILE *tmpfp;

    filename = tmpnam(tmpname);

    printf("Temporary file name is: %s\n", filename);

    tmpfp = tmpfile();
    if(tmpfp)
  {
        printf("Opened a temporary file OK\n");
  exit(1);
  }
    else
  {
        perror("tmpfile");
  exit(0);
  }
}
```

Computer Information

At times, it is necessary to obtain information about a computer. This is particularly important when an application is designed to support multiple users or different types of hardware and operating systems. Some of the pieces of information that applications require are:

- The hostname
- The operating system name

- The network name of the computer
- The release level of the operating system
- The version number of the operating system
- The hardware platform name

In UNIX, you would use a combination of **gethostname** and **uname** functions to obtain this information. When using Windows, you have the option of using **gethostname**, but **uname** is not available as standard in the Win32 API. It is possible to add uname using a POSIX layer. Applications that use this function need to be rewritten to use a different set of services.

The Platform SDK has functionality to obtain the similar set of information provide by the **uname** function. The Platform SDK mappings are covered in this text, but it is recommended that you consider using the Windows Management Instrumentation (WMI) API. The WMI interface is a superset to the Win32API for obtaining information about the computer. It is highly extensible and supports not only static information about a platform, but also dynamic information such as configuration and performance data. Another source to consider is the Active Directory Services Interface (ADSI) a COM interface that facilitates access to information stored in Microsoft Active Directory® directory-service database for the enterprise. Both of these interfaces represent the preferred mechanism for gathering information on Microsoft Windows 2000 and later computers and networks.

For a complete list of the system-information functions provided by the platform SDK, you can refer to the System Information Functions in the online platform SDK documentation. Two functions that are not used in the following Win32 example, but are also useful, are GetVersionEx and VerifyVersionInfo.

UNIX example: using system information

```
#include <unistd.h>
#include <stdio.h>
#include <sys/utsname.h>

int main()
{
    char computer[256];
    struct utsname uts;

    if(gethostname(computer, 255) != 0 || uname(&uts) < 0) {
        fprintf(stderr, "Could not get host information\n");
        exit(1);
    }

    printf("Computer host name is %s\n", computer);
    printf("System is %s on %s hardware\n", uts.sysname,
      uts.machine);
    printf("Nodename is %s\n", uts.nodename);
    printf("Version is %s, %s\n", uts.release, uts.version);
    exit(0);
}
```

Windows example: using system information

```
#define _WIN32_WINNT 0X0500
#include <windows.h>
#include <stdlib.h>
#include <stdio.h>
```

```
void errabt(char *msg)
{
    fprintf(stderr, msg);  // use GetLastError for more detailed
info.
    exit(1);
}

void main()
{
    DWORD nSize= 255;
    char computer[256];
    char nodename[256];
    SYSTEM_INFO siSysInfo;    // Struct for hardware info
    OSVERSIONINFO siVerInfo;    // Struct for version info

    GetSystemInfo(&siSysInfo);  // Get hardware OEM

    // Get major and minor number
    ZeroMemory(&siVerInfo, sizeof(OSVERSIONINFO));
    siVerInfo.dwOSVersionInfoSize = sizeof(OSVERSIONINFO);
    if (!GetVersionEx((OSVERSIONINFO *) &siVerInfo))
        errabt("Could not get OS Version info\n");

    nSize = 255;
    if (GetComputerNameEx(ComputerNameNetBIOS, computer, &nSize) ==
FALSE)
        errabt("Could not get NETBIOS name of computer\n");

    nSize = 255;
    if (GetComputerNameEx(ComputerNameDnsFullyQualified, nodename,
      &nSize) == FALSE)
        errabt("Could not get FQDNS Name of computer\n");

    printf("Computer host name is %s\n", computer);
    printf("System is %u on %u hardware\n",
                siVerInfo.dwMajorVersion,siSysInfo.dwProcessorType);
                  printf("Nodename is %s\n", nodename);
    printf("Version is %d.%d %s (Build %d)\n",
        siVerInfo.dwMajorVersion,
        siVerInfo.dwMinorVersion,
        siVerInfo.szCSDVersion,
        siVerInfo.dwBuildNumber & 0xFFFF);
    exit(0);
}
```

Logging System Messages

Logging diagnostic messages in UNIX, as in the following example, is carried out by writing formatted output to the system logger. The message is written to system log files such as USERS, or forwarded to the appropriate computer. If log daemon process is not running, the log information may be written to a standard log file such as /var/adm/log/logger.

The daemon syslogd in UNIX contains numerous levels of logged information, as can be seen in Table 18, below:

Table 18. UNIX logging system messages

Priority	Description
LOG_EMERG	A panic condition
LOG_ALERT	A condition that should be corrected immediately
LOG_CRIT	Critical conditions such as hard device errors
LOG_ERR	Errors
LOG_WARNING	Warnings
LOG_NOTICE	Non error related conditions
LOG_INFO	Informational messages
LOG_DEBUG	Messages intended for debug purposes

In contrast, the Windows Event Log also supports logging levels as can be seen in Table 19.

Table 19. Windows event logging messages

Priority	Description
EVENTLOG_SUCCESS	Information events indicate infrequent but significant successful operations. For example, when Microsoft SQL Server™ successfully loads, it may be appropriate to log an information event stating that "SQL Server has started." Note that while this is appropriate behavior for major server services, it is generally inappropriate for a desktop application (Microsoft Excel, for example) to log an event each time it starts.
EVENTLOG_ERROR_TYPE	Error events indicate significant problems that the user should know about. Error events usually indicate a loss of functionality or data. For example, if a service cannot be loaded as the system boots, it can log an error event.
EVENTLOG_WARNING_TYPE	Warning events indicate problems that are not immediately significant, but that may indicate conditions that could cause future problems. Resource consumption is a good candidate for a warning event. For example, an application can log a warning event if disk space is low. If an application can recover from an event without loss of functionality or data, it will generally classify the event as a warning event.
EVENTLOG_INFORMATION_TYPE	Information events indicate infrequent but significant successful operations. For example, when Microsoft SQL Server successfully loads, it may be appropriate to log an information event stating that "SQL Server has started." Note that while this is appropriate behavior for major server services, it is generally inappropriate for a desktop application (Microsoft Excel, for example) to log an event each time it starts.
EVENTLOG_AUDIT_SUCCESS	Success audit events are security events that occur when an audited access attempt is successful. For

	example, a successful logon attempt is a successful audit event.
EVENTLOG_AUDIT_FAILURE	Failure audit events are security events that occur when an audited access attempt fails. For example, a failed attempt to open a file is a failure audit event.

As you may have observed, the Windows event logging mechanism supports a smaller selection of event priorities. You can augment the priority status of event messages by including category information and binary data in the event log. This additional event information is part of the Win32 example below.

System logging in UNIX

```
#include <syslog.h>
#include <stdio.h>

int main()
{
    FILE *fp;

    fp = fopen("Bad_File_Name","r");
    if(!fp)
        syslog(LOG_INFO|LOG_USER,"error - %m\n");
    exit(0);
}
```

On a "typically" configured Linux system, this message would be logged to /var/log/messages, on a Solaris system to /var/adm/messages, and on Interix (when syslogd in running) to /var/adm/log/messages. Consult the /etc/syslog.conf file for more specific information, specifically a *.info entry will specify the file where the above message will be logged.

System logging in Win32

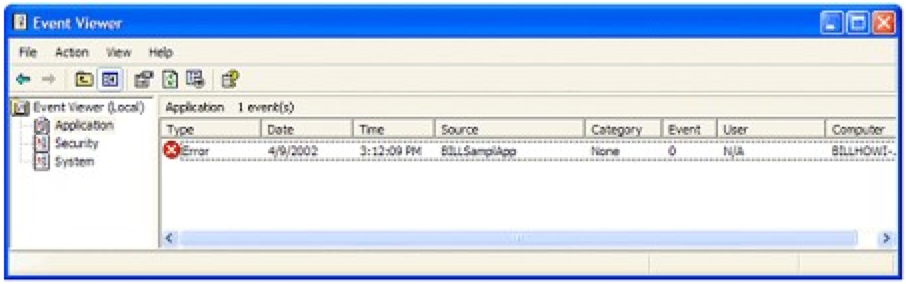
```
#include <windows.h>
#include <stdlib.h>

void main()
{
    HANDLE h;
    LPSTR mstr = "This is an error from my sample app.";


    h = RegisterEventSource(NULL,  // uses local computer
             TEXT("BILLSamplApp"));     // source name
    if (h == NULL)
        exit(1);

    ReportEvent(h,                 // event log handle
            EVENTLOG_ERROR_TYPE,  // event type
            0,                    // category zero
            0,                    // event identifier
            NULL,                 // no user security identifier
            1,                    // one substitution string
            0,                    // no data
    (LPCSTR*)&mstr,       // pointer to string array
            NULL);                // pointer to data
    DeregisterEventSource(h);
  exit(0);
}
```

In the above example, the source name to the RegisterEventSource call is not available in the system registry. As a result, you will not see valid mapping or lookup data when you view the event log with the event Viewer. After running this sample, Eventvwr.exe should yield something similar to Figure 9.2.

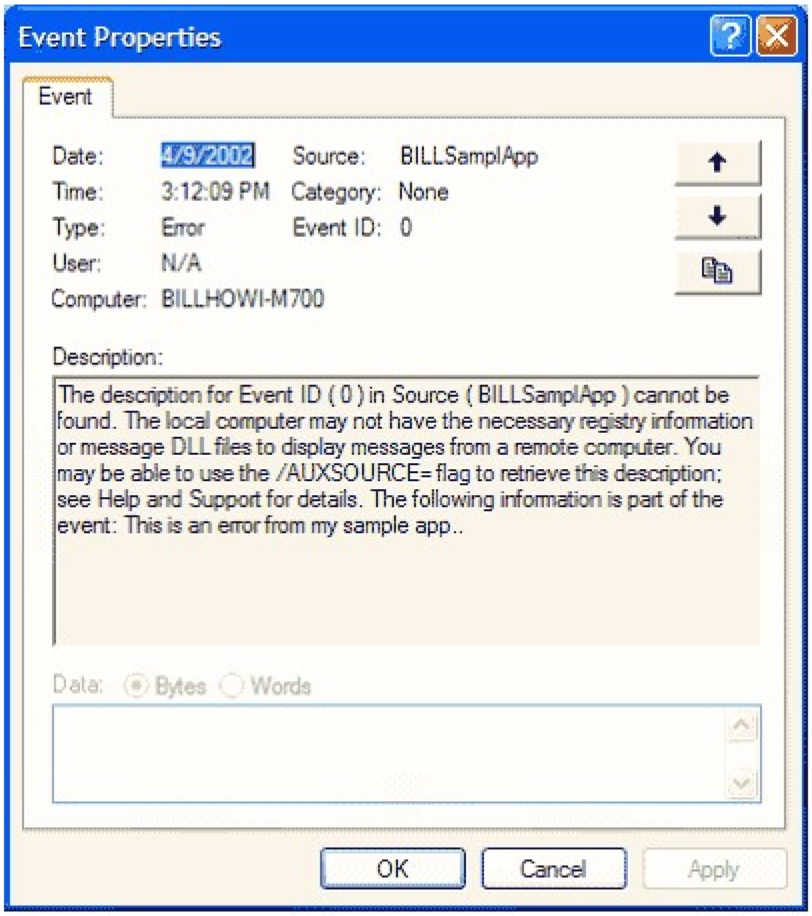


Application 1 event(s)

Type	Date	Time	Source	Category	Event	User	Computer
Error	4/9/2002	3:12:09 PM	BILLSampApp	None	0	N/A	BILLHOWI-...

Figure 2. Windows Event Viewer

Double-clicking the error line opens a detailed view of the event (see Figure 9.3).



Date: 4/9/2002 Source: BILLSamplApp
Time: 3:12:09 PM Category: None
Type: Error Event ID: 0
User: N/A
Computer: BILLHOWI-M700

Description:

The description for Event ID (0) in Source (BILLSamplApp) cannot be found. The local computer may not have the necessary registry information or message DLL files to display messages from a remote computer. You may be able to use the /AUXSOURCE= flag to retrieve this description; see Help and Support for details. The following information is part of the event: This is an error from my sample app..

Data: [✓ Bytes] Words

Figure 3. Details of an Event in Windows Event Viewer

The preceding example is a very simple example of generating log information and posting it to the Windows Event log. A complete application would use more of the Platform SDK facilities to create an application entry in the registry or perhaps create an entirely separate event log file. For a complete dissertation on the details and complexities of event logging in Windows, refer to Event Logging on the MSDN® Web site.

Multiprocessor Considerations

Computationally intensive tasks are characterized by intensive processor usage with relatively few I/O operations. The ongoing challenge with these applications is to improve the performance. You can do this by getting a faster computer, choosing a more efficient algorithm, improving the implementation or by using more processors. Improving the performance is accomplished by tuning techniques, which are covered in the Using the Tools section of Chapter 7. Utilizing more processors can mean taking advantage of a symmetric multiprocessor (SMP) computer or by using distributed computing with multiple networked computers. This section looks at these techniques and how to port UNIX implementations to Win32.

Process and Thread Solutions

How to take advantage of symmetric multiprocessors is discussed in this section. On a multiprocessor UNIX system, the typical approach is to spawn several processes to run on each of the available processors. Each processor can run one process simultaneously with all the other processors so you achieve true parallelism. The technique usually employed is to have an administrative parent process that starts a computational child process for each available processor.

The other technique for exploiting SMP hardware is to use POSIX threads on UNIX releases that support them. The details of converting POSIX threads to Windows were covered earlier in this chapter in the preceding "Threads" section. If your computationally intensive application already uses POSIX threads for parallelism, you can use the porting techniques covered there.

The classic parent process code on UNIX will call **fork()** to create a child process for each processor and wait for the children to finish computing. Multiple processes and maintaining a parent-child relationship on Windows is demonstrated here. The code shown in the preceding "Waiting for a Spawned Process" section is extended below to create multiple children:

```
#include <unistd.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#define MAX_PROCS 4

int main()
{
    pid_t pid;
    pid_t children[MAX_PROCS];
    int   child_running[MAX_PROCS];
    int   tstat, ii, running;

    for (ii = 0; ii < MAX_PROCS; ++ii)
    {
        pid = fork();
        switch(pid)
        {
        case -1:
            perror("fork failed");
```

```
                child_running[ii] = 0;
                break;
            case 0:
                printf("I'm child #%d\n", ii);
                exit(0);
            default:
                children[ii] = pid;
                child_running[ii] = 1;
                break;
            }
        }

// wait for children to finish
    running = MAX_PROCS;
    while (running) {
        pid = wait(&tstat);
        for (ii = 0; ii < MAX_PROCS; ++ii) {
// determine which child
            if (pid == children[ii]) {
// child done
                child_running [ii] = 0;
                running--;
                printf("child #%d finished\n", ii);
            }
        }
    }
    exit(0);
}
```

The first problem in porting this type of code to Win32 is to do something with the **fork()** call. Recall from the previous sections on "Process" code migration that Windows starts processes with the **CreateProcess** call, which starts a new process and the application running in it. This combines both the **fork()** and the **exec()** calls.

The child process must have some way to communicate with the parent to get the data it needs to work with, to update status, or to return results. There are several ways this can be done. If the child process continues with the same application, the data will probably already be set up in global variables. If the child executes a different application, the data may be transferred by using pipes, message queues, shared memory or even files. To keep the following example uncomplicated, the parent application passes all the required data by using the environment set up for each child. This data is used to connect to common events that allow the parent to signal the children to perform an action, or exit.

Winforks Example

The **Winforks.exe** example follows the UNIX program outlined above. The parent creates a number of child processes that do the parts of some jobs in parallel. The parent waits for the child processes to exit.

The parent passes the data necessary for the child to execute in the environment set up for the child. The environments are created from scratch by the parent and none of the existing environment is copied. This technique can be used to pass file names, flags, and other small amounts of data to the child processes. However, this can only be used to pass data to the child and cannot be used to retrieve data from the child. The details of passing the results back are not shown here, but if the initial data input is from a file then the output file could also be specified to the child and retained by the parent.

This program also handles error indications from the child processes and stops those that are still executing on error. In Windows, killing a process is immediate, and the process is not informed of the action so it cannot perform any cleanup operations. To allow the child process a chance to clean up and close any resources it is using, the parent passes the name of an exit event object to each child. If the child detects this exit event, it can clean up and stop gracefully.

This sample has three files. The first is a shared include file, Both.h, that contains the names of the environment variables that the parent sets and the child reads. The child process code, ChildExe.c, contains only a **main()** function, which gets the initialization data from the environment and loops until it is done.

The parent process file, WinForks.c, contains these definitions and functions:

- A child process information structure to hold the child process handle, exit event, and other information.
- A **main()** function that creates the child processes and waits for them to exit.
- A **ChildStart** function to start a child process. This takes the place of the UNIX **fork()** call. This function creates the environment passed to the child and the exit event. Finally it sets up the information necessary to start the child and calls **CreateProcess**.
- A **ChildCleanup** function to release resources when a child exits.
- A function to stop a child when an error occurs.

Winforks example code is shown in Appendix K.

Common Variations

If the application is using **fork()** to create multiple copies of itself (that is, if the child process does not **exec()** to run another program), there are two completely different methods to approach the migration. The first is to continue to use multiple processes and use shared memory to allow the child processes to access the parent's data. This works if the application is using the technique where global memory has already been set up for the child processes to access. Shared memory in Windows is clear-cut and was covered in the Shared Memory section of this chapter.

The second and preferred solution where an application is using **fork()** is to create copies of itself is to convert the program to use Win32 threads. Creating a thread in Windows is much faster and uses fewer system resources than creating a process. The conversion of this type of code to use threads should be uncomplicated. Of course, code to make the application thread-safe will likely be needed. Threading locks can be implemented using mutexes or critical sections. The details of how to do this were covered earlier in this chapter.

Another variation you might see in applications that depend on **fork()** is the use of pipes to transfer data and synchronize the parent and child processes. Again, converting this code to use Windows named pipes is easy. See the preceding "Interprocess Communication" section for details on converting UNIX pipes to Windows pipes.

Daemons and Services

A UNIX daemon is a process that runs in the background and does not require a user interface. A service application is the equivalent on Windows. Normally, a daemon is started when the system is booted and runs without supervision until the system is shut down. Similarly, a Windows service can be started at boot time and run until shut down. However, the service control manager (SCM) controls all services, so to convert daemon code to run on Windows, you must add code to interface with the SCM.

Unless the **main()** function of the daemon is extremely simple, the best strategy is to rename it to something else, like **service_main()**. Then create a new main that contains code to install, uninstall, and run the service depending on command line arguments.

To install the program as a service, the program must call **OpenSCManager** to get a handle to the SCM. It must then call **CreateService**, passing the SCM handle and several arguments, including the service name, display name, service type, path to the executable, and identity the service uses to run. Once the service is installed, the administrative tools services applet may be used to examine and modify many these values.

Uninstalling the service is similar to installing it. Call **OpenSCManager** to get a handle to the SCM, and then call **OpenService** to get a handle to the service. If it is running, the service should be stopped by calling the **ControlService** function. Finally, call **DeleteService** passing the service handle, and close the handles to clean up.

Running the daemon as a service is also pretty straightforward. The new main function sets up a SERVICE_TABLE_ENTRY structure that contains a name and a pointer to the service main, which is the old main function renamed **service_main**. This structure is passed to **StartServiceCtrlDispatcher**, which does not return until the service stops. Note that more than one service entry can be in the service entry structure, so a single executable can support more than one service. With definitions the same as a main() function, the arguments to the service main function are supplied by the SCM and can be set by the services applet or the **CreateService** call.

The service main function needs new code to call the function **SetServiceStatus** that keeps the SCM informed of the service's status during startup. If the SCM does not receive status updates within a specified time period, it assumes that the service has stopped running and logs an error. The SCM must also be given the address of a service control function that it uses to inform the service when it should stop and for other requests. Call **RegisterServiceCtrlHandler** or **RegisterServiceCtrlHandlerEx** to set this address. When the service is fully initialized, it should call **SetServiceStatus** with the SERVICE_RUNNING status to complete the startup sequence.

Refer to the MSDN Web site for sample service programs and details of the service functions.

Appendixes

These appendixes contain a number of code examples demonstrating how to convert code to Win32.

Appendix A: Shared Memory

The following is an example of typical UNIX code that uses shared memory.

UNIX example: shared memory

Consumer

```
/* This program is a consumer. The shared memory segment is
   created with a call to shmget, with the IPC_CREAT bit specified.
*/
```

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#include "shm_com.h"

int main()
{
    void *shared_memory_loc = (void *)0;
    struct shared_struct *shared_stuff;
    int shmid;

    shmid = shmget((key_t)1111, sizeof(struct shared_struct), 0666 |
      IPC_CREAT);
    if (shmid == -1) {
        fprintf(stderr, "shmget function failed\n");
        exit(EXIT_FAILURE);
    }

/* Make the shared memory accessible to the program. */
    shared_memory_loc = shmat(shmid, (void *)0, 0);
    if (shared_memory_loc == (void *)-1) {
        fprintf(stderr, "shmat function failed\n");
        exit(EXIT_FAILURE);
    }
    printf("Memory attached at %X\n", (int)shared_memory_loc);

/* Assign the shared_memory_loc segment to shared_stuff.
   Echo any text in "some_text".
   Continues until end is found in "some_input" (1 in stored by
Provider).
*/
    shared_stuff = (struct shared_struct *)shared_memory_loc;
    shared_stuff->some_input = 0;
    while(1) {
        if (shared_stuff->some_input) {
            printf("You wrote: %s", shared_stuff->some_text);
            sleep(1); /* the Provider is waiting for this process */
            shared_stuff->some_input = 0;
            if (strncmp(shared_stuff->some_text, "done", 4) == 0) {
                break;
            }
        }
    }

/* Detach and Delete shared memory */
    if (shmdt(shared_memory_loc) == -1) {
        fprintf(stderr, "shmdt function failed\n");
        exit(EXIT_FAILURE);
    }
    if (shmctl(shmid, IPC_RMID, 0) == -1) {
        fprintf(stderr, "shmctl(IPC_RMID) function failed\n");
        exit(EXIT_FAILURE);
    }
    exit(EXIT_SUCCESS);
}
```

Provider

```
/* This program is a provider of input text for the consumer. */
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#include "shm_com.h"

int main()
{
    void *shared_memory_loc = (void *)0;
    struct shared_struct *shared_stuff;
    char buffer[BUFSIZ];
    int shmid;

    shmid = shmget((key_t)1111, sizeof(struct shared_struct), 0666 |
IPC_CREAT);
    if (shmid == -1) {
        fprintf(stderr, "shmget function failed\n");
        exit(EXIT_FAILURE);
    }

    shared_memory_loc = shmat(shmid, (void *)0, 0);
    if (shared_memory_loc == (void *)-1) {
        fprintf(stderr, "shmat function failed\n");
        exit(EXIT_FAILURE);
    }
    printf("Memory attached at %X\n", (int)shared_memory_loc);

    shared_stuff = (struct shared_struct *)shared_memory_loc;
    while(1) {
        while(shared_stuff->some_input == 1) {
            printf("waiting for Consumer...\n");
            sleep(1);
        }
        printf("Enter some text: ");
        fgets(buffer, BUFSIZ, stdin);
        strncpy(shared_stuff->some_text, buffer, TEXT_SZ);
        shared_stuff->some_input = 1;
        if (strncmp(buffer, "done", 4) == 0) {
              break;
        }
    }

    if (shmdt(shared_memory_loc) == -1) {
        fprintf(stderr, "shmdt function failed\n");
        exit(EXIT_FAILURE);
    }
    exit(EXIT_SUCCESS);
}
```

Header

```
/* A common header file to describe the memory being shared. */
```

```
#define TEXT_SZ 256
struct shared_struct {
    int some_input;
    char some_text[TEXT_SZ];
};
```

Windows example: shared memory

Provider

```
/* This program is a provider of input text for the consumer. */
#include <windows.h>
#include <stdio.h>
3#include <stdlib.h>
#include <string.h>

#include "shm_com.h"

void main()
{
    void *shared_memory_loc = NULL;
    struct shared_struct *shared_stuff;
    char buffer[BUFSIZ];
    HANDLE hMapObject = NULL;  // handle to file mapping

    hMapObject = CreateFileMapping(
        INVALID_HANDLE_VALUE, // use paging file
        NULL,                 // no security attributes
        PAGE_READWRITE,       // read/write access
        0,                    // size: high 32-bits
        sizeof(struct shared_struct),            // size: low 32-bits
        "MMF1");     // name of map object
  if (hMapObject != NULL) {
    // Get a pointer to the file-mapped shared memory.
    shared_memory_loc = MapViewOfFile(
      hMapObject,     // object to map view of
      FILE_MAP_WRITE, // read/write access
      0,              // high offset:  map from
      0,              // low offset:   beginning
      0);             // default: map entire file
    if (shared_memory_loc == NULL) {
      CloseHandle(hMapObject);
      fprintf(stderr, "MapViewOfFile function failed\n");
      exit(EXIT_FAILURE);
    } else
      memset(shared_memory_loc, '\0', sizeof(struct shared_struct));
  } else {
        fprintf(stderr, "CreateMappedFile function failed\n");
        exit(EXIT_FAILURE);
    }

    printf("Memory attached at %X\n", (int)shared_memory_loc);
    shared_stuff = (struct shared_struct *)shared_memory_loc;
    while(1) {
        while(shared_stuff->some_input == 1) {
            Sleep(1000);
            printf("waiting for Consumer...\n");
        }
        printf("Enter some text: ");
        fgets(buffer, BUFSIZ, stdin);

        strncpy(shared_stuff->some_text, buffer, TEXT_SZ);
        shared_stuff->some_input = 1;

        if (strncmp(buffer, "done", 4) == 0) {
            break;
        }
    }

    if (!UnmapViewOfFile(shared_memory_loc)) {
        fprintf(stderr, "UnmapViewOfFile function failed\n");
        exit(EXIT_FAILURE);
    }

    CloseHandle(hMapObject);
    exit(EXIT_SUCCESS);
}
```

Consumer

```
/* This program is a consumer. The shared memory segment is
   created with a call to shmget, with the IPC_CREAT bit specified.
*/
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "shm_com.h"

void main()
{
    void *shared_memory_loc = (void *)0;
    struct shared_struct *shared_stuff;
    HANDLE hMapObject = NULL;  // handle to file mapping

    hMapObject = CreateFileMapping(
        INVALID_HANDLE_VALUE, // use paging file
        NULL,                 // no security attributes
        PAGE_READWRITE,       // read/write access
        0,                    // size: high 32-bits
        sizeof(struct shared_struct),            // size: low 32-bits
        "MMF1");     // name of map object
/* Make the shared memory accessible to the program. */
  if (hMapObject != NULL) {
    // Get a pointer to the file-mapped shared memory.
    shared_memory_loc = MapViewOfFile(
      hMapObject,     // object to map view of
      FILE_MAP_WRITE, // read/write access
      0,              // high offset:  map from
      0,              // low offset:   beginning
      0);             // default: map entire file
    if (shared_memory_loc == NULL) {
      CloseHandle(hMapObject);
      fprintf(stderr, "MapViewOfFile function failed\n");
      exit(EXIT_FAILURE);
    } else
      memset(shared_memory_loc, '\0', sizeof(struct shared_struct));
  } else {
        fprintf(stderr, "CreateMappedFile function failed\n");
        exit(EXIT_FAILURE);
    }
    printf("Memory attached at %X\n", (int)shared_memory_loc);

/* Assign the shared_memory_loc segment to shared_stuff.
   Echo any text in "some_text".
   Continues until end is found in "some_input" (1 in stored by
Provider).
*/
    shared_stuff = (struct shared_struct *)shared_memory_loc;
    shared_stuff->some_input = 0;
    while(1) {
        if (shared_stuff->some_input) {
            printf("You wrote: %s", shared_stuff->some_text);
            Sleep(1000 ); /* the Provider is waiting for this process
*/
            shared_stuff->some_input = 0;
            if (strncmp(shared_stuff->some_text, "done", 4) == 0) {
                break;
            }
        }
    }

/* Detach and Delete shared memory */
    if (!UnmapViewOfFile(shared_memory_loc)) {
        fprintf(stderr, "UnmapViewOfFile function failed\n");
        exit(EXIT_FAILURE);
    }
    CloseHandle(hMapObject);
    exit(EXIT_SUCCESS);
}
```

Header

```
/* A common header file to describe the memory being shared. */
#define TEXT_SZ 256
struct shared_struct {
    int some_input;
    char some_text[TEXT_SZ];
};
```

Appendix B: Limiting File I/O

The following code shows how file I/O can be limited in UNIX and Windows.

UNIX example: limiting file I/O

```
#include <unistd.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/resource.h>
#include <sys/time.h>
```

```
// The load function writes a string to a temporary file 1000 times
// Then performs arithmetic to generate a load on the CPU.
void load()
{
    FILE *f;
    int i;
    double x = 20.999, y = 10;

    f = tmpfile();
    for(i = 0; i < 1000; i++) {
        fprintf(f,"Perform some output\n");
        if(ferror(f)) {
            fprintf(stderr,"Error writing to a temporary file\n");
            exit(1);
        }
    }
    for(i = 0; i < 1000000; i++)
        y = cbrt(x + y);
}

int main()
{
    struct rusage usage;
    struct rlimit limits;
    int priority;

// Call load, then getrusage function to discover how much CPU time
was used.
    load();
    getrusage(RUSAGE_SELF, &usage);
    printf("CPU usage: User = %ld.%06ld, System = %ld.%06ld\n",
        usage.ru_utime.tv_sec, usage.ru_utime.tv_usec,
        usage.ru_stime.tv_sec, usage.ru_stime.tv_usec);

// Call getpriority & getrlimit for current priority and file size
limits
    priority = getpriority(PRIO_PROCESS, getpid());
    printf("Current priority = %d\n", priority);

    getrlimit(RLIMIT_FSIZE, &limits);
    printf("Current File Size limit: soft = %ld, hard = %ld\n",
        limits.rlim_cur, limits.rlim_max);

// Set a file size limit using setrlimit and call load again.
    limits.rlim_cur = 4096;
    limits.rlim_max = 8192;
    printf("Setting a 4K file size limit\n");
    setrlimit(RLIMIT_FSIZE, &limits);

// This call fails because it attempts to create a file too large.
    load();
    exit(0);
}
```

Windows example: limiting file I/O

Resource.h

```
#ifndef  __resource_h__
#define  __resource_h__

#include   <stdio.h>

#define RLIMIT_CPU      0   /* limit on CPU time per process */
#define RLIMIT_FSIZE    1   /* limit on file size */
#define RLIMIT_DATA     2   /* limit on data segment size */
#define RLIMIT_STACK    3   /* limit on process stack size */
#define RLIMIT_CORE     4   /* limit on size of core dump file */
#define RLIMIT_NOFILE   5   /* limit on number of open files */
#define RLIMIT_AS       6   /* limit on process total address space
size */
#define RLIMIT_VMEM     RLIMIT_AS

#define RLIM_NLIMITS    7




/*
 * process resource limits definitions
 */

struct rlimit {
//        LARGE_INTEGER  rlim_cur;
//        LARGE_INTEGER  rlim_max;
          __int64    rlim_cur;
          __int64    rlim_max;
};

typedef struct rlimit  rlimit_t;

/*
 * Prototypes
 */
int getrlimit(int resource, struct rlimit *);
int setrlimit(int resource, const struct rlimit *);

size_t rfwrite( const void *buffer, size_t size, size_t count, FILE
*stream );
int _rwrite( int handle, const void *buffer, unsigned int count );

//
// Following are the prototypes for the real functions...
//
// size_t fwrite( const void *buffer, size_t size, size_t count, FILE
*stream );
// int _write( int handle, const void *buffer, unsigned int count );

#endif
```

Resource.c

```
///////////////////////////////////////////////////////////////
//
// Sample implementation of getrlimit() and setrlimit() for Win32.
//
// Includes wrappers around fwrite() and _write() where the wrappers
// are resource limit aware.
//
//
///////////////////////////////////////////////////////////////

#include   <windows.h>
#include   "resource.h"
#include   <io.h>
#include   <errno.h>

static BOOL     rInitialized = FALSE;       // Indicates if the
      rlimit structure has been initialized

static rlimit_t rlimits[RLIM_NLIMITS];      // Resource limits array
      - on element for each limit we
                                            // keep track of.



///////////////////////////////////////////////////////////////
//
// InitializeRlimits()
//
// Sets the initial values in the rlimits arrar for the process.
//
///////////////////////////////////////////////////////////////
void InitializeRlimits()
{
    int    i;                               // Index variable
    //
    // Initialize the rlimits structure with 0 for the current value,
    // and 2^32 - 1 for the max.  This function could be modified
    // to read the initial values from...
    //     ...the registry...
    //     ...an environment variable...
    //     ...a disk file...
    //     ...other...
    // which would then be used to populate the rlimits structure.
    //
    for( i=0; i<RLIM_NLIMITS; i++ )
    {
        rlimits[i].rlim_cur = 0;
        rlimits[i].rlim_max = 0xffffffff;
    }
    rInitialized = TRUE;
}

/////////////////////////////////////////////////////////////////
// getrlimit()
//
// NOTE: Posix spec states function returns 0 on success and -1
//       when an error occurs and sets errno to the error code.
//       Currently, if an error occurs, the errno value is returned
//       rather than -1.  errno is not set.
//
/////////////////////////////////////////////////////////////////int
      getrlimit(int resource, struct rlimit *rlp)
{
    int   iRet = 0;                     // return value - assume
success
```

```
    //
    // If we have not initialized the limits yet, do so now
    //
    if( !rInitialized )
        InitializeRlimits();

    //
    // Check to make sure the resource value is within range
    //
    if( (resource < 0) || (resource >= RLIM_NLIMITS) )
    {
        iRet = EINVAL;
    }

    //
    // Return both rlim_cur and rlim_max
    //
    *rlp = rlimits[resource];

    return iRet;
}

//////////////////////////////////////////////////////////////////
// setrlimit()
//
// NOTE: Posix spec states function returns 0 on success and -1
//       when an error occurs and sets errno to the error code.
//       Currently, if an error occurs, the errno value is returned
//       rather than -1.  errno is not set.
//
//////////////////////////////////////////////////////////////////int
      setrlimit(int resource, const struct rlimit *rlp)
{
    int    iRet = 0;                     // return value - assume
success

    if( !rInitialized )
        InitializeRlimits();
    //
    // Check to make sure the resource value is within range
    //
    if( (resource < 0) || (resource >= RLIM_NLIMITS) )
    {
        iRet = EINVAL;
    }
    //
    // Only change the current limit - do not change the max limit.
    // We could pick some NT privilege, which if the user held, we
    // would allow the changing of rlim_max.
    //
    if( rlp->rlim_cur < rlimits[resource].rlim_max )
        rlimits[resource].rlim_cur = rlp->rlim_cur;
    else
        iRet = EINVAL;
    //
    // We should not let the user set the max value.  However,
    // since currently there is no defined source for initial
    // values, we will let the user change the max value.
    //
    rlimits[resource].rlim_max = rlp->rlim_max;

    return iRet;
}

/////////////////////////////////////////////////////////////////////
// Wrap the real fwrite() with this rfwrite() function, which is
// resource limit aware.
//
//
/////////////////////////////////////////////////////////////////////size_t
      rfwrite( const void *buffer, size_t size, size_t count, FILE
      *stream )
{
    long            position;
    size_t          written;
    __int64         liByteCount,
                    liPosition;
    //
    // Convert the count to a large integer (64 bit integer)
    //
    liByteCount = (__int64)count;

    //
    // Get the current file position
    //
    position = ftell( stream );
    liPosition = (__int64)position;

    //
    // Check to make sure the write will not exceed the RLIMIT_FSIZE
limit.
    //
    if ( (liPosition + liByteCount) > rlimits[RLIMIT_FSIZE].rlim_cur
)
    {
        //
        // report an error
        //
        written = 0;

    }
    else
    {
        //
        // Do the actual write the user requested
        //
        written = fwrite( buffer, size, count, stream );
    }
    return written;
}

/////////////////////////////////////////////////////////////////////
// Wrap the real _write() function with the _rwrite() function
// which is resource aware.
//
//
/////////////////////////////////////////////////////////////////////int
      _rwrite( int handle, const void *buffer, unsigned int count )
{
    long            position;
    DWORD           dwWritten;
    _int64          liByteCount,
                    liPosition;
    //
    // Convert the count to a large integer
    //
    liByteCount = (__int64)count;

    //
    // Get the Current file position
    //
    position = _tell( handle );
    liPosition = (__int64)position;

    //
    // Check to make sure the write will not exceed the RLIMIT_FSIZE
limit.
    //
    if ( (liPosition + liByteCount) > rlimits[RLIMIT_FSIZE].rlim_cur
)
    {
        //
        // report an error
        //
        dwWritten = 0;
    }
    else
    {
        //
        // Do the actual write the user requested
        //
        dwWritten = _write( handle, buffer, count );
    }
    return dwWritten;
}
```

RLimit.c

```
//////////////////////////////////////////////////////////////////
// Test program to test the rlimit functions
//
//
//
////////////////////////////////////////////////////////////////
#include "resource.h"

int main()
{
    int        i;
    rlimit_t   limits,
               lim;
    FILE       *fh;

    //
    // First set a limit
    //
    limits.rlim_cur = 1000;
    limits.rlim_max = 5000;
    setrlimit( RLIMIT_FSIZE, &limits );

    //
    // Now read and display the limit
    //
    getrlimit( RLIMIT_FSIZE, &lim );
    printf( "\nLimits\n  cur = %I64d.\n  max = %I64d.\n",
      lim.rlim_cur, lim.rlim_max );

    //
    // Now create a file and start writing to it...
    //
    fh = fopen( "a.a", "w" );
    for( i=0; i<10000; i++ )
        if( !rfwrite("a", 1, 1, fh) )
            break;

    printf( "\nBroke out of write loop with i = %d.\n", i );

  return 0;
}
```

Appendix C: Creating a Thread in Windows

The following example application is a simple Windows application with a title bar, menu, and list window. The **Test** menu contains four items: **Semaphore**, **Mutex**, **Critical Section**, and **Event**. You can exercise multiple executions of a given test simultaneously by selecting the test type repeatedly. Keep in mind that the tests are only guaranteed to guard against overwriting the shared memory area within a single type of test. Invoking two different test types simultaneously will not yield a synchronized result. Moreover, running multiple instances of the application and invoking the same test in both instances will yield correct results for all synchronization mechanisms except the critical section test.

This is because the example uses the **WaitForSingleObject** function. Perhaps a better solution would be to use the **WaitForMultipleObjects**, but this is only an example of what you might do. As the designer and implementer, you may want to do otherwise after considering all of the variants of the **WaitFor** function.

stdafx.h

```
// stdafx.h : include file for standard system include files,
// or project specific include files that are used frequently, but
// are changed infrequently
//
#pragma once

#define WIN32_LEAN_AND_MEAN  // Exclude rarely-used stuff from
      Windows headers
// Windows Header Files:
#include <windows.h>
// C RunTime Header Files
#include <stdlib.h>
#include <malloc.h>
#include <memory.h>
#include <tchar.h>
// TODO: reference additional headers your program requires here
```

resource.h

```
//{{NO_DEPENDENCIES}}
// Microsoft Visual C++ generated include file.
// Used by Named Shared Memory.rc
//
#define IDC_MYICON                      2
#define IDD_NAMEDSHAREDMEMORY_DIALOG    102
#define IDS_APP_TITLE                   103
#define IDD_ABOUTBOX                    103
#define IDM_ABOUT                       104
#define IDM_EXIT                        106
#define IDI_NAMEDSHAREDMEMORY           107
#define IDC_NAMEDSHAREDMEMORY           109
#define IDM_SEM                         110
#define IDM_MUT                         111
#define IDM_CRS                         112
#define IDM_EVT                         113
#define IDR_MAINFRAME                   128
#define ID_TEST_IDM                     129
#define ID_TEST_IDM130                  130
#define IDC_STATIC                      -1

// Next default values for new objects
//
#ifdef APSTUDIO_INVOKED
#ifndef APSTUDIO_READONLY_SYMBOLS
#define _APS_NO_MFC                     1
#define _APS_NEXT_RESOURCE_VALUE        131
#define _APS_NEXT_COMMAND_VALUE         32771
#define _APS_NEXT_CONTROL_VALUE         1000
#define _APS_NEXT_SYMED_VALUE           110
#endif
#endif
```

Named shared memory.rc script

```
// Microsoft Visual C++ generated resource script.
//
#include "resource.h"

#define APSTUDIO_READONLY_SYMBOLS
/////////////////////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 2 resource.
//
#define APSTUDIO_HIDDEN_SYMBOLS
#include "windows.h"
#undef APSTUDIO_HIDDEN_SYMBOLS

/////////////////////////////////////////////////////////////////////////////
#undef APSTUDIO_READONLY_SYMBOLS

/////////////////////////////////////////////////////////////////////////////
// English (U.S.) resources

#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_ENU)
#ifdef _WIN32
LANGUAGE LANG_ENGLISH, SUBLANG_ENGLISH_US
```

```
#pragma code_page(1252)
#endif //_WIN32

/////////////////////////////////////////////////////////////////////////////
//
// Icon
//

// Icon with lowest ID value placed first to ensure application icon
// remains consistent on all systems.
IDI_NAMEDSHAREDMEMORY   ICON
  "NamedSharedMemory.ico"

/////////////////////////////////////////////////////////////////////////////
//
// Menu
//

IDC_NAMEDSHAREDMEMORY MENU
BEGIN
    POPUP "&File"
    BEGIN
        MENUITEM "E&xit",                      IDM_EXIT
    END
    MENUITEM SEPARATOR
    POPUP "&Test"
    BEGIN
        MENUITEM "&Semephore",                 IDM_SEM
        MENUITEM "&Mutex",                     IDM_MUT
        MENUITEM "&CriticalSection",           IDM_CRS
        MENUITEM "&Event",                     IDM_EVT
    END
    MENUITEM SEPARATOR
    POPUP "&Help"
    BEGIN
        MENUITEM "&About ...",                 IDM_ABOUT
    END
END

/////////////////////////////////////////////////////////////////////////////
//
// Accelerator
//

IDC_NAMEDSHAREDMEMORY ACCELERATORS
BEGIN
    "?",            IDM_ABOUT,              ASCII,  ALT
    "/",            IDM_ABOUT,              ASCII,  ALT
END

/////////////////////////////////////////////////////////////////////////////
//
// Dialog
//

IDD_ABOUTBOX DIALOG  22, 17, 230, 75
STYLE DS_SETFONT | DS_MODALFRAME | WS_CAPTION | WS_SYSMENU
CAPTION "About"
FONT 8, "System"
BEGIN
```

```
    ICON            IDI_NAMEDSHAREDMEMORY,IDC_MYICON,14,9,16,16
    LTEXT           "Named Shared Memory Version
      1.0",IDC_STATIC,49,10,119,8,
                    SS_NOPREFIX
    LTEXT           "Copyright (C) 2002",IDC_STATIC,49,20,119,8
    DEFPUSHBUTTON   "OK",IDOK,195,6,30,11,WS_GROUP
END

#ifdef APSTUDIO_INVOKED
/////////////////////////////////////////////////////////////////////
//
// TEXTINCLUDE
//

1 TEXTINCLUDE
BEGIN
    "resource.h\0"
END

2 TEXTINCLUDE
BEGIN
    "#define APSTUDIO_HIDDEN_SYMBOLS\r\n"
    "#include ""windows.h""\r\n"
    "#undef APSTUDIO_HIDDEN_SYMBOLS\r\n"
    "\0"
END

3 TEXTINCLUDE
BEGIN
    "\r\n"
    "\0"
END

#endif    // APSTUDIO_INVOKED

/////////////////////////////////////////////////////////////////////
//
// String Table
//

STRINGTABLE
BEGIN
    IDS_APP_TITLE           "Named Shared Memory"
    IDC_NAMEDSHAREDMEMORY   "NAMEDSHAREDMEMORY"
END

#endif    // English (U.S.) resources
/////////////////////////////////////////////////////////////////////

#ifndef APSTUDIO_INVOKED
/////////////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 3 resource.
//

/////////////////////////////////////////////////////////////////////
#endif    // not APSTUDIO_INVOKED
```

Named shared Memory.ico

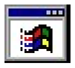


Figure 4.

Named shared Memory.cpp

```
#include "stdafx.h"
#include "resource.h"
#define MAX_LOADSTRING 100
#define SHMEMSIZE 4096
#define ELEMENTS(x) (sizeof(x) / sizeof(x[0]))

static LPVOID lpvMem = NULL; // pointer to shared memory

// Global Variables:
HINSTANCE hInst;                // current instance
TCHAR szTitle[MAX_LOADSTRING];          // The title bar text
TCHAR szWindowClass[MAX_LOADSTRING];      // The main window class
name
LPCTSTR lpszSemaphore = "SEMAPHORE-EXAMPLE";
LPCTSTR lpszMutex = "MUTEX-EXAMPLE";
LPCTSTR lpszEvent = "EVENT-EXAMPLE";

// Forward declarations of functions included in this code module:
ATOM        MyRegisterClass(HINSTANCE hInstance);
BOOL        InitInstance(HINSTANCE, int);
LRESULT CALLBACK  WndProc(HWND, UINT, WPARAM, LPARAM);
LRESULT CALLBACK  About(HWND, UINT, WPARAM, LPARAM);

int APIENTRY _tWinMain(HINSTANCE hInstance,
                     HINSTANCE hPrevInstance,
                     LPSTR     lpCmdLine,
                     int       nCmdShow)
{
   // TODO: Place code here.
  MSG msg;
  HACCEL hAccelTable;

  // Initialize global strings
  LoadString(hInstance, IDS_APP_TITLE, szTitle, MAX_LOADSTRING);
  LoadString(hInstance, IDC_NAMEDSHAREDMEMORY, szWindowClass,
      MAX_LOADSTRING);
  MyRegisterClass(hInstance);

  // Perform application initialization:
  if (!InitInstance (hInstance, nCmdShow))
  {
    return FALSE;
  }

  hAccelTable = LoadAccelerators(hInstance,
      (LPCTSTR)IDC_NAMEDSHAREDMEMORY);

  // Main message loop:
  while (GetMessage(&msg, NULL, 0, 0))
  {
    if (!TranslateAccelerator(msg.hwnd, hAccelTable, &msg))
    {
      TranslateMessage(&msg);
      DispatchMessage(&msg);
    }
  }
  return (int) msg.wParam;
}

//
//  FUNCTION: MyRegisterClass()
//
//  PURPOSE: Registers the window class.
//
//  COMMENTS:
//
//    This function and its usage are only necessary if you want this
code
//    to be compatible with Win32 systems prior to the
      'RegisterClassEx'
//    function that was added to Microsoft Windows 95. It is
        important to call this function
//    so that the application will get 'well formed' small icons
        associated
//    with it.
//
ATOM MyRegisterClass(HINSTANCE hInstance)
{
  WNDCLASSEX wcex;

  wcex.cbSize = sizeof(WNDCLASSEX);

  wcex.style      = CS_HREDRAW | CS_VREDRAW;
  wcex.lpfnWndProc  = (WNDPROC)WndProc;
  wcex.cbClsExtra    = 0;
  wcex.cbWndExtra    = 0;
  wcex.hInstance    = hInstance;
  wcex.hIcon      = LoadIcon(hInstance,
    (LPCTSTR)IDI_NAMEDSHAREDMEMORY);
  wcex.hCursor    = LoadCursor(NULL, IDC_ARROW);
  wcex.hbrBackground  = (HBRUSH)(COLOR_WINDOW+1);
  wcex.lpszMenuName  = (LPCTSTR)IDC_NAMEDSHAREDMEMORY;
  wcex.lpszClassName  = szWindowClass;
  wcex.hIconSm    = LoadIcon(wcex.hInstance,
    (LPCTSTR)IDI_NAMEDSHAREDMEMORY);

  return RegisterClassEx(&wcex);
}
//
//   FUNCTION: InitInstance(HANDLE, int)
//
//   PURPOSE: Saves instance handle and creates main window
//
//   COMMENTS:
//
//        In this function, we save the instance handle in a global
      variable and
//        create and display the main program window.
//
BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
{
   HWND hWnd;
   hInst = hInstance; // Store instance handle in our global variable

   hWnd = CreateWindow(szWindowClass, szTitle, WS_OVERLAPPEDWINDOW,
      CW_USEDEFAULT, 0, CW_USEDEFAULT, 0, NULL, NULL, hInstance,
NULL);

   if (!hWnd)
   {
      return FALSE;
   }

   ShowWindow(hWnd, nCmdShow);
   UpdateWindow(hWnd);

   return TRUE;
}

//
//  FUNCTION: SemaphoreChildThreadProc( LPVOID lpData )
//
//  PURPOSE:  Child thread procedure that waits for a semaphore.
//
// Holds the semaphore for five seconds, and then releases the
semaphore.
// Threads that cannot obtain semaphores will wait.
//

DWORD WINAPI SemaphoreChildThreadProc( LPVOID lpData )
{
   TCHAR  szBuffer[256];
   LONG   dwSemCount  = 0;
   HWND   hWnd       = (HWND)lpData;
   HANDLE hSemaphore = OpenSemaphore( SYNCHRONIZE |
     SEMAPHORE_MODIFY_STATE, FALSE, lpszSemaphore );

   wsprintf( szBuffer,"Thread %x waiting for semaphore %x",
             GetCurrentThreadId(), hSemaphore );

   SendMessage( hWnd, LB_INSERTSTRING, (WPARAM)-1, (LPARAM)szBuffer
);

   // Wait for signaled semaphore.
   WaitForSingleObject( hSemaphore, INFINITE );
   wsprintf( szBuffer,"Thread %x got semaphore", GetCurrentThreadId()
);
   SendMessage( hWnd, LB_INSERTSTRING, (WPARAM)-1, (LPARAM)szBuffer
);

   // Plug string into shared memory segment
   wsprintf( (LPTSTR) lpvMem,"Thread %x Shared memory content",
     GetCurrentThreadId());

   // Shut out other threads.
   Sleep( 5000 );

   // Display contents of shared memory region.
   wsprintf( szBuffer,"Thread %x Shared memory content is %s",
GetCurrentThreadId(), (LPTSTR) lpvMem);
   SendMessage( hWnd, LB_INSERTSTRING, (WPARAM)-1, (LPARAM) szBuffer
);

   // Release semaphore.
   ReleaseSemaphore( hSemaphore, 1, &dwSemCount );

   wsprintf( szBuffer,"Thread %x is done with semaphore. Its count
     was %ld.",
             GetCurrentThreadId(), dwSemCount );

   SendMessage( hWnd, LB_INSERTSTRING, (WPARAM)-1, (LPARAM)szBuffer
);

   CloseHandle( hSemaphore );

   return( 0 );
}

HANDLE hMutex = NULL;

//
//  FUNCTION: MutexChildThreadProc( LPVOID lpData )
//
//  PURPOSE:  Child thread procedure that waits for a mutex.
//
// Thread procedure waits until mutex becomes signaled,
// holds the object for five seconds, and then releases it.
//
DWORD WINAPI MutexChildThreadProc( LPVOID lpData )
{
   TCHAR  szBuffer[128];
   HWND   hWnd      = (HWND)lpData;
   HANDLE hTmpMutex = OpenMutex( MUTEX_ALL_ACCESS, FALSE, lpszMutex
);

   wsprintf( szBuffer,"Thread %x waiting for Mutex %x",
             GetCurrentThreadId(), hMutex );
   SendMessage( hWnd, LB_INSERTSTRING, (WPARAM)-1, (LPARAM)szBuffer
);

   // Wait for signaled mutex.
   WaitForSingleObject( hMutex, INFINITE );
   wsprintf( szBuffer,"Thread %x got mutex!", GetCurrentThreadId() );
   SendMessage( hWnd, LB_INSERTSTRING, (WPARAM)-1, (LPARAM)szBuffer
);

   __try {
  // Plug string into shared memory segment
  wsprintf( (LPTSTR) lpvMem,"Thread %x Shared memory content",
    GetCurrentThreadId());

  // Shut out other threads.
  Sleep( 5000 );

  // Display shared memory region
  wsprintf( szBuffer,"Thread %x Shared memory content is %s",
    GetCurrentThreadId(), (LPTSTR) lpvMem);
  SendMessage( hWnd, LB_INSERTSTRING, (WPARAM)-1, (LPARAM) szBuffer
);
   }
   __finally {
  // Release mutex.
  wsprintf( szBuffer,"Thread %x is done with mutex",
    GetCurrentThreadId() );
  SendMessage( hWnd, LB_INSERTSTRING, (WPARAM)-1, (LPARAM) szBuffer
);
  ReleaseMutex( hMutex );
   }

   return( 0 );
}


//
//  FUNCTION: CRSChildThreadProc( LPVOID lpData )
//
//  PURPOSE:  Child thread procedure wrapped in Critical Section.
//
// Thread procedure executes the shared memory piece in a critical
section,
// and holds the object for five seconds before leaving the critical
section.
//
CRITICAL_SECTION cs;

DWORD WINAPI CRSChildThreadProc( LPVOID lpdwData )
{
   TCHAR szBuffer[256];
   HWND  hWnd  = (HWND)lpdwData;

   wsprintf( szBuffer, "Thread %x waiting for critical section",
     GetCurrentThreadId() );
   SendMessage( hWnd, LB_INSERTSTRING, (WPARAM)-1, (LPARAM)szBuffer
);

   // Beginning of protected section.
   __try {
  EnterCriticalSection( &cs );
  wsprintf( szBuffer,"Thread %x in critical section",
    GetCurrentThreadId() );
  SendMessage( hWnd, LB_INSERTSTRING, (WPARAM)-1, (LPARAM)szBuffer );

  // Critical code goes here.
  // Plug string into shared memory segment
  wsprintf( (LPTSTR) lpvMem,"Thread %x Shared memory content",
    GetCurrentThreadId());

  // Shut out other threads.
  Sleep( 5000 );

  // Display shared memory region
  wsprintf( szBuffer,"Thread %x Shared memory content is %s",
    GetCurrentThreadId(), (LPTSTR) lpvMem);
  SendMessage( hWnd, LB_INSERTSTRING, (WPARAM)-1, (LPARAM) szBuffer
);
   }

   __finally {
  // End of protected section
  LeaveCriticalSection( &cs );
  wsprintf( szBuffer,"Thread %x has exited critical section",
    GetCurrentThreadId() );
  SendMessage( hWnd, LB_INSERTSTRING, (WPARAM)-1, (LPARAM)szBuffer );
   }
   return( 0 );
}

//
//  FUNCTION: EventChildThreadProc( LPVOID lpData )
//
//  PURPOSE:  Child thread procedure which uses Events.
//
// Thread procedure executes the shared memory piece in an event
section,
// and holds the object for five seconds before leaving the event
section.
//

DWORD WINAPI EventChildThreadProc( LPVOID lpdwData )
{
   TCHAR szBuffer[256];
   HWND  hWnd  = (HWND)lpdwData;
   HANDLE hEvent = OpenEvent( SYNCHRONIZE, FALSE, lpszEvent );

   wsprintf( szBuffer, "Thread %x waiting for event %x",
     GetCurrentThreadId(), hEvent );
   SendMessage( hWnd, LB_INSERTSTRING, (WPARAM)-1, (LPARAM)szBuffer
);

   // Wait for event.
   WaitForSingleObject( hEvent, INFINITE );
   wsprintf( szBuffer,"Thread %x has received event",
     GetCurrentThreadId() );
   SendMessage( hWnd, LB_INSERTSTRING, (WPARAM)-1, (LPARAM)szBuffer
);

   __try {
  // Plug string into shared memory segment
  wsprintf( (LPTSTR) lpvMem,"Thread %x Shared memory content",
    GetCurrentThreadId());

  // Shut out other threads.
  Sleep( 5000 );

  // Display shared memory region
  wsprintf( szBuffer,"Thread %x Shared memory content is %s",
    GetCurrentThreadId(), (LPTSTR) lpvMem);
  SendMessage( hWnd, LB_INSERTSTRING, (WPARAM)-1, (LPARAM) szBuffer
);
   }

   __finally {
  // Release event.
  wsprintf( szBuffer,"Thread %x is done with event",
    GetCurrentThreadId() );
  SendMessage( hWnd, LB_INSERTSTRING, (WPARAM)-1, (LPARAM) szBuffer
);
  SetEvent( hEvent );
  CloseHandle( hEvent );
   }
   return( 0 );
}

//
//  FUNCTION: WndProc(HWND, unsigned, WORD, LONG)
//
//  PURPOSE:  Processes messages for the main window.
//
//  WM_COMMAND  - process the application menu
//  WM_PAINT  - Paint the main window
//  WM_DESTROY  - post a quit message and return
//
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam,
  LPARAM lParam)
{
  int wmId, wmEvent;
  PAINTSTRUCT ps;
  HDC hdc;
  static HWND   hList = NULL;
  static HANDLE hSemaphore = NULL;
    static HANDLE hEvent = NULL;

    HANDLE hMapObject = NULL;  // handle to file mapping

  switch (message)
  {
  case WM_CREATE:
    // Create list box.
    hList  = CreateWindowEx( WS_EX_CLIENTEDGE, "LISTBOX", "",
                LBS_STANDARD | LBS_NOINTEGRALHEIGHT |
                WS_CHILD | WS_VISIBLE,
                0, 0, 10, 10,
                hWnd, (HMENU)101,
                hInst, NULL );
            hMapObject = CreateFileMapping(
                INVALID_HANDLE_VALUE, // use paging file
                NULL,                 // no security attributes
                PAGE_READWRITE,       // read/write access
                0,                    // size: high 32-bits
                SHMEMSIZE,            // size: low 32-bits
                "dllmemfilemap");     // name of map object
      if (hMapObject != NULL) {
         // Get a pointer to the file-mapped shared memory.
        lpvMem = MapViewOfFile(
          hMapObject,     // object to map view of
          FILE_MAP_WRITE, // read/write access
          0,              // high offset:  map from
          0,              // low offset:   beginning
          0);             // default: map entire file
        if (lpvMem == NULL) {
          CloseHandle(hMapObject);
        } else
          memset(lpvMem, '\0', SHMEMSIZE);
      }

    // Initialize Semaphore object.
    hSemaphore = CreateSemaphore( NULL, 1, 1, lpszSemaphore );

    // Initialize Mutex object.
```

```
        hMutex = CreateMutex( NULL, FALSE, lpszMutex );

    // Initialize the critical section object.
    InitializeCriticalSection( &cs);

    // Initialize the event object.
    hEvent = CreateEvent( NULL, FALSE, TRUE, lpszEvent );

    break;
  case WM_SIZE :
        if ( wParam != SIZE_MINIMIZED )
            MoveWindow( hList, 0, 0, LOWORD( lParam ), HIWORD( lParam
              ), TRUE );
        break;
  case WM_COMMAND:
    wmId    = LOWORD(wParam);
    wmEvent = HIWORD(wParam);
    // Parse the menu selections:
    switch (wmId)
    {
    case IDM_SEM:
      {
            DWORD dwChildId;

            CreateThread( NULL, 0, SemaphoreChildThreadProc, hList,
              0, &dwChildId );
      }
        break;
    case IDM_MUT:
      {
            DWORD dwChildId;

            CreateThread( NULL, 0, MutexChildThreadProc, hList, 0,
              &dwChildId );
      }
        break;
    case IDM_CRS:
      {
            DWORD dwChildId;

            CreateThread( NULL, 0, CRSChildThreadProc, hList, 0,
              &dwChildId );
      }
        break;
    case IDM_EVT:
      {
            DWORD dwChildId;

            CreateThread( NULL, 0, EventChildThreadProc, hList, 0,
              &dwChildId );
      }
        break;
    case IDM_ABOUT:
      DialogBox(hInst, (LPCTSTR)IDD_ABOUTBOX, hWnd, (DLGPROC)About);
      break;
    case IDM_EXIT:
      DestroyWindow(hWnd);
      break;
    default:
      return DefWindowProc(hWnd, message, wParam, lParam);
    }
    break;
  case WM_PAINT:
    hdc = BeginPaint(hWnd, &ps);
    // TODO: Add any drawing code here...
    EndPaint(hWnd, &ps);
    break;
  case WM_DESTROY:
    // Close semaphore object
        if ( hSemaphore )
      CloseHandle( hSemaphore );

    // Close mutex object
    if ( hMutex )
            CloseHandle( hMutex );

    // Release resources used by the critical section object.
    DeleteCriticalSection( &cs );

    // Close event object
    if ( hEvent )
      CloseHandle( hEvent );

    // Unmap shared memory from the process's address space
    UnmapViewOfFile(lpvMem);

        // Close the process's handle to the file-mapping object.
        CloseHandle(hMapObject);

    PostQuitMessage(0);
    break;
  default:
    return DefWindowProc(hWnd, message, wParam, lParam);
  }
  return 0;
}

// Message handler for about box.
LRESULT CALLBACK About(HWND hDlg, UINT message, WPARAM wParam, LPARAM
lParam)
{
  switch (message)
  {
  case WM_INITDIALOG:
    return TRUE;

  case WM_COMMAND:
    if (LOWORD(wParam) == IDOK || LOWORD(wParam) == IDCANCEL)
    {
      EndDialog(hDlg, LOWORD(wParam));
      return TRUE;
    }
    break;
  }
  return FALSE;
}
```

Appendix D: Process Pipes

The following is another example of how process pipes are used in UNIX code. The migrated code for W2in32 is shown immediately following the UNIX code.

UNIX example: process pipes

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main()
{
    FILE *read_fp;
    char buffer[BUFSIZ + 1];
    int chars_read;

    memset(buffer, '\0', sizeof(buffer));
    read_fp = popen("ps -l", "r");
    if (read_fp != NULL) {
        chars_read = fread(buffer, sizeof(char), BUFSIZ, read_fp);
        while (chars_read > 0) {
            buffer[chars_read - 1] = '\0';
            printf("Reading:-\n %s\n", buffer);
            chars_read = fread(buffer, sizeof(char), BUFSIZ,
              read_fp);
        }
        pclose(read_fp);
        exit(EXIT_SUCCESS);
    }
    exit(EXIT_FAILURE);
}
```

Win32 example: process pipes

```
#include <windows.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main()
{
    FILE *read_fp;
    char buffer[BUFSIZ + 1];
    size_t chars_read;

    memset(buffer, '\0', sizeof(buffer));
    read_fp = _popen("ps -l", "r");
    if (read_fp != NULL) {
        chars_read = fread(buffer, sizeof(char), BUFSIZ, read_fp);
        while (chars_read > 0) {
            buffer[chars_read - 1] = '\0';
            printf("Reading:-\n %s\n", buffer);
            chars_read = fread(buffer, sizeof(char), BUFSIZ,
read_fp);
        }
        _pclose(read_fp);
        exit(EXIT_SUCCESS);
    }
    exit(EXIT_FAILURE);
}
```

Appendix E: Pipes across a Process

In this example, two separate processes are created. The first process represents the provider and is responsible for creating the pipe and spawning the consumer. The consumer accepts the single command line argument that represents the file number for the read end of the pipe. After the consumer is started, the provider writes a message on the write end of the pipe, and then waits for the consumer process to terminate.

Since this example is spawning a separate process, the Win32 version does not use **fork** or **exec** to spawn the consumer. The **spawnlp** function is used in place of **fork** or **exec** in Win32. In addition to the difference in the way the consumer process is spawned and waited on, you should notice a difference in the set of required header files and a preceding underscore for the pipe function.

As you can see, the differences between these two implementations are trivial.

UNIX example: pipes across a process

Provider

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main()
{
    int data_processed;
    int file_pipes[2];
    const char some_data[] = "123";
    char buffer[BUFSIZ + 1];
    pid_t pid;
    int tstat;


    memset(buffer, '\0', sizeof(buffer));

    if (pipe(file_pipes) == 0) {
        pid = fork();
        if (pid == (pid_t)-1) {
            fprintf(stderr, "Fork failure");
            exit(EXIT_FAILURE);
        }

        if (pid == 0) {
            sprintf(buffer, "%d", file_pipes[0]);
            (void)execl("Consumer", "Consumer", buffer, (char *)0);
            exit(EXIT_FAILURE);
        }
        else {
            data_processed = write(file_pipes[1], some_data,
                                   strlen(some_data));
            printf("%d - wrote %d bytes\n", getpid(),
data_processed);
            waitpid (pid, &tstat, 0);


            if (tstat & 0x0)
                printf("Consumer failed\n");
        }
    }
    exit(EXIT_SUCCESS);
}
```

Consumer

```
// The 'consumer' program, pipe4.c, that reads the data is much
simpler.

#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[])
{
    int data_processed;
    char buffer[BUFSIZ + 1];
    int file_descriptor;

    memset(buffer, '\0', sizeof(buffer));
    sscanf(argv[1], "%d", &file_descriptor);
    data_processed = read(file_descriptor, buffer, BUFSIZ);

    printf("%d - read %d bytes: %s\n", getpid(), data_processed,
buffer);
    exit(EXIT_SUCCESS);
}
```

Win32 example: pipes across a process

Provider

```
#include <windows.h>
#include <stdlib.h>
#include <stdio.h>
#include <io.h>
#include <fcntl.h>
#include <process.h>
#include <string.h>

int main()
{
    size_t data_processed;
    int file_pipes[2];
    const char some_data[] = "123";
    char buffer[BUFSIZ + 1];
    size_t pid;
    int tstat;

    memset(buffer, '\0', sizeof(buffer));
    if (_pipe(file_pipes, 32, O_BINARY) == 0) {
        sprintf(buffer, "%d", file_pipes[0]);
        if ((pid = _spawnlp(P_NOWAIT, "..\\Consumer.exe", "Consumer",
          buffer, NULL)) < 0) {
        exit(EXIT_FAILURE);
      } else {
            data_processed = write(file_pipes[1], some_data,
                                 (UINT)strlen(some_data));
            printf("%d - wrote %d bytes\n", getpid(),
              data_processed);

    // Wait until spawned program is done processing.
            _cwait(&tstat, pid, WAIT_CHILD);
           if(tstat & 0x0)
               printf("Consumer failed\n");
        }
    }
    exit(EXIT_SUCCESS);
}
```

Consumer

```
#include <windows.h>
#include <stdlib.h>
#include <stdio.h>
#include <io.h>
#include <fcntl.h>
#include <process.h>
#include <string.h>

int main(int argc, char *argv[])
{
    size_t data_processed;
    char buffer[BUFSIZ + 1];
    int file_descriptor;

    memset(buffer, '\0', sizeof(buffer));
    sscanf(argv[1], "%d", &file_descriptor);
    data_processed = read(file_descriptor, buffer, BUFSIZ);

    printf("%d - read %d bytes: %s\n", getpid(), data_processed,
buffer);
    exit(EXIT_SUCCESS);
}
```

Appendix F: Pipes Used as Standard Input/Output

This example reuses the provider and consumer samples seen in the previous section. The examples in this section differ in two important ways.

First, the provider has been modified so that the read end of the pipe is duplicated and established as the STDIN device for the consumer, and the consumer is modified to simply use STDIN as the input device rather than the file handle passed as an argument to **main()**.

Second, the Win32 example in the previous section used **_spawnlp** to spawn the Consumer process. For the purposes of demonstrating the **CreateProcess** as a viable alternative, it is used in the Windows example. As part of this alternative approach, the Win32 API functions are used for creating the pipe and duplicating the handles instead of the standard C runtime library equivalents.

UNIX example: pipes used as standard input/output

Provider

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main()
{
    int data_processed;
    int file_pipes[2];
    const char some_data[] = "123";
    char buffer[BUFSIZ + 1];
    pid_t pid;
    int tstat;


    memset(buffer, '\0', sizeof(buffer));

    if (pipe(file_pipes) == 0) {
        pid = fork();
        if (pid == (pid_t)-1) {
            fprintf(stderr, "Fork failure");
            exit(EXIT_FAILURE);
        }

        if (pid == 0) {
            close(0);
            dup2(file_pipes[0], 0);
            close(file_pipes[0]);
            close(file_pipes[1]);

            (void)execl("Consumer", "Consumer", (char *)0);
            exit(EXIT_FAILURE);
        }
        else {
            close(file_pipes[0]);
            data_processed = write(file_pipes[1], some_data,
                                   strlen(some_data));
            close(file_pipes[1]);
            printf("%d - wrote %d bytes\n", getpid(),
              data_processed);
            waitpid (pid, &tstat, 0);


            if (tstat & 0x0)
                printf("Consumer failed\n");
        }
    }
    exit(EXIT_SUCCESS);
}
```

Consumer

```
// The 'consumer' program, pipe4.c, that reads the data is much
simpler.

#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main()
{
    int data_processed;
    char buffer[BUFSIZ + 1];

    memset(buffer, '\0', sizeof(buffer));
    data_processed = fread(buffer, 1, 3, stdin);

    printf("%d - read %d bytes: %s\n", getpid(), data_processed,
buffer);
    exit(EXIT_SUCCESS);
}
```

Win32 example: pipes used as standard input/output

Provider

```
#include <windows.h>
#include <stdlib.h>
#include <stdio.h>
#include <io.h>
#include <fcntl.h>
#include <process.h>
#include <string.h>

int main()
{
    DWORD data_processed;
    HANDLE file_pipes[2];
    const char some_data[] = "123\n";
    DWORD tstat;
    SECURITY_ATTRIBUTES sa;
    STARTUPINFO    si;
    PROCESS_INFORMATION  pi;

    sa.nLength = sizeof(SECURITY_ATTRIBUTES);      // Size of struct
    sa.lpSecurityDescriptor = NULL;                // Default
descriptor
    sa.bInheritHandle = TRUE;                     // Inheritable
    // Create the pipe
    if (CreatePipe(&file_pipes[0], &file_pipes[1], &sa, 0)) {

        // Create duplicate of write end so that original
        if (!DuplicateHandle(
                    GetCurrentProcess(),
                    file_pipes[0],           // Original handle
                    GetCurrentProcess(),
                    NULL,                    // don't create new handle
                    0,
                    FALSE,                   // Not inheritable
                    DUPLICATE_SAME_ACCESS)) {
```

```
            CloseHandle(file_pipes[0]);
            CloseHandle(file_pipes[1]);
             exit(EXIT_FAILURE);
        }

        // Now populate startup info for CreateProcess
        GetStartupInfo(&si);
        si.dwFlags    = STARTF_USESTDHANDLES;
        si.hStdInput  = file_pipes[0];
        si.hStdOutput = GetStdHandle(STD_OUTPUT_HANDLE);
        si.hStdError  = GetStdHandle(STD_ERROR_HANDLE);

        // Spawn the Consumer process
        if (!CreateProcess(
          "C:\\users\\user1\\PAG\\ProcessPipe\\Debug\\Consumer.exe",
                NULL,            // command line
                NULL,            // process security
                NULL,            // thread security
                TRUE,            // inherit handles-yes
                0,               // creation flags
                NULL,            // environment block
                NULL,            // current directory
                &si,             // startup info
                &pi))            // process info (out)
        {
            CloseHandle(file_pipes[1]);
          exit(EXIT_FAILURE);
      } else {
      if (!WriteFile(file_pipes[1],// outbound handle of pipe
                      &some_data,        // buffer to write
                      strlen(some_data), // size of buffer
                      &data_processed,   // bytes written
                      NULL))             // overlapping i/o structure
      {
          exit(EXIT_FAILURE);
      }

            printf("%d - wrote %d bytes\n", getpid(),
              data_processed);
            CloseHandle(file_pipes[1]);
        // Wait until spawned program is done processing.
           WaitForSingleObject(pi.hProcess, INFINITE);

           // Get the exit code for the process
           GetExitCodeProcess(pi.hProcess, &tstat);

           CloseHandle(pi.hThread);
           CloseHandle(pi.hProcess);

           if(tstat & 0x0)
               printf("Consumer failed\n");
        }
    }
    exit(EXIT_SUCCESS);
}
```

Consumer

```
#include <windows.h>
#include <stdlib.h>
#include <stdio.h>
#include <io.h>
#include <fcntl.h>
#include <process.h>
#include <string.h>

int main()
{
    size_t data_processed;
    char buffer[BUFSIZ + 1];

  memset(buffer, '\0', sizeof(buffer));
    data_processed = fread(buffer, 1, 3, stdin);
  //data_processed = 0;

  //gets(buffer);
    printf("%d - read %d bytes: %s\n", getpid(), data_processed,
buffer);
    exit(EXIT_SUCCESS);
}
```

Appendix G: Waiting for a Spawned Process

The following shows how UNIX code that waits for a child process can be migrated to Win32.

UNIX solution

```
#include <unistd.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>

int main()
{
    pid_t pid;
    int   tstat;

    printf("Running ps with fork and execlp\n");
    pid = fork();
    switch(pid)
    {
    case -1:
        perror("fork failed");
        exit(1);
    case 0:
        if (execlp("ps", NULL) < 0) {
            perror("execlp failed");
            exit(1);
        }
        break;
    default:
        break;
    }

    waitpid(pid, &tstat, 0);
    printf("Child process %d terminated with code %d.\n", pid,
tstat);
    exit(0);
}
```

Windows solutions

CreateProcess

```
#include <windows.h>
#include <process.h>
#include <stdio.h>

int main()
{
    STARTUPINFO    si;
    PROCESS_INFORMATION  pi;
    DWORD               tstat;

    GetStartupInfo(&si);

    printf("Running ps with CreateProcess\n");
    CreateProcess(NULL, "ps",  // Name of app to launch
  NULL,        // Default process security attributes
  NULL,        // Default thread security attributes
  FALSE,      // Don't inherit handles from the parent
  0,        // Normal priority
  NULL,        // Use the same environment as the parent
  NULL,        // Launch in the current directory
  &si,        // Startup Information
  &pi);        // Process information stored upon return

   // Suspend our execution until the child has terminated.
   WaitForSingleObject(pi.hProcess, INFINITE);

   // Obtain the termination code.
   GetExitCodeProcess(pi.hProcess, &tstat);

    printf("Child process %d terminated with code %d.\n",
      pi.dwProcessId, tstat);
   exit(0);
}
```

Spawn solution

```
#include <windows.h>
#include <process.h>
#include <stdio.h>

int main()
{
//  IF Visual Studio 7
//    intptr_t pid;

//  IF Visual Studio 6
    int pid;
    int tstat;
    printf("Running ps with spawnlp\n");
    pid = _spawnlp( _P_NOWAIT, "ps", "ps", NULL );

// Suspend our execution until the child has terminated.
// obtain termination code upon completion.
    _cwait(&tstat, pid, WAIT_CHILD);

    printf("Child process %d terminated with code %d.\n", pid,
tstat);
    exit(0);
}
```

Appendix H: Creating a Named Pipe

The following example illustrates creating and using a named pipe in UNIX.

UNIX example: creating a named pipe

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>

int main()
{
    int res = mkfifo("/tmp/my_fifo", 0777);
    if (res == 0)
        printf("FIFO created\n");
    exit(EXIT_SUCCESS);
}
```

Win32 example: creating a named pipe

```
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>
#include <string.h>

#define BUFSIZE 1024
#define PIPE_TIMEOUT 5000    // 5 Seconds

int main()
{
    BOOL fConnected;
    DWORD dwThreadId;
    HANDLE hPipe, hThread;
    LPTSTR lpszPipename = "\\\\.\\pipe\\mynamedpipe";

  // The following is an approximation of the mode bits used
  // in the UNIX example.  Will suffice until verified. 0777
  hPipe = CreateNamedPipe(
          lpszPipename,             // pipe name
          PIPE_ACCESS_DUPLEX,       // read/write access
          PIPE_TYPE_MESSAGE |       // message type pipe
          PIPE_READMODE_MESSAGE |   // message-read mode
          PIPE_WAIT,                // blocking mode
          PIPE_UNLIMITED_INSTANCES, // max. instances
          BUFSIZE,                  // output buffer size
          BUFSIZE,                  // input buffer size
          PIPE_TIMEOUT,             // client time-out
          NULL);                    // no security attribute

    if (hPipe != INVALID_HANDLE_VALUE)
        printf("FIFO created\n");
    exit(EXIT_SUCCESS);
}
```

Appendix I: Opening a FIFO

The following example shows how you should migrate UNIX FIFO code to Win32.

UNIX example: opening a FIFO

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>

#define FIFO_NAME "/tmp/my_fifo"

int main(int argc, char *argv[])
{
    int fd;
    int i;

if (argc < 2) {
    fprintf(stderr, "Usage call once with \"r\" and then with
      \"w\"\n");
    exit(EXIT_FAILURE);
}

// Check if the FIFO exists and create it if necessary.
    if (access(FIFO_NAME, F_OK) == -1) {
        fd = mkfifo(FIFO_NAME, 0777);
        if (fd != 0) {
            fprintf(stderr, "Could not create fifo %s\n", FIFO_NAME);
            exit(EXIT_FAILURE);
        }
    }
    printf("Process %d opening FIFO\n", getpid());

// Open FIFO and output status result
    if (strncmp(argv[1], "r", 1) )
        fd = open(FIFO_NAME, O_RDONLY | O_NONBLOCK);
    else if (strncmp(argv[1], "w", 1) )
        fd = open(FIFO_NAME, O_WRONLY | O_NONBLOCK);
    else {
        fprintf(stderr, "Usage call once with \"r\" and then with
          \"w\"\n");
        exit(EXIT_FAILURE);
    }
    printf("Process %d file descriptor: %d\n", getpid(), fd);
    sleep(3);
// Close FIFO
    if (fd != -1) close(fd);
    printf("Process %d finished\n", getpid());
    exit(EXIT_SUCCESS);
}
```

Win32 example: opening a FIFO

```
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>
#include <io.h>
#include <fcntl.h>
#include <process.h>
#include <string.h>

#define FIFO_NAME "\\\\.\\pipe\\mynamedpipe"
#define BUFSIZE 1024
#define PIPE_TIMEOUT 5000    // 5 Seconds
#define F_OK 0

void main()
{
    int fd;
    HANDLE hPipe;

// Check if the FIFO exists and create it if necessary.
    if (_access(FIFO_NAME, F_OK) == -1) {
        hPipe = CreateNamedPipe(
      FIFO_NAME,                 // pipe name
      PIPE_ACCESS_DUPLEX,        // read/write access
      PIPE_TYPE_MESSAGE |        // message type pipe
      PIPE_READMODE_MESSAGE |    // message-read mode
      PIPE_WAIT,                 // blocking mode
      PIPE_UNLIMITED_INSTANCES, // max. instances
      BUFSIZE,                   // output buffer size
      BUFSIZE,                   // input buffer size
      PIPE_TIMEOUT,              // client time-out
      NULL);                     // no security attribute

    if (hPipe == INVALID_HANDLE_VALUE) {
            fprintf(stderr, "Could not create fifo %s\n", FIFO_NAME);
            exit(EXIT_FAILURE);
        }
    }

    printf("Process %d opening FIFO\n", getpid());
// Open FIFO and output status result
    fd = open(FIFO_NAME, O_RDONLY);

    printf("Process %d file descriptor: %d\n", getpid(), fd);
    Sleep(PIPE_TIMEOUT);
// Close FIFO
    if (fd != -1)
        (void)close(fd);
    printf("Process %d finished\n", getpid());
    exit(EXIT_SUCCESS);
}
```

Appendix J: Interprocess Communication with FIFOs

The following example demonstrates how you would convert UNIX FIFO interprocess communications into Win32 code.

UNIX example: interprocess communication with FIFOs

Provider

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <limits.h>
#include <sys/types.h>
#include <sys/stat.h>

#define FIFO_NAME "/tmp/my_fifo"
#define BUFFER_SIZE PIPE_BUF
#define TEN_MEG (1024 * 1024 * 10)

int main()
{
    int pipe_fd;
    int res;
    int open_mode = O_WRONLY;
    int bytes_sent = 0;
    char buffer[BUFFER_SIZE + 1];

    if (access(FIFO_NAME, F_OK) == -1) {
        res = mkfifo(FIFO_NAME, 0777);
        if (res != 0) {
            fprintf(stderr, "Could not create fifo %s\n", FIFO_NAME);
            exit(EXIT_FAILURE);
        }
    }

    printf("Process %d opening FIFO O_WRONLY\n", getpid());
    pipe_fd = open(FIFO_NAME, open_mode);
    printf("Process %d result %d\n", getpid(), pipe_fd);

    if (pipe_fd != -1) {
        while(bytes_sent < TEN_MEG) {
            res = write(pipe_fd, buffer, BUFFER_SIZE);
            if (res == -1) {
                fprintf(stderr, "Write error on pipe\n");
                exit(EXIT_FAILURE);
            }
            bytes_sent += res;
        }
        (void)close(pipe_fd);
    }
    else {
        exit(EXIT_FAILURE);
    }
    printf("Process %d finished\n", getpid());
    exit(EXIT_SUCCESS);
}
```

Consumer

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <limits.h>
#include <sys/types.h>
#include <sys/stat.h>

#define FIFO_NAME "/tmp/my_fifo"
#define BUFFER_SIZE PIPE_BUF

int main()
{
    int pipe_fd;
    int res;
    int open_mode = O_RDONLY;
    char buffer[BUFFER_SIZE + 1];
    int bytes_read = 0;

    memset(buffer, '\0', sizeof(buffer));

    printf("Process %d opening FIFO O_RDONLY\n", getpid());
    pipe_fd = open(FIFO_NAME, open_mode);
    printf("Process %d result %d\n", getpid(), pipe_fd);

    if (pipe_fd != -1) {
        do {
            res = read(pipe_fd, buffer, BUFFER_SIZE);
            bytes_read += res;
        } while (res > 0);
        (void)close(pipe_fd);
    }
    else {
        exit(EXIT_FAILURE);
    }

    printf("Process %d finished, %d bytes read\n", getpid(),
      bytes_read);
    exit(EXIT_SUCCESS);
}
```

Win32 example: interprocess communication with FIFOs

Provider

```
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>
#include <io.h>
#include <fcntl.h>
```

```
#include <process.h>
#include <string.h>

#define FIFO_NAME "\\\\.\\pipe\\mynamedpipe"
#define PIPE_SIZE 1024
#define BUFFER_SIZE PIPE_SIZE
#define TEN_MEG (1024 * 1024 * 10)
#define PIPE_TIMEOUT 5000    // 5 Seconds
#define F_OK 0

void main()
{
    int open_mode = O_WRONLY;
    HANDLE hPipe;
    int bytes_sent = 0;
    DWORD NumberOfBytesWritten;
    char buffer[BUFFER_SIZE + 1];
    BOOL fConnected;

// Check whether the FIFO exists and create it if necessary.
// The FIFO is then opened and output given to that effect while the
program
// catches forty winks. Last of all, the FIFO is closed.

  if (_access(FIFO_NAME, F_OK) == -1) {
    hPipe = CreateNamedPipe(
      FIFO_NAME,                // pipe name
      PIPE_ACCESS_DUPLEX,       // read/write access
      PIPE_TYPE_BYTE|           // I/O as stream of bytes
      PIPE_WAIT,                // blocking mode
      PIPE_UNLIMITED_INSTANCES, // max. instances
      PIPE_SIZE,                // output buffer size
      PIPE_SIZE,                // input buffer size
      PIPE_TIMEOUT,             // client time-out
      NULL);                    // no security attribute

    if (hPipe == INVALID_HANDLE_VALUE) {
            fprintf(stderr, "Could not create fifo %s\n", FIFO_NAME);
            exit(EXIT_FAILURE);
        }
    }

  fConnected = ConnectNamedPipe(hPipe, NULL) ?
         TRUE : (GetLastError() == ERROR_PIPE_CONNECTED);

      printf("Process %d opening FIFO O_WRONLY\n", getpid());
  if (fConnected)
  {
    while(bytes_sent < TEN_MEG) {
        if (!WriteFile(
          hPipe,                   // handle to file
      buffer,                  // data buffer
      BUFFER_SIZE,             // number of bytes to write
      &NumberOfBytesWritten,   // number of bytes written
      NULL)                    // overlapped buffer
        ) {
      fprintf(stderr, "Write error on pipe\n");
      exit(EXIT_FAILURE);
        }
```

```
                bytes_sent += NumberOfBytesWritten;
    }

    printf("Process %d finished\n", getpid());
  }
  else {
    // The client could not connect, so close the pipe.
    CloseHandle(hPipe);
  }

    exit(EXIT_SUCCESS);
}
```

Consumer

```
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>
#include <io.h>
#include <fcntl.h>
#include <process.h>
#include <string.h>

#define FIFO_NAME "\\\\.\\pipe\\mynamedpipe"
#define PIPE_SIZE 1024
#define BUFFER_SIZE PIPE_SIZE
#define TEN_MEG (1024 * 1024 * 10)
#define PIPE_TIMEOUT 5000    // 5 Seconds
#define F_OK 0

void main()
{
    HANDLE hPipe1;
    DWORD NumberOfBytesRead = 0;
    char buffer[BUFFER_SIZE + 1];
    int bytes_read = 0;

    memset(buffer, '\0', sizeof(buffer));

    printf("Process %d opening FIFO O_RDONLY\n", getpid());
  hPipe1 = CreateFile(FIFO_NAME,       // Open the FIFO
                GENERIC_READ,              // open for reading
                0,                         // share for writing
                NULL,                      // no security
                OPEN_EXISTING,             // existing file only
                FILE_ATTRIBUTE_NORMAL,     // normal file
                NULL);                     // no attr. template

  if (hPipe1 == INVALID_HANDLE_VALUE)
  {
        printf("Could not create file %s\n", FIFO_NAME);
        exit(EXIT_FAILURE);
  }
    printf("Process %d result %X\n", getpid(), hPipe1);

    do {
    ReadFile(
      hPipe1,             // handle to file
      buffer,             // data buffer
      BUFFER_SIZE,        // number of bytes to read
      &NumberOfBytesRead, // number of bytes read
      NULL                // overlapped buffer
      );

        bytes_read += NumberOfBytesRead;
    } while (NumberOfBytesRead > 0);

  CloseHandle(hPipe1);

    printf("Process %d finished, %d bytes read\n", getpid(),
            bytes_read);
    exit(EXIT_SUCCESS);
}
```

Appendix K: Winforks Example

The following is the code for the Winforks example discussed in this chapter.

```
// Both.h
// Environment name definitions shared by both parent and child
#define EXIT_NAME    "Exit"
#define LOOP_COUNT   "LoopCount"
#define PROCESS_NAME "ProcessName"

// ChildExe.c
// Code for child processes
#include <windows.h>
#include <stdlib.h>
#include <stdio.h>
#include "Both.h"

#define MY_NAME "ChildProcess"

// FUNCTION: main
// PURPOSE:  entrypoint for child executable
// PARAMETERS:
//   argc - number of command line arguments
//   argv - array of command line arguments
// RETURN VALUE:
//   int process exit code: 0 = success
//
int main(int argc, char *argv[])
{
  char   *ProcessName;           // Process name from env
  char   *ExitName;              // Exit event name from env
  char   *LoopBuf;               // String buf for loop count
  int    LoopCount;              // Total loop count from env
  int    Count;                  // Main loop counter
  DWORD  WaitStatus;             // Status code from wait call
  DWORD  TimeoutMSec = 2 * 1000; // Wait timeout in msec
  HANDLE ExitEventHandle;        // Exit event handle

  // Get process name from env
  ProcessName = getenv(PROCESS_NAME);
  if (!ProcessName)
  {
    printf("%s: getenv error for process name: %s\n",
MY_NAME, PROCESS_NAME);
```

```
    return 1;
  }

  // Get main loop count from env
  LoopBuf = getenv(LOOP_COUNT);
  if (!LoopBuf)
  {
    printf("%s %s: getenv error for loop count %s\n",
MY_NAME, ProcessName, LOOP_COUNT);
    return 2;
  }
  LoopCount = atoi(LoopBuf);

  // Get Exit event name from env
  ExitName = getenv(EXIT_NAME);
  if (!ExitName)
  {
    printf("%s %s: getenv error for event name %s\n",
MY_NAME, ProcessName, EXIT_NAME);
    return 5;
  }

  // Get handle for Exit event
  ExitEventHandle = OpenEvent(
          EVENT_ALL_ACCESS, // access rights - all
          FALSE,            // event not inheritable
          ExitName          // event name from env
          );
  if (!ExitEventHandle)
  {
    printf("%s %s: error opening %s event %s\n", MY_NAME,
ProcessName, EXIT_NAME, ExitName);
    return 6;
  }

  /////////////////////
  // processing loop
  /////////////////////
  for (Count = 1; Count <= LoopCount; ++Count)
  {
    // Wait for event to occur or timeout
    WaitStatus = WaitForSingleObject(
          ExitEventHandle, // object handle
          TimeoutMSec      // timeout value in msec
          );

    // If it's timeout, do background processing
    if (WaitStatus == WAIT_TIMEOUT)
    {
      // TODO: put periodic polling processes here
      printf("%s %s: waiting %d\n",
MY_NAME, ProcessName, Count);
    }
    else if (WaitStatus == WAIT_OBJECT_0)
    {
      // Event happened, WaitForSingleObject returns
      // a specific value indicating that the
      // exit event object was signaled.
      printf("%s %s: got %s event: %s\n", MY_NAME,
ProcessName, EXIT_NAME, ExitName);
```

```
      Count = LoopCount + 1;
    }
    else
    {
      // unhandled return status, give up
      printf("%s %s: got unknown event: %d\n",
MY_NAME, ProcessName, WaitStatus);
      Count = LoopCount + 1;
    }
  } // end processing loop

  printf("%s %s: exiting\n", MY_NAME, ProcessName);
  return 0;
}

// WinForks.c
// Code for parent process
#include <windows.h>
#include <stdlib.h>
#include <stdio.h>
#include "Both.h"

// Event and other names
#define EXIT_EVENT     "TheExitNowEvent"
#define MY_NAME        "ParentProcess"
#define CHILD_PROJECT  "ChildExe"

// Handle array indices
#define MAX_CHILDREN  4
#define MAX_HANDLES   MAX_CHILDREN

// Child process info structure and functions
typedef struct ChildProcInfo
{
  BOOL    IsRunning;
  UINT    ProcIndex;
  HANDLE  ExitEvent;
  PROCESS_INFORMATION ProcInfo;
} ChildProcInfo;

UINT ChildStart(ChildProcInfo *ProcInfoPtr,
LPCTSTR ChildProject,
UINT ProcIndex);
BOOL ChildStop(ChildProcInfo *ProcInfoPtr);
BOOL ChildCleanup(ChildProcInfo *ProcInfoPtr);

// FUNCTION: main
// PURPOSE:  application entry point
// PARAMETERS:
//   argc - number or count of command line arguments
//   argv - array of command line argument strings
// RETURN VALUE: int success code
//
int main(int argc, char* argv[])
{
  DWORD  HandleCount;            // Count of child process handles
  DWORD  WaitStatus;             // Status returned by wait func
  DWORD  HandleIndex;            // Handle index returned by wait
  DWORD  TimeoutMSec = 4 * 1000; // Wait timeout in milliseconds
  HANDLE ObjectHandles[MAX_HANDLES]; // Array of child proc handles
  int    Count, ii;                  // Main loop counter, another
  BOOL   IsRunning = TRUE;           // Keep running flag
  ChildProcInfo ChildInfo[MAX_CHILDREN]; // Array of child procinfo

  // Initial clear all handles
  memset(ObjectHandles, 0, sizeof(ObjectHandles));

  // Start a number of child processes
  for (ii = 0; ii < MAX_CHILDREN; ++ii)
  {
    if (ChildStart(&ChildInfo[ii], CHILD_PROJECT, ii + 1) != 0)
    {
      // error starting child
      IsRunning = FALSE;
      break;
    }
  }

  /////////////////////
  // processing loop //
  /////////////////////
  for (Count = 1; IsRunning; ++Count)
  {
    // Build array of child process handles
    for (HandleCount = 0, ii = 0; ii < MAX_CHILDREN; ++ii)
    {
      if (ChildInfo[ii].IsRunning)
        ObjectHandles[HandleCount++] =
ChildInfo[ii].ProcInfo.hProcess;
    }

    // Wait for child event to occur or timeout
    WaitStatus = WaitForMultipleObjects(
          HandleCount,   // num objs waiting for
          ObjectHandles, // array ofobject handles
          FALSE,         // wait for 1 (not all)
          TimeoutMSec    // timeout value in msec
          );

    // If it's timeout, do background processing
    if (WaitStatus == WAIT_TIMEOUT)
    {
      // TODO: put periodic polling processes here
      printf("%s: loop number %d\n", MY_NAME, Count);
    }
    else
    {
      // Event happened, WaitForMultipleObjects returns
      //   the index of the handle signaled + an offset
      HandleIndex = WaitStatus - WAIT_OBJECT_0;
      if (HandleIndex < HandleCount)
      {
        // child exited, stop running when none left
        for (ii = 0, IsRunning = FALSE;
ii < MAX_CHILDREN;
++ii)
        {
          if (ChildInfo[ii].ProcInfo.hProcess ==
ObjectHandles[HandleIndex])
          {
            printf("child %d [%x] exited\n",
ChildInfo[ii].ProcIndex,
ChildInfo[ii].ProcInfo.hProcess);

            // Clean up this one's info
            ChildCleanup(&ChildInfo[ii]);
          }
          else if (ChildInfo[ii].IsRunning)
            IsRunning = TRUE;
        }
      }
      else
      {
        // unhandled return status. stop children, self
        printf("unknown event %d\n", WaitStatus);
        IsRunning = FALSE;
      }
    } // else event happened
  } // while IsRunning

  // Final cleanup
  for (ii = 0; ii < MAX_CHILDREN; ++ii)
  {
    ChildStop(&ChildInfo[ii]);
  }
  return 0;
}

// FUNCTION: ChildStart
// PURPOSE: start a child process
// PARAMETERS:
//   ProcInfoPtr  - pointer to process information structure
//   ChildProject - project name for child executable
//   ProcIndex    - child process index (count)
// RETURN VALUE:
//   0    - OK
//   else - some error happened
//
UINT ChildStart(ChildProcInfo *ProcInfoPtr, LPCTSTR ChildProject,
  UINT ProcIndex)
{
  char  Name[40];           // String for building names
  char  StrBuf[256];        // String for building env entries
  char  EnvStrings[512];    // The environment passed to child
  char  *NextEnvChar;       // Pointer to next output char in env
  STARTUPINFO StartupInfo; // Startup info for CreateProcess

  // Start anew
  memset(ProcInfoPtr, 0, sizeof(ChildProcInfo));
  ProcInfoPtr->ProcIndex = ProcIndex;

  // Setup environment as we go along
  NextEnvChar = EnvStrings;

  // Add "LoopCount=<some number>" to environment:
  sprintf(StrBuf, "%s=%d", LOOP_COUNT, ProcIndex + 2);
  strcpy(NextEnvChar, StrBuf);
  NextEnvChar += strlen(StrBuf) + 1;

  // Create a process name
  sprintf(Name, "P-%d", ProcIndex);

  // Add "ProcessName=<proc name>" to environment:
  sprintf(StrBuf, "%s=%s", PROCESS_NAME, Name);
  strcpy(NextEnvChar, StrBuf);
  NextEnvChar += strlen(StrBuf) + 1;

  // Create an exit event name
  sprintf(Name, "%s%d", EXIT_EVENT, ProcIndex);

  // Add "Exit=<exit event name>" to environment:
  sprintf(StrBuf, "%s=%s", EXIT_NAME, Name);
  strcpy(NextEnvChar, StrBuf);
  NextEnvChar += strlen(StrBuf) + 1;

  // Create the actual Exit event
  ProcInfoPtr->ExitEvent = CreateEvent(
              NULL,  // Def security attr
              TRUE,  // Manual reset
              FALSE, // Initial not signal
              Name   // Name (defined here)
              );
  if (!ProcInfoPtr->ExitEvent)
  {
    printf("error creating event %s", Name);
    ChildCleanup(ProcInfoPtr);
    return 2;
  }

  // Add null terminator to environment block (2 nulls at end)
  *NextEnvChar = 0;

  // Setup startup info
  memset(&StartupInfo, 0, sizeof(StartupInfo));
  StartupInfo.cb = sizeof(StartupInfo);

  // Set application path
  // Release assumes parent and child exes are in same dir
  GetModuleFileName(NULL, StrBuf, sizeof(StrBuf));
  NextEnvChar = strrchr(StrBuf, '\\');
#if _DEBUG
  // Get past Debug dir
  *NextEnvChar = 0;
  NextEnvChar = strrchr(StrBuf, '\\');
  *(++NextEnvChar) = 0;
  strcat(NextEnvChar, ChildProject);
  strcat(NextEnvChar, "\\Debug\\");
#else
  *(++NextEnvChar) = 0;
#endif
  strcat(NextEnvChar, ChildProject);
  strcat(NextEnvChar, ".exe");

  // Create "child" process
  if (!CreateProcess(
        StrBuf,            // Application path
        NULL,              // No cmd line, uses env
        NULL,              // Default process security
        NULL,              // Default thread security
        FALSE,                   // No inherit handles
        NORMAL_PRIORITY_CLASS,   // Creation flags
        EnvStrings,              // The env we built
        NULL,                    // Same current dir
        &StartupInfo,            // Startup info
        &ProcInfoPtr->ProcInfo)) // Returned proc info
  {
    printf("error %d creating child process", GetLastError());
    ChildCleanup(ProcInfoPtr);
    return 3;
  }

  // Completed OK
  ProcInfoPtr->IsRunning = TRUE;
  return 0;
}

// FUNCTION: ChildStop
// PURPOSE:  Stop a child process
// PARAMETERS:
//   ProcInfoPtr - pointer to process information structure
// RETURN VALUE:
//   TRUE:  success
//   FALSE: error
//
BOOL ChildStop(ChildProcInfo *ProcInfoPtr)
{
  if (ProcInfoPtr->IsRunning)
  {
    return SetEvent(ProcInfoPtr->ExitEvent);
  }
  return FALSE;
}

// FUNCTION: ChildCleanup
// PURPOSE:  Clean up a child process info struct
// PARAMETERS:
//   ProcInfoPtr - pointer to process information structure
// RETURN VALUE:
//   TRUE:  success
//   FALSE: error
//
BOOL ChildCleanup(ChildProcInfo *ProcInfoPtr)
{
  BOOL RetStatus = TRUE;

  // Close all the handles, remember any failure
  if (ProcInfoPtr->ExitEvent)
    RetStatus &= CloseHandle(ProcInfoPtr->ExitEvent);
  if (ProcInfoPtr->ProcInfo.hThread)
    RetStatus &= CloseHandle(ProcInfoPtr->ProcInfo.hThread);
  if (ProcInfoPtr->ProcInfo.hProcess)
    RetStatus &= CloseHandle(ProcInfoPtr->ProcInfo.hProcess);

  // Clear struct
  memset(ProcInfoPtr, 0, sizeof(ChildProcInfo));

  return RetStatus;
}
```

Send feedback to Microsoft