# UNIX Custom Application Migration Guide

Version 2.0

# Volume 2: Migrate Using Windows Services for UNIX 3.5

Published: May 2006

Microsoft

# Contents

# About This Volume

## Introduction to Volume 2

Volume 1 of the *UNIX Custom Application Migration Guide* discussed how to apply the Envisioning and Planning Phases of the Microsoft® Solutions Framework (MSF) Process Model when conducting a UNIX to Microsoft Windows® migration project. This volume, Volume 2: *Migrate Using Windows Services for UNIX 3.5,* applies the next phases in the Process Model, the Developing Phase and the Stabilizing Phase, and directs it specifically for using Windows Services for UNIX 3.5. This volume describes the architectural differences between UNIX and Microsoft Interix environments and the features of Interix. Because there are certain differences between UNIX and Microsoft Interix, the UNIX code must be modified for it to work in the Microsoft Windows environment using Windows Services for UNIX 3.5. This volume addresses these potential coding differences by looking at the solution from various categories.

These categories are:

- Process management.
- Thread management.
- Memory management.
- File management.
- Infrastructure services.
- User interface migration.
- Development considerations for deployment.
- Functions to change for Interix.
- Deployment considerations and testing activities.
- Stabilizing Phase activities.

For each of these categories, this volume:

- Describes the coding differences between UNIX and Interix.
- Outlines options for converting the code.
- Illustrates the options with source code examples.

This information helps you choose the solution that is appropriate to your application; you can use these examples as the basis for constructing your Windows code. This volume provides sufficient information so that you can choose the best method of converting the code. After choosing a method, refer to the standard documentation for details of the Interix functions and application programming interfaces (APIs). References are provided throughout this volume for information on the recommended coding changes.

For more information on activities in the Developing Phase as they relate to a migration project, refer to Chapter 2, “Developing Phase: Process Milestones and Technology Considerations” of this volume.

## Intended Audience

This volume is for UNIX developers and testers who are responsible for migrating UNIX code to Windows Services for UNIX 3.5. These developers are UNIX programmers involved in developing the solution on Windows using Windows Services for UNIX 3.5. Various advantages that the developers and testers would gain from this volume are:

- **Developers**. Developers can learn about the various alternative methods for migrating from UNIX to Windows and how to choose the best strategy to fit their environment and the application types.
- **Testers**. Testers can gain more insight on the testing methodology that is best suited for their migration scenario. Using this guide, they can test the application for various aspects, such as functionality, management, performance, and stability.

## Knowledge Prerequisites

The readers of this volume should possess the following knowledge prerequisites:

- Basic knowledge of UNIX and Windows internals.
- Hands-on experience on Windows environments.
- Familiarity with UNIX administration skills.
- Understanding the configurations with Windows Services for UNIX 3.5.

It is also suggested that you read “About This Guide” (the master preface) as well as the rest of Volume 1, *Plan* before reading this volume.

# Layout of the Guide: Volume 2

The following diagram depicts the layout of the guide and how the volumes of the guide correlate with the components of the MSF Process Model. The white-shaded portion indicates the position of the current volume in the layout of the entire guide.

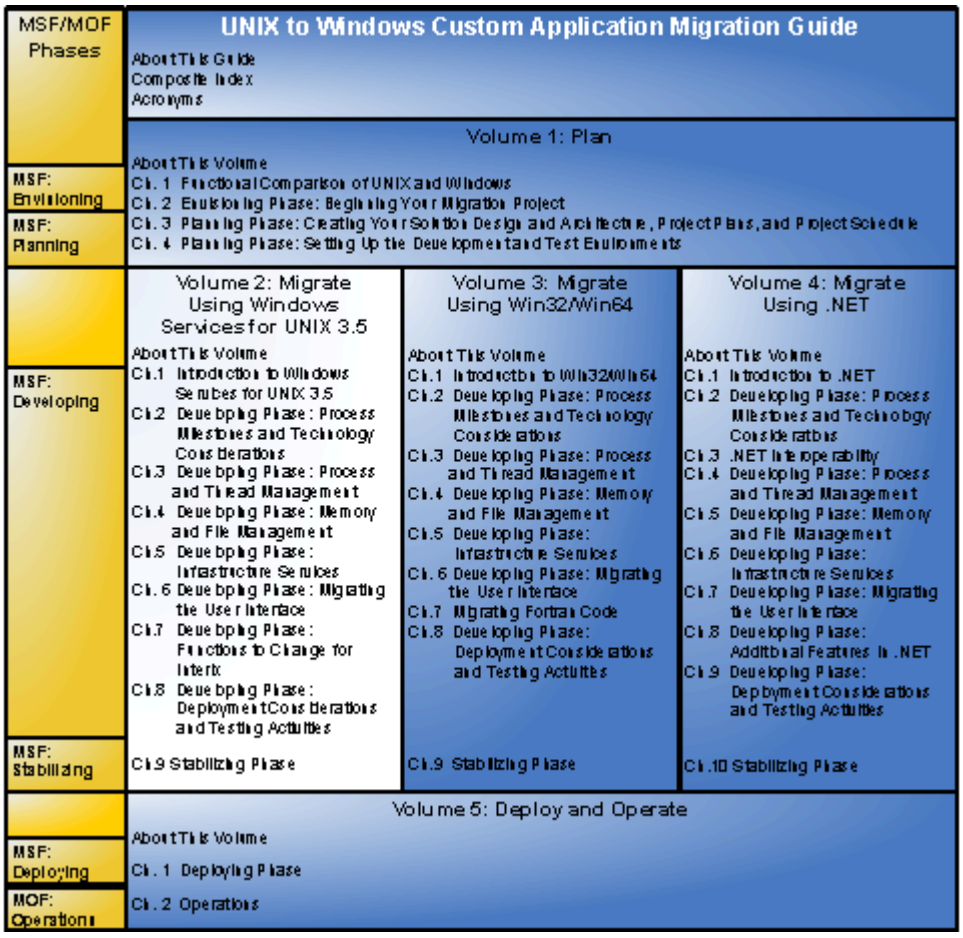


**Figure 0.1. UCAMG organization**

# Organization of Content

The content of this volume is organized into the following chapters:

- **About This Volume**. This chapter provides information about the organization of the guide and about its intended audience. It also lists the knowledge prerequisites required for this volume and provides resources, such as document conventions, used in this guide.
- **Chapter 1: Introduction to Windows Services for UNIX 3.5**. This chapter provides an overview of Windows Services for UNIX 3.5, including its installation and configuration. It also discusses the architectural differences that exist between UNIX and Interix, the core subsystem of Windows Services for UNIX 3.5, to support the UNIX environment on Windows.
- **Chapter 2: Developing Phase: Process Milestones and Technology Considerations**. This chapter describes Developing Phase activities and the tools used in the development environment of the Planning Phase for Interix migration. This chapter provides the instructions for developing the solution components in the Interix environment. It also details the application build instruction and debugging techniques, as well as providing a comparison of the source code header files shipped with Interix and other environments.
- **Chapter 3: Developing Phase: Process and Thread Management**. This chapter discusses process and thread management in Interix. It also discusses the differences in the process and thread management models of UNIX and Interix environments, providing the suggested workarounds for them.
- **Chapter 4: Developing Phase: Memory and File Management**. This chapter discusses memory and file management in Interix and the functions that are used to implement them. It also provides information on file security in the Interix and Windows environments.
- **Chapter 5: Developing Phase: Infrastructure Services**. This chapter provides information on the different infrastructure services for Interix, such as signals, IPC components, networking protocols, and sockets.
- **Chapter 6: Developing Phase: Migrating the User Interface**. This chapter describes the process of migrating from a UNIX user interface (UI) to a Windows UI. It also guides readers through the steps required to migrate X Windows, Motif, and POSIX applications to the Windows UI.
- **Chapter 7: Developing Phase: Functions to Change for Interix**. This chapter describes the functions that need to be changed or removed so that the code compiles under Interix. It also describes the list of C library functions and other APIs along with their recommended replacements in the Interix environment.
- **Chapter 8: Developing Phase: Deployment Considerations and Testing Activities**. This chapter discusses the activities to be performed before closing the Developing Phase. It also discusses the Interix development considerations for deployment and the various testing activities in the Developing Phase. It also discusses the key activities that a developer needs to perform in order to prepare the Interix migrated application for a smooth deployment.
- **Chapter 9: Stabilizing Phase**. This chapter discusses the different levels of testing and tuning that must be administered to the applications that are migrated to Windows using Windows Services for UNIX 3.5.

# Resources

This section describes the various resources that are included in the *UNIX Custom Application Migration Guide* and information that will assist in using the guide.

## Acronyms

See the acronyms list accompanying this guide for a list of the acronyms and their meanings used in this volume.

## Document Conventions

The document conventions used in this volume are primarily designed to help you quickly identify the operating system and the interface (command line or graphical) being discussed—Windows or UNIX. In general, Windows operating-system commands are executed by clicking user interface (UI) elements, and these elements are visually distinguishable by the use of bold text. In contrast, the UNIX operating system typically uses a command-line interface, and these instructions are visually distinguished by the use of Monospace font.

These interface and execution differences are not absolute; and in case visual cues do not unambiguously delineate between operating systems, the text will clearly make this distinction.

Table 0.1 lists the document conventions used in this guide.

**Table 0.1. Document Conventions**

| Text Element | Meaning |
|---|---|
| **Bold text** | Used in the context of paragraphs for commands; literal arguments to commands (including paths when they form part of the command); switches; and programming elements, such as methods, functions, data types, and data structures.<br>Also used to identify UI elements. |
| *Italic text* | Used in the context of paragraphs for variables to be replaced by the user.<br>Also used to emphasize important information. |
| Monospace font | Used for excerpts from configuration files, code examples, and terminal sessions. |
| **Monospace bold font** | Used to represent commands or other text that the user types. |
| *Monospace italic font* | Used to represent variables the reader supplies in command-line examples and terminal sessions. |
| Shell prompts | The Ksh prompt is used in Windows. |
| **Note** | Represents a note. |
| Code | Represents code. |

## Code Samples

The build volumes, Volume 2: *Migrate Using Windows Service for UNIX 3.5*, Volume 3: *Migrate Using Win32/64,* and Volume 4: *Migrate Using .NET,* of this guide contain several code samples to illustrate certain programming concepts. These code samples are available as source files in a Tools and Templates folder in the download version of this guide, available at http://go.microsoft.com/fwlink/?LinkId=30864.

# Chapter 1: Introduction to Windows Services for UNIX 3.5

## Overview of Windows Services for UNIX 3.5

Windows Services for UNIX 3.5 components provide interoperability between Microsoft® Windows® and UNIX platforms. Accordingly, you can use Windows Services for UNIX to manage and conduct activities and tasks on both platforms and to share information between the two. This includes sharing files, centralizing and managing passwords and groups, and conducting character-based terminal sessions.

Windows Services for UNIX includes Interix, a complete, high-performance UNIX environment that provides two command shells (C and Korn) and more than 350 shell scripts, commands, and tools (such as the **vi** editor and Perl). Windows Services for UNIX also includes a complete set of development tools and libraries that you can use to port your UNIX-based applications to the Interix subsystem.

Figure 1.1 illustrates Windows, Windows Services for UNIX 3.5, and Interix.

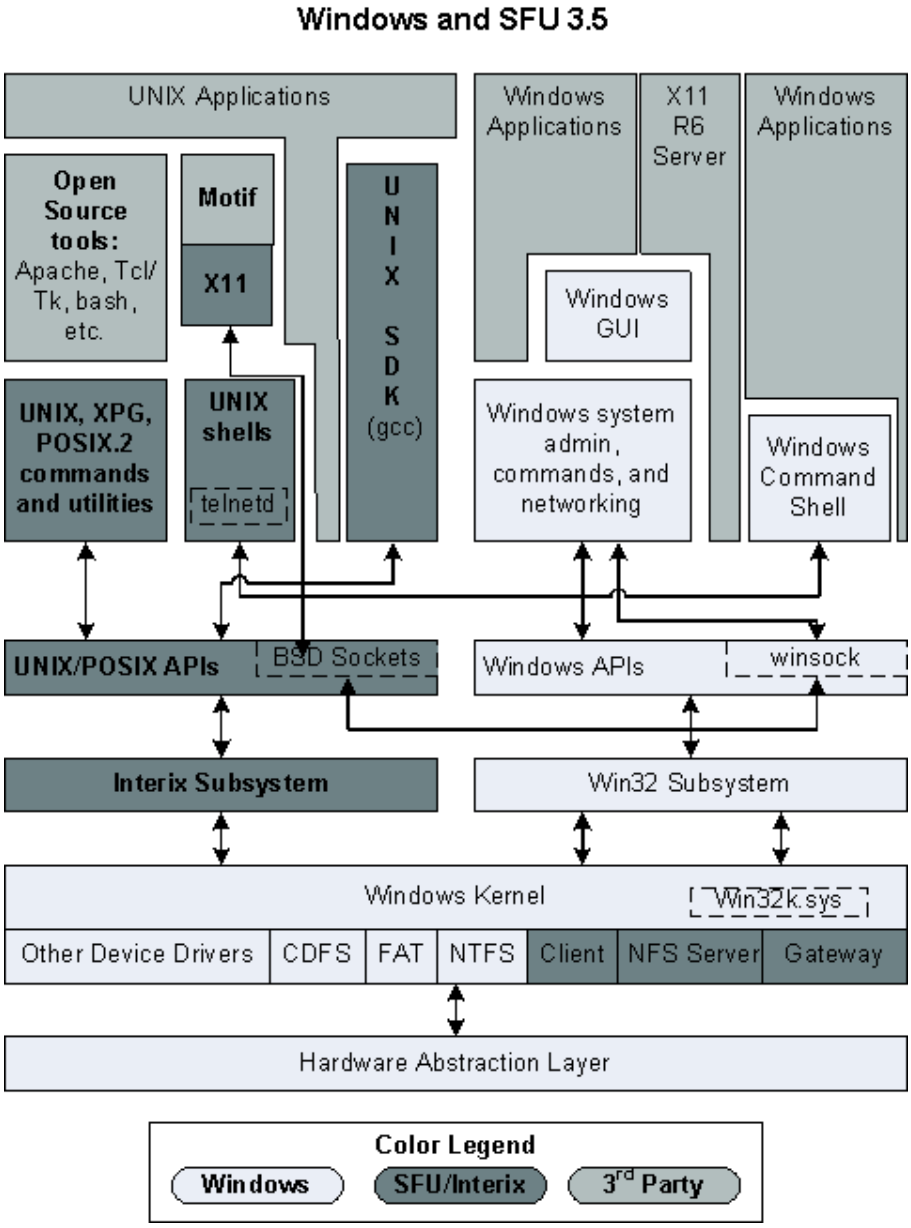


**Figure 1.1. Windows Services for UNIX 3.5 and the Interix subsystem**

This section and the subsequent sections explain the features of Windows Services for UNIX 3.5 and discuss the architectural differences between Windows Services for UNIX 3.5 and UNIX. You can use this information to understand the architectural differences between the two and analyze your application components accordingly.

Interix is a multiuser UNIX environment that operates on Windows. The Interix subsystem and its accompanying tools provide an environment that resembles any other UNIX system. It also includes case-sensitive file names, job control, and compilation tools. The Interix subsystem offers true UNIX functionality without any emulation because it is layered on top of the Windows kernel.

A computer running Windows Services for UNIX provides two different command-line environments—the UNIX environment and the Windows environment. The applications run on specific subsystems and in specific environments. When you run applications on the Interix subsystem, you get a UNIX environment; when you run applications on the Windows subsystem, you get a Windows environment.

The Interix subsystem provides both the tools and the API libraries for porting applications to run on Windows-based computers.

When installing Windows Services for UNIX 3.5, select the Custom installation choice. As a general user (not as a developer), install Active State Perl with the Custom installation. As a developer, choose Active State Perl plus both the Interix Software Development Kit (SDK) and the GNU SDK during installation. In addition, select Case Sensitive Pathnames and setuid Binaries to get the best UNIX behavior. With the SDK, which provides a front end for Microsoft Visual C++®, you can have a UNIX environment for development, but still have the benefits of the native compiler for Windows.

# Architectural Differences Between UNIX and Interix

This section compares UNIX and Interix architectures, emphasizing the areas that directly affect software development. It explains architectural differences between the UNIX and Interix environments at a high level. You can use this information to become familiar with the high-level architecture of the Interix and Windows subsystems. UNIX architecture can be divided into the following three levels of functionality (as shown in Figure 1.2):

1. **Kernel**. Schedules tasks, manages resources, and controls security.
2. **Shell**. Acts as the user interface, interpreting user commands and starting applications.
3. **Utilities**. Supplement the basic capabilities of the operating system by providing useful utility functions.

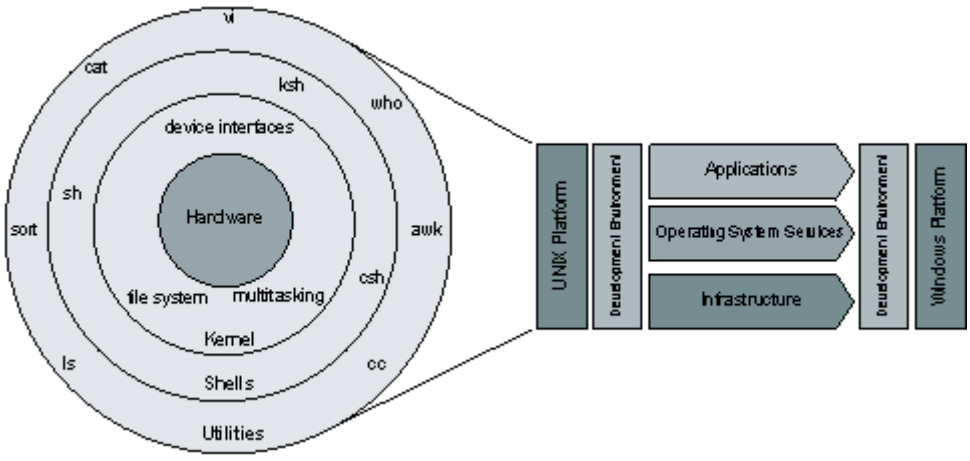


**Figure 1.2. The source platform: UNIX**

Interix is a full application execution subsystem running beside the Microsoft Win32® subsystem that allows you to compile and run UNIX programs and scripts on Windows operating systems.

The structure of Windows can be divided into the following four parts:

1. **Environment subsystems**. A set of programming environments that provide services to applications through an API, such as Win32 and Interix.
2. **Functional subsystems**. A set of protected subsystems that provides key operating system services to the environment subsystems, such as security, memory management, and input/output (I/O).
3. **Windows NT® kernel**. This contains the Windows NT Executive. This exports generic services that protected subsystems call to obtain basic operating system services.
4. **Hardware abstraction layer (HAL)**. This allows kernel and device drivers to interact with hardware devices at a general level.

Figure 1.3 shows the Windows architecture and how Interix is placed in it.

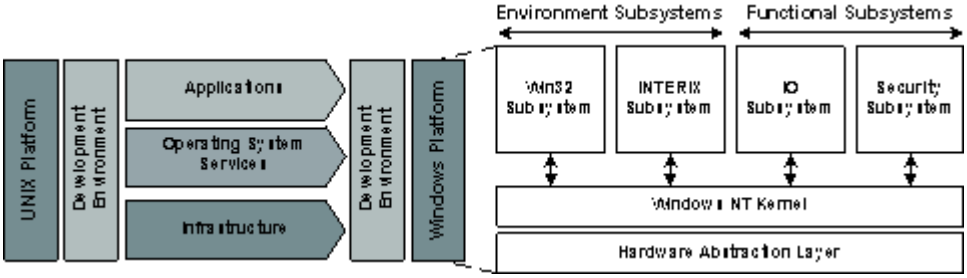


**Figure 1.3. The target platform: Windows**

# Features in Interix

This section discusses the implementation mechanism of several features in the Interix subsystem. Because of the implementation differences between major features in UNIX and Interix, UNIX code must be modified for it to work in the Windows environment using Windows Services for UNIX 3.5. The implementation differences between the two are described in the following sections of this chapter:

- Process Management
- Thread Management
- Memory Management
- File Management
- Infrastructure Services
- Development and Debugging Tools

## *Process Management*

This section describes the implementation of the process management-related mechanisms in the Interix subsystem. Multitasking operating systems such as Windows and UNIX must manage and control many processes simultaneously. Each process has its own code, data, system resources, and state. These resources include virtual address space, files, and synchronization objects. The following sections provide more information on the manner in which UNIX and Windows manage these processes. This section will help you understand the implementation of these mechanisms in the Interix environment.

## Multitasking

Using Windows Services for UNIX 3.5, jobs can be run in the foreground or background, or they can be suspended and resumed. From within an Interix shell, it is even possible to control Windows jobs. The priority value of the process can be adjusted using the **nice** command. However, Interix maps *nice* values to Windows process scheduling priorities according to a set of rules, which are discussed in Chapter 2, “Developing Phase: Process Milestones and Technology Considerations” of this volume.

## Multiple Users

The Interix system uses the UNIX multiuser model for hosting multiple users on a server running Windows. As a result, Windows can host a similar number of users running UNIX applications as expected on a stand-alone system running UNIX.

## Multithreading

With the release of Windows Services for UNIX 3.5, the Interix subsystem has extended its POSIX support to include much of the Pthread model and Pthread interfaces necessary for conformance to the IEEE Std1003.1 2001 standard. Interix supports many **pthread** functions associated with Portable Operating System Interface (POSIX) options, but the corresponding symbolic constants are not defined in the **unistd.h** file because Interix does not implement the complete set of functions associated with that option.

Although POSIX_THREAD_PRIORITY_SCHEDULING is defined, Interix only supports the SCHED_OTHER scheduling policy and does not support the SCHED_RR or SCHED_FIFO policy. PTHREAD_SCOPE_SYSTEM is the only scheduling contention scope that Interix provides. This means that all threads on the system are scheduled with respect to each other, regardless of the process that owns the thread.

**Note** More information on Pthread APIs implementation in Interix is available in the “Appendix A–Pthread APIs Implemented in Interix” section at http://www.microsoft.com/technet/interopmigration/unix/sfu/pthreads0.mspx.

## Process Hierarchy

Interix maintains a process hierarchy, and even tracks the parent-child relationship of Win32 processes on the same system. The Interix command **ps -efi** displays processes and their relationship to each other. The **-i** option is specific to Interix and its output is indented to display the process hierarchy. The indentation represents the process tree. Each process is followed by its child processes recursively.

In some versions of UNIX, an application can provide for automatic cleanup of child processes by calling the **signal**() function as follows:

```
Signal (SIGCHLD, SIG_IGN)
```

However, in the POSIX standard, the result of setting SIG_IGN as the signal handler for SIGCHLD is undefined. Therefore, you cannot use this method to automatically clean up child processes in portable applications in Interix.

## Signals

The Interix subsystem supports only one set of signal semantics—the POSIX.1 set. However, it supports several different sets of signal-handling APIs. Table 1.1 lists these signal sets.

**Table 1.1. Signal Sets Supported by Interix**

| ANSI | POSIX.1 | BSD 4.3 | System V |
|---|---|---|---|
| Signals are supported with the function **signal**(). Built on POSIX.1 **sigaction**().<br>It might not behave in the same way. | Signals are supported with the functions:<br>**sigaction**()<br>**sigpending**()<br>**sigprocmask**()<br>**sigsuspend**()<br>**sigemptyset**()<br>**sigfillset**()<br>**sigaddset**()<br>**sigdelset**()<br>**sigismember**() | Signals are supported with the functions:<br>**killpg**()<br>**sigsetmask**()<br>**sigblock**()<br>**sigvec**().<br>The signal mask for these functions is an **int**, not a **sigset_t**. Although **sigpause**() is provided, it is a System V call, which does not behave in the same way as the Berkeley Software Distribution (BSD) call. | Signals are supported with the functions:<br>**sighold**()<br>**sigignore**()<br>**sigpause**()<br>**sigrelse**()<br>**sigset**() |

The following important information applies when you are using signals with Interix:

- Signals on a POSIX.1 system are neither BSD nor System V Interface Definition (SVID). POSIX defined a new signal mechanism based on the **sigaction**() API. However, modern BSD and SV systems support the POSIX.1 signal model. Therefore, source will become more portable using the POSIX.1 model.
- The set of signals available is the set described in POSIX.1 and the single UNIX specification. The code of the user should use symbolic names instead of the actual values because some numbers might differ from traditional implementations.

Because an Interix process has a signal mask, it can block certain signals from arriving, except for *SIGKILL* or *SIGSTOP*. A process starts with a signal mask inherited from its parent. If any signals are generated and then blocked by the signal mask, they go into the set of pending signals.

If you are using the **signal**() API, the signal is still masked and remains masked until the mask is cleared. This can be a significant problem if your code does a **longjmp**() from the handler. Using the **sigaction**() API directly and the **siglongjmp**() API will correct some of those unexpected behaviors.

**Note** More information on process management in Interix is available in “Process Management” in Chapter 3, “Process and Thread Management” of this volume.

## Thread Management

Interix supports POSIX-compliant threads, a mechanism for providing concurrent code execution through multiple flows of control in a single program. In addition to its own thread ID, each thread has its own scheduling priority and policy, as well as a thread-specific value for error number. You can use this information to identify the thread-specific implementations in the Interix subsystem.

Writing code that executes properly in multithreaded applications requires special care. For example, some functions use global or static data, and therefore calling these functions from separate threads might cause a function call in one thread to corrupt the data relied upon by a call to the same function in another thread. The individual reference entries in the Interix subsystem interface reference and the SDK **man** pages indicate whether each routine is multithread-safe (MT-safe). If you want to call functions that are not MT-safe in a multithreaded application, you should ensure that the functions cannot be called concurrently in more than one thread.

In a multithreaded application, problems can occur when routines are called between a call to the **fork**() function and a call to a member of the **exit**() or **exec**() function families. A prototype of the functions **fork**, **exit**, and **exec** family is defined as follows.

```
pid_t fork (void)
void exit (int status)
int execl (const char *path, const char *arg ...)
int execle (const char *path, const char *arg ... char *const envp[])
int execlp (const char *file, const char *arg ...)
int execv (const char *path, char *const argv[])
int execve (const char *path, char *const argv[], char *const envp[])
int execvp (const char *file, char *const argv[])
```

If a routine is async-signal safe, that is, can be safely called in this situation, the reference topic in the Interix subsystem interface reference and **man** page for that function states that it is async-signal safe. Do not call routines that are not explicitly stated to be async-signal safe between calls to the **fork**() and **exit**() or **exec**() function.

**Note** More information on thread support in Windows Services for UNIX 3.5 is available at http://www.microsoft.com/technet/interopmigration/unix/sfu/pthreads0.mspx.

More details on thread management in the Interix subsystem is also available in “Thread Management” in Chapter 3, “Process and Thread Management” of this volume.

## Memory Management

The Interix subsystem supports the following memory management-related features:

- Most memory functions available in UNIX.
- Memory-mapped files by using the **mmap** function.
- All of the System V IPC mechanisms, including the shared memory routines **shmat**, **shmctl**, **shmdt**, and **shmget**.
- POSIX V IPC.

The command-line interfaces **ipcs** and **ipcrm** are also provided for the management of shared memory segments. The **ipcs** interface reports the status of interprocess communication objects. The **ipcrm** interface removes an interprocess communication identifier, such as a shared memory segment.

Interix supports all of the System V semaphores, including the shared memory routines **semctl**, **semget,** and **semop**.

**Note** More details on memory management in the Interix subsystem is available in “Memory Management” in Chapter 4, “Memory and File Management” of this volume.

# File Management

Interix supports the UNIX file management system. Interix has a single-rooted, case-sensitive file system that supports both hard and symbolic links. The single-rooted system is mapped to a multidrive Windows file system.

The Interix root directory (/) is mapped to the Windows Services for UNIX installation directory, which is typically C:\SFU. Under the root directory are typical UNIX directories, such as **/**usr, /bin, /dev, and /etc. In addition, there are also other, less familiar Interix-specific directories, such as /net, as well as directories that are a part of the Windows Services for UNIX installation, such as /admin.

You can access the drive letters mounted on your Windows system through the special /dev/fs/A through /dev/fs/Z directories. For example, to change the current directory to the root of the C drive, type **cd /dev/fs/C**. Case-sensitivity of the directory and file names is an optional choice and can be selected by the user during the installation of Windows Services for UNIX 3.5. You can access the folder only as /dev/fs/C if case-sensitivity is enabled. If case sensitivity is enabled, two files can be created on Interix that have the same name, but differ only in case. In such a case, only one file will be visible on Windows, and it is not possible to predict which one will be visible.

## File Names and Path Names

Although Windows and UNIX file systems do not allow certain characters in file names, the characters that are prohibited by each operating system are not the same. For example, a valid Windows file name cannot contain a colon (:), but a valid UNIX file name can. If a UNIX user attempts to create a file in a network file system (NFS) share on Server for NFS and if that file contains an illegal character in its name, the attempt will fail. To prevent this problem, Interix allows file-name character mapping to replace characters that are not allowed in a file system.

The following points compare the UNIX and Interix file systems and also discuss how Interix helps overcome these differences:

- UNIX systems have a file system with a single, top-level directory called the root directory. The root directory, which is referred to with a forward slash (/), contains both files and subdirectories. All files in the file system are located below the root directory (/).
- Interix has a single-rooted file system that supports both hard and symbolic links. The single-rooted system is mapped to a multidrive Windows file system. The Interix root directory (/) is mapped to the Windows Services for UNIX installation directory, which is typically C:\SFU. While viewing the contents of the root directory, Interix also displays certain virtual file systems such as /dev, /net, and /proc apart from the usual /etc and /usr that are displayed in UNIX.
- The Interix shell interprets the back slash (\) as an escape character; this causes Windows path names to work incorrectly in this environment. You can use the **winpath2unix** and **unixpath2win** tools to convert the format of a specified path name between the Interix format (such as /dev/fs/C/Cat/dog) and the Windows format (such as C:\Cat\dog).
- Path names in the Interix subsystem are case-sensitive. If Windows Services for UNIX is installed on a computer running Windows XP, the path names are case-sensitive only if you click the option of changing the default behavior of the computer to support case-sensitive file names during installation. There are instances of open source applications where multiple files differ only by case, hence turning this option on is recommended as a practice to avoid such conflicts and possible errors.
- A path name that begins with more than one forward slash (/) is treated as though it begins with just one.
- Interix does not support the Universal Naming Convention (UNC) syntax. However, Windows Services for Unix 3.5 provides a new virtual directory, /net, to access remote file systems by using names that are similar to UNC names. The standard UNC syntax of \\hostname\sharename must be changed to the Interix syntax–/net/hostname/sharename.

- The mount command-line tool mounts the file system identified by sharename exported by the network file system (NFS) server identified by ComputerName and associates it with the drive letter specified by DeviceName. When mapping a drive to a NFS share, you can choose to perform either a hard or soft mount. In both hard mount and soft mount options, the application makes a remote procedure call (RPC) to access a file on the mapped drive. In case of a hard mount, if the call times out, the client for NFS will retry the call indefinitely until it succeeds. But in case of a soft mount option (the default), if the call times out, the client for NFS will retry the call a fixed number of times. Then, if the NFS server still does not return successfully from the call, it returns an error to the calling application.
- Interix defines a line as ending with the \n character. But Windows defines a line as ending with the \r\n sequence. Some applications on Windows and Interix are sensitive to the precise line termination sequence. Windows-based programs may expect any text files to be in the Windows format (end-of-line marked by CR-LF) instead of the POSIX format (end-of-line marked by LF). You can use the **flip** tool, which is a file interchange program that converts text-file formats between POSIX and other formats (such as MS-DOS® and Apple Macintosh).

**Note** More details on file management in the Interix subsystem are also available in "File Management" in Chapter 4, "Memory and File Management" of this volume.

# *Infrastructure Services*

This section discusses the major features related to infrastructure services available in the Interix subsystem, such as security and daemons/services, and provides detailed information about their implementations in the Interix subsystem.

## Security

The security model for Interix on Windows differs from that of UNIX in the following ways:

- In Windows, either a user or a group can own an object; whereas in UNIX, only a user owns an object.
- In classical UNIX systems, a user with UID = 0 is termed as a superuser to whom all privileges are granted. Typically only the user called "root" is given this uid.
- Because the Interix subsystem is built on the POSIX specification, it does not recognize a root user. Instead of a root user, the POSIX standard defines appropriate privileges for some operations. All supported privileges on a given system are granted to all users who are members of the Administrators group on that system. Not all privileges defined in POSIX or UNIX are available under Interix, which means that there are certain privileges that are granted to no user at all and certain privileges are granted only to the super user.
- Interix does not recognize a single super user. In Interix, the Administrator account is closest in power and privileges to the root user in UNIX. Because the Security Accounts Manager database stores user and group accounts in the same database, group and user names must be unique. No group can have a name that is same as that of a user and vice versa. This is different from the single UNIX specification, in which users and groups are separate. Users can belong to many groups.
- Windows Services for UNIX 3.5 enables a Windows-based server to function as a Network Information Service (NIS) server, integrating NIS domains with Microsoft Active Directory® directory service and giving Windows Active Directory administrators the capability to manage Windows and NIS domains together.
- The client for NFS uses user name mapping to associate Windows users with user identifiers (UIDs) and group identifiers (GIDs). Mapping allows the actual user and group names to appear as the file owner and file group when a long directory listing is requested. This mapping is done solely for the purpose of communicating with NFS servers.

## Daemons and Services

On traditional UNIX systems, a daemon is a process that runs for an extended period of time but does not have a controlling terminal. A Windows service is a background process that is similar to a daemon process. A daemon can run directly on the Interix subsystem, or it can be ported to run as a Windows service.

Unlike a daemon, a Windows service logs on to the computer with a user account. This allows the administrator to have greater control over the privileges granted to the service. When the service logs on with a domain account, Windows even allows the service to access network resources.

Interix lets you take advantage of both of these mechanisms to provide services such as **inetd**, which is a "super-server" for Internet services.

The Interix programs designed to run as daemons can be controlled using the usual UNIX mechanisms, such as by sending signals using the **kill** tool.

To support running Interix programs as Windows services, Interix has a program called **service** that administers Interix services. The service program registers, installs, starts, and stops an Interix program that is running as a service. This tool can also be used for some administration activities of Win32 services such as listing the Win32 services.

Services use the program **psxrun.exe**, which is installed by the Windows Services for UNIX 3.5 setup into the Windows system32 directory. Two daemons that can also run as services, **inetd** and **syslogd**, are also provided. These two tools have specific command-line options that should be specified while invoking them. The details of the command-line options can be found in the manual pages of the tools. The manual pages can be opened from a Windows Services for UNIX 3.5 shell session. The same information is also available in the Help manual of Windows Services for UNIX 3.5.

Windows includes the Service Control Manager, which starts with Windows and runs in the background, handling services on behalf of the operating system. Services are either automatic (which means they are started on system startup) or manual (which means the user starts them). Services are controlled through the Windows Services for UNIX administration tool. You can use this tool to add, remove, view, start, and stop services.

To run a service, your account must be assigned the service logon right. This privilege is automatically given to your account the first time you start or install a service using the Interix service tool. However, it is possible for the administrator to remove this right from your account to prevent it from running services.

A service has no console display. You can stop it in Windows by opening **Services** or by using the **service** tool in Interix.

A service runs in a minimal environment, which consists of the TZ environment variable and the environment assigned to the default user of the system. Its standard input, output, and error are all redirected to /dev/null.

**Note** More details on infrastructure services in the Interix subsystem are available in Chapter 5, “Infrastructure Services” of this volume.

# *Development and Debugging Tools*

The Interix SDK provides a front end for Microsoft Visual Studio® to compile C programs. This provides a native UNIX environment for development based on a native Windows compiler. If Interix GNU SDK is also installed in the development environment, then the standard UNIX development tools, such as the GNU **gcc**, **g++, g77** compilers and the **gdb** debugger, are also available. The Interix SDK supports using Visual C++ in the compilation of C programs, but not C++ programs. The **make** tool is based on the OpenBSD version of **make**. The **lex** and **yacc** tools are based on the **flex** and BSD **yacc.**

The Interix SDK supports shared libraries. Dynamic linking is supported through standard calls—**dlopen**(), **dlsym**(), **dlclose**(), and **dlerror**(). Dynamically linked applications and shared libraries can only be created using **gcc** and the other GNU compiler tools. However, **cc** and **c89** will still produce statically linked binaries. The Interix GNU SDK should be installed in order to use the **gcc** compiler to create the shared libraries.

Developers can use the Interix GNU SDK to create real, UNIX-style .so libraries. Although similar to Windows dynamic-link libraries (DLLs), they are not the same in implementation or semantics.

For efficient debugging, the Interix GNU SDK provides the GNU debugger, **gdb**. The base installation also contains debugging tools such as **pstat** and **truss**.

# Chapter 2: Developing Phase: Process Milestones and Technology Considerations

This chapter introduces and discusses activities in the Microsoft® Solutions Framework (MSF) Developing Phase as they are performed in a migration project. The chapter addresses every code component of the solution and provides instructions on how to apply code changes for migrating to the Interix environment. This chapter looks at adapting and extending the components to meet the project requirements in the Interix environment with minimal changes.

## Goals for the Developing Phase

For a migration project, the Developing Phase is the time when you build the solution components,code, and infrastructure, as well as prepare the documentation. Typically, to achieve this primary goal, you must modify existing code in a way that enables it to work within the new environment—in this case, Microsoft Windows® Services for UNIX 3.5. Generally, when new code is written, some aspect of the original component remains unchanged—for example, exposed application programming interfaces (APIs) or specific component behavior. In this context, both modifying the existing code and developing new code are considered to be migration activities. For this reason, when MSF is applied to a migration project, the MSF Developing Phase is equivalent to the actual act of migrating.

During this phase, all roles are active in building and testing the deliverables. The team continues to identify all risks throughout the phase and to address new risks as they emerge.

The phase formally ends with the Scope Complete Milestone. At this major milestone, the team gains formal approval from the sponsor and key stakeholders that all solution elements are built and that the solution features and functionality are complete according to the functional specifications agreed upon during the Planning Phase.

Figure 2.1 shows the Developing Phase, its associated interim milestones, and its major milestone—Scope Complete.

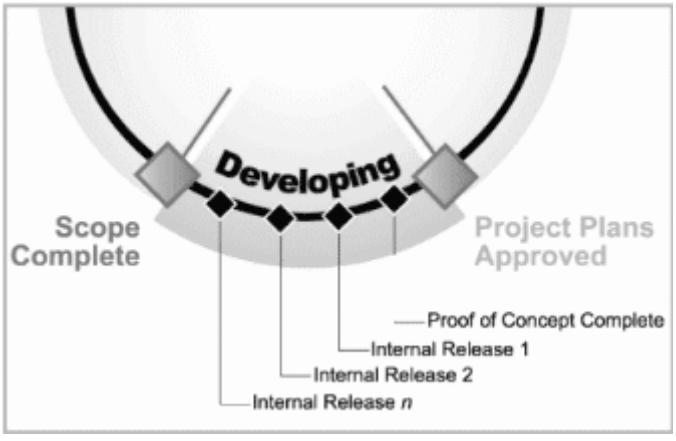


**Figure 2.1. The Developing Phase in the MSF Process Model**

Table 2.1 describes the major tasks and deliverables associated with the Developing Phase and lists which roles are responsible for them.

**Table 2.1. Major Tasks and Deliverables**

| Major Tasks and Deliverables | Owners |
|---|---|
| **Starting the development cycle**<br>The team begins the development cycle by verifying that all tasks identified during the Envisioning and Planning Phases have been completed. Their main focus during this phase is to identify and analyze any risks that might occur during the Developing Phase. | Development |
| **Building a proof of concept**<br>Before development, the team finally verifies the concepts from the designs within an environment that mirrors production as closely as possible. | Development team |
| **Developing the solution components**<br>The team develops the solution using the core components and extends them to the specific needs of the solution. The team also develops and conducts unit functional tests to ensure that individual features perform according to the specifications. | Development, User Experience, Test |
| **Developing the testing tools and tests**<br>The team develops the testing infrastructure and populates it with test cases. This ensures that the entire solution performs according to specification. This solution test suite typically incorporates, as a subset, the individual feature tests used by developers to build the solution components. | Test |
| **Building the solution**<br>A series of daily or frequent builds culminate with major internal builds and identification of points at which the development team will deliver key features of the solution. These builds are subjected to all or part of the entire project test suite as a way of tracking the overall progress of the solution and the solution test suite. | Development, Test |
| **Closing the Developing Phase**<br>The team completes all features, delivers the code and documentation, and considers the solution complete, thus entering the approval process for the Scope Complete Milestone. | Project |

**Note** Refer to the *UNIX Migration Project Guide* (UMPG) for an overview of MSF, general information on the processes that belong to each phase, and additional information about the team roles responsible for the processes. The UMPG is meant to be used in conjunction with the technical and solution-specific information in this guide.

# Starting the Development Cycle

This section focuses on identifying and addressing the risks in the Developing Phase. The Developing Phase can be the most volatile and trying, yet the most stimulating and challenging, part of any UNIX migration project. Major issues and risks become evident soon after this phase begins. For example, team members might later realize that Microsoft Windows Services for UNIX 3.5 does not support a particular function that their code is using. This can be categorized as a risk because replacement code might need to be written at that stage. Resolution of such issues is the distinguishing factor that determines if schedules will change, whether funding is sufficient, and if the project will be successful.

Design and technology choices involving various techniques and tools were discussed briefly as part of the Envisioning Phase and the Planning Phase in Volume 1: *Plan* of this guide. These same choices are discussed in detail in the chapters of this volume. If any item on the related task lists from the Envisioning Phase and the Planning Phase is not completely satisfied, it could present itself as a risk during the Developing Phase.

The following actions can help mitigate the risks:

- Prepare a requirements specification document, which details the scope of the migration project and the design and architecture to be followed.
- Perform an impact analysis of the changes and get customer sign-off on the project-execution approach for the requested changes.
- Establish the existence of and procure licenses for source code to any necessary third-party libraries.

Implementing the preceding actions is easier if the risks are identified and mitigation plans are formulated and evaluated well ahead of time. Risk mitigation, as part of the risk management process, can be used to keep a project on track in adverse situations.

**Note** Information on Microsoft Office Solution Accelerator for Six Sigma is available at http://www.microsoft.com/office/solutions/accelerators/sixsigma/default.mspx.

# Building a Proof of Concept

Typically, the proof of concept is a continuation of some initial pre-development work (the preliminary proof of concept) that occurred during the Planning Phase. Creating a proof of concept helps you to implement risk management. It also helps assess the worthiness and ease of the migration process through testing key elements of the solution in a nonproduction simulation of the proposed operational environment. Your development team tries to compile the source code and obtains a good idea of how hard the port will be after a few rounds of quick fixes and recompiles. The team guides operations staff and users through the solution to validate their requirements. Also, during such a review process, developers might discover design flaws and bugs in the original application being ported that need to be addressed.

There may be some solution code or documentation that carries through to the eventual solution-development deliverables. However, the proof of concept is not meant to be production-ready. The proof of concept is considered as throwaway development that gives the team a final chance to verify functional specification content and address any other issues before fully moving into development. The proof of concept also helps in obtaining metrics for projecting the developer effort required to port the overall application environment.

## *Interim Milestone: Proof of Concept Complete*

Reaching this interim milestone marks the point where the team is fully moving from conceptual validation to building the solution architecture and components. The proof of concept should result in at least one prototype interoperation scenario being built before production development begins. There can be a number of strategies to accommodate interoperability:

- **Create a Windows-only environment**. If the project is low risk, you may be able to run the Windows and UNIX environments in parallel just for the time it takes for users to make the transition to the migrated application.
- **Create a separate Windows environment alongside the UNIX environment**. This approach is suitable for short-term parallel running or when the application has both UNIX and Windows modules.
- **Create an integrated cross-platform environment**. This approach is ideal when the two environments have to coexist for a long time.

# Developing the Solution Components

Developing the solution components identifies the suitable development environment for migrating applications. You can use the development environment for building and debugging the applications. Using the development environment, you can make changes to the UNIX code to run in the Interix environment. Because there are certain differences between UNIX and Interix, you must modify the UNIX code for it to work in the Interix environment. This volume addresses these potential coding differences by looking at the solution from various categories, which are described in detail in subsequent chapters of this volume. For each of these categories, this volume:

- Describes the coding differences between UNIX and Interix.
- Outlines options for converting the code.
- Illustrates the options with source code examples.

These categories are:

- Process management.
- Thread management.
- Memory management.
- File management.
- Infrastructure services.
- Migrating user interface.
- Functions to change for Interix.

The following subsection describes the purpose of the development environment and how to use the development environment for building and debugging applications.

## *Development Environment*

The development environment is the environment in which the user develops and builds the solution. The development environment provides the necessary compiler, linker, libraries, and reference objects. In some cases, the integrated development environment (IDE) is also provided. Setting up the development environment is explained in “Setting Up the Development Environment” in Chapter 4, “Planning: Setting up the Development and Test Environments” of Volume 1. This section explains how to use the development environment after setting it up in the Planning Phase.

This section also discusses the important components of the development environment for Interix applications and using Interix for building and debugging an application’s environment.

## Using Interix

You can use the Interix subsystem in conjunction with the Windows Platform SDK or Microsoft Visual Studio®. The Windows Platform SDK, which provides a front end for Microsoft Visual C++®, offers the benefits of the native compiler for Windows, while retaining a UNIX development environment. But the Interix environment also offers the benefits of the GNU **gcc** and **g++** compilers.

### *Header Files Included in Interix*

Table 2.2 lists the header files that are included with Interix in the /usr/include directory and also lists the header files found in the Linux and Solaris variants. Differences are indicated with an X. By identifying the differences between Interix and other environments, you can replace the unsupported features with third-party, or user-defined, libraries when migrating the applications to the Interix environment.

**Table 2.2. Interix Header Files in /usr/include, with Linux and Solaris Variants**

| Header | Interix | Linux | Solaris |
|---|---|---|---|
| **alloca.h** | ✓ | ✓ | ✓ |
| **ar.h** | ✓ | ✓ | ✓ |
| **assert.h** | ✓ | ✓ | ✓ |
| **blf.h** | ✓ | ✓ | ✓ |
| **cast.h** | ✓ | ✓ | ✓ |
| **cpio.h** | ✓ | ✓ | ✓ |
| **ctype.h** | ✓ | ✓ | ✓ |
| **curses.h** | ✓ | ✓ | ✓ |
| **db.h** | ✓ | ✓ | X |
| **dirent.h** | ✓ | ✓ | ✓ |
| **dlfcn.h** | X | ✓ | ✓ |
| **err.h** | ✓ | ✓ | **sys/err.h** |
| **errno.h** | ✓ | ✓ | ✓ |
| **eti.h** | ✓ | ✓ | ✓ |
| **excpt.h** | ✓ | X | X |
| **fcntl.h** | ✓ | ✓ | ✓ |
| **features.h** | ✓ | ✓ | X |
| **float.h** | ✓ | X | ✓ |
| **fnmatch.h** | ✓ | ✓ | ✓ |
| **form.h** | ✓ | ✓ | ✓ |
| **fts.h** | ✓ | ✓ | X |
| **ftw.h** | ✓ | ✓ | ✓ |
| **glob.h** | ✓ | ✓ | ✓ |
| **grp.h** | ✓ | ✓ | ✓ |
| **histedit.h** | ✓ | ✓ | ✓ |

| Header | Interix | Linux | Solaris |
|---|---|---|---|
| **iconv.h** | ✓ | ✓ | ✓ |
| **langinfo.h** | ✓ | ✓ | ✓ |
| **libgen.h** | ✓ | ✓ | ✓ |
| **limits.h** | ✓ | ✓ | ✓ |
| **locale.h** | ✓ | ✓ | ✓ |
| **malloc.h** | ✓ | ✓ | ✓ |
| **math.h** | ✓ | ✓ | ✓ |
| **md4.h** | ✓ | ✓ | ✓ |
| **md5.h** | ✓ | ✓ | ✓ |
| **memory.h** | ✓ | ✓ | ✓ |
| **menu.h** | ✓ | ✓ | ✓ |
| **monetary.h** | ✓ | ✓ | ✓ |
| **mpool.h** | ✓ | X | X |
| **ndbm.h** | ✓ | X | ✓ |
| **netdb.h** | ✓ | ✓ | ✓ |
| **new.h** | X | X | X |
| **nl_types.h** | ✓ | ✓ | ✓ |
| **nl_types_private.h** | ✓ | X | X |
| **ohash.h** | ✓ | ✓ | X |
| **panel.h** | ✓ | ✓ | ✓ |
| **paths.h** | ✓ | ✓ | X |
| **poll.h** | ✓ | ✓ | ✓ |
| **pthread.h** | ✓ | ✓ | ✓ |
| **pty.h** | ✓ | ✓ | X |
| **pwcache.h** | ✓ | X | X |
| **pwd.h** | ✓ | ✓ | ✓ |
| **regex.h** | ✓ | ✓ | ✓ |
| **rmd160.h** | ✓ | ✓ | ✓ |
| **sched.h** | ✓ | ✓ | ✓ |
| **search.h** | ✓ | ✓ | ✓ |
| **semaphore.h** | ✓ | ✓ | ✓ |
| **setjmp.h** | ✓ | ✓ | ✓ |
| **sha1.h** | ✓ | ✓ | ✓ |
| **signal.h** | ✓ | ✓ | ✓ |

| Header | Interix | Linux | Solaris |
|---|---|---|---|
| **skipjack.h** | ✓ | X | ✓ |
| **stdarg.h** | ✓ | X | ✓ |
| **stddef.h** | ✓ | X | ✓ |
| **stdio.h** | ✓ | ✓ | ✓ |
| **stdlib.h** | ✓ | ✓ | ✓ |
| **string.h** | ✓ | ✓ | ✓ |
| **strings.h** | ✓ | ✓ | ✓ |
| **stropts.h** | ✓ | ✓ | ✓ |
| **syslog.h** | ✓ | ✓ | ✓ |
| **tar.h** | ✓ | ✓ | ✓ |
| **term.h** | ✓ | ✓ | ✓ |
| **termios.h** | ✓ | ✓ | ✓ |
| **time.h** | ✓ | ✓ | ✓ |
| **tzfile.h** | ✓ | X | ✓ |
| **ucontext.h** | ✓ | ✓ | ✓ |
| **ulimit.h** | ✓ | ✓ | ✓ |
| **unctrl.h** | ✓ | ✓ | ✓ |
| **unistd.h** | ✓ | ✓ | ✓ |
| **utime.h** | ✓ | ✓ | ✓ |
| **utmpx.h** | ✓ | ✓ | ✓ |
| **va_list.h** | ✓ | X | X |
| **varargs.h** | ✓ | X | ✓ |
| **vis.h** | ✓ | X | X |
| **wait.h** | ✓ | ✓ | ✓ |
| **wchar.h** | ✓ | ✓ | ✓ |
| **wctype.h** | ✓ | ✓ | ✓ |
| **xti.h** | ✓ | X | ✓ |

Table 2.3 lists the header files that are included with Interix in the /usr/include/sys directory and the header files found in the Linux and Solaris variants.

**Table 2.3. Interix Header Files in /usr/include/sys, with Linux and Solaris Variants**

| Header | Interix | Linux | Solaris |
|---|---|---|---|
| **cdefs.h** | ✓ | ✓ | X |
| **dir.h** | ✓ | ✓ | X |
| **endian.h** | ✓ | X | X |
| **errno.h** | ✓ | ✓ | ✓ |
| **fault.h** | ✓ | X | ✓ |
| **fcntl.h** | ✓ | ✓ | ✓ |
| **file.h** | ✓ | ✓ | ✓ |
| **fsid.h** | ✓ | X | ✓ |
| **ioctl.h** | ✓ | ✓ | ✓ |
| **ipc.h** | ✓ | ✓ | ✓ |
| **mkdev.h** | ✓ | X | ✓ |
| **mman.h** | ✓ | ✓ | ✓ |
| **msg.h** | ✓ | ✓ | ✓ |
| **param.h** | ✓ | ✓ | ✓ |
| **procfs.h** | ✓ | ✓ | ✓ |
| **queue.h** | ✓ | ✓ | X |
| **reg.h** | ✓ | ✓ | ✓ |
| **regset.h** | ✓ | X | ✓ |
| **resource.h** | ✓ | ✓ | ✓ |
| **select.h** | ✓ | ✓ | ✓ |
| **sem.h** | ✓ | ✓ | ✓ |
| **shm.h** | ✓ | ✓ | ✓ |
| **siginfo.h** | ✓ | X | ✓ |
| **signal.h** | ✓ | ✓ | ✓ |
| **socket.h** | ✓ | ✓ | ✓ |
| **stat.h** | ✓ | ✓ | ✓ |
| **statvfs.h** | ✓ | ✓ | ✓ |
| **stropts.h** | ✓ | ✓ | ✓ |
| **syscall.h** | ✓ | ✓ | ✓ |
| **syslog.h** | ✓ | ✓ | ✓ |
| **termios.h** | ✓ | ✓ | ✓ |
| **time.h** | ✓ | ✓ | ✓ |

| Header | Interix | Linux | Solaris |
|---|---|---|---|
| **time_impl.h** | ✓ | ✓ | ✓ |
| **timeb.h** | ✓ | ✓ | ✓ |
| **times.h** | ✓ | ✓ | ✓ |
| **types.h** | ✓ | ✓ | ✓ |
| **ucontext.h** | ✓ | ✓ | ✓ |
| **uio.h** | ✓ | ✓ | ✓ |
| **un.h** | ✓ | ✓ | ✓ |
| **unistd.h** | ✓ | ✓ | ✓ |
| **user.h** | ✓ | ✓ | ✓ |
| **utsname.h** | ✓ | ✓ | ✓ |
| **wait.h** | ✓ | ✓ | ✓ |

Table 2.4 lists the header files that are included with Interix in the /usr/include/arpa directory and also lists the header files found in the Linux and Solaris variants.

**Table 2.4. Interix Header Files in /usr/include/arpa, with Linux and Solaris Variants**

| Header | Interix | Linux | Solaris |
|---|---|---|---|
| **ftp.h** | ✓ | ✓ | ✓ |
| **inet.h** | ✓ | ✓ | ✓ |
| **nameser.h** | X | ✓ | ✓ |
| **telnet.h** | ✓ | ✓ | ✓ |
| **tftp.h** | ✓ | ✓ | ✓ |

Table 2.5 lists the header files that are included with Interix in the /usr/include/netinet directory and also lists the header files found in the Linux and Solaris variants.

**Table 2.5. Interix Header Files in /usr/include/netinet, with Linux and Solaris Variants**

| Header | Interix | Linux | Solaris |
|---|---|---|---|
| **if_ether.h** | X | ✓ | ✓ |
| **if_ieee.h** | X | X | X |
| **igmp.h** | X | ✓ | X |
| **igmp_var.h** | X | ✓ | ✓ |
| **ip.h** | ✓ | ✓ | ✓ |
| **ip_info.h** | X | ✓ | ✓ |
| **mib_kern.h** | X | ✓ | ✓ |
| **udp.h** | ✓ | ✓ | ✓ |
| **in.h** | ✓ | ✓ | ✓ |
| **ip_icmp.h** | ✓ | ✓ | ✓ |
| **ip_mroute.h** | X | ✓ | ✓ |

| | | | |
|---|---|---|---|
| **tcp.h** | ✓ | ✓ | X |
| **if_ieee.h** | X | ✓ | ✓ |
| **udp_var.h** | X | ✓ | ✓ |
| **igmp.h** | X | ✓ | ✓ |
| **in_systm.h** | ✓ | ✓ | ✓ |
| **ip_igmp.h** | X | ✓ | ✓ |
| **ip_var.h** | ✓ | ✓ | ✓ |
| **tcpip.h** | X | ✓ | ✓ |

Table 2.6 lists the header files that are included with Interix in the /usr/include/interix directory. These Interix-specific headers are useful for developers when they are looking for a particular functionality.

**Note** There are no **man** pages for the application programming interface (API) prototypes in the header files included in /usr/include/interix.

**Table 2.6. Interix Header Files in /usr/include/interix**

| Header File | Description |
|---|---|
| **registry.h** | This contains the registry paths for Interix. |
| **security.h** | This has flags for the **setuser** API. |
| **env.h** | This has various environment variables values set. |
| **interix.h** | This contains routines to get information from the registry and to convert path names, specifically winpath2unix and unixpath2win. |
| **ldsostartup.h** | This contains details for ld.so library. |
| **path_convert.h** | This contains routines to convert path names. |

**Note** The preceding tables do not list any header files that may exist on other platforms but not on Interix. However, header files present on systems other than Interix are often superseded by another header file or a set of header files. Moving to the header files present on Interix provides better portability across all UNIX platforms. Use the manual pages and the **grep** tool to find where the header file content is located.

## *Building the Application*

Applications are built in the Interix environment in the same way they are built in the UNIX environment. Consider the following when building applications in the Interix environment:

- The Interix SDK supports Visual C++ for compiling C programs. It does not support Visual C++ for compiling C++ programs. For C++ programs, the GNU **g++** compiler is provided.
- Compiler options and C language definitions may need a GNU **gcc** compiler or the **cc/c89** tools. The **cc/c89** compiler interface tools in turn invoke the Microsoft Visual C++ compiler cl.exe for compiling and Link.exe for linking. The **gcc** compiler of GNU should be used in these instances or the source code should be changed. (For example, "long long" should change to quad_t or some compiler options that are valid for **gcc** but not **c89/cc**, such as **–x** option for **gcc.**)
- The version of **make** provided with the Interix SDK is based on the Berkeley Software Distribution (BSD) 4.4 **make** and supports all BSD features. It also conforms to Portable Operating System Interface (POSIX).2. The GNU **make** (**gmake**) tool can help when porting the makefiles of the application. The **gmake** tool is not included in the Interix SDK, but it can be downloaded at http://www.interopsystems.com/tools/warehouse.htm.

- To convert existing makefiles to run under the Interix subsystem, change macro definitions. Edit the value of $(CFLAGS) to use only flags supported by **cc** or **c89**.
- The Interix SDK supports shared libraries or dynamically linked objects. Dynamic linking is supported by using standard calls, such as **dlopen**(). Dynamically linked applications and shared libraries can be created only by using **gcc** and other GNU compiler tools. The Interix GNU SDK should be installed to use the **gcc** compiler to create the shared libraries.
- A typical problem when porting a UNIX application to Interix is that (on builds) **gcc** defaults to shared libraries in the Interix SDK. To force the compiler to link to the static libraries, use the compiler option –**static**. However, **cc/c89/MSVC** can only build static binaries.
- Shared libraries should be linked to the **libc.so** library visible to them. It is usually incorrect to link shared libraries to **libc.a**.
- The Interix SDK includes the **liblock** command, which locks a library to prevent the linker from using it. If any process attempts to use a locked library to link, the **ld** command reports a fatal error. There is no tool provided to unlock a library that has been locked by using **liblock**. However, locking is vulnerable and is not a high-security option. The syntax is as follows:

```
% liblock lib.so
```

Although the **gcc** command is capable of handling these rules of linking, use extreme caution if you are invoking **ld** directly.

**Note** It is recommended that you make copies of the original libraries before using **liblock**.

## *Debugging the Application*

The **gdb** debugger can be used to debug applications that are compiled using the **gcc -g** in the Interix environment. It cannot debug binaries that are created with the **cc** or **c89.** It provides a help command for the various operations. To see a list of help issues, type **help** at the **gdb** prompt. You can also refer to the **man** page for more details.

In addition to **gdb**, the Interix SDK includes **pstat** and **truss** tools to help in debugging programs. These tools are part of the base installation.

The **pstat** tool displays detailed information about a specified process.

**Note** If you have Administrator privileges (that is, if you have signed on as Administrator and are a member of the local Administrators group), you can view any process identifier (PID). If you do not have Administrator privileges, you can only view PIDs owned by your shell (that is, launched from your shell).

For example, this is the output for PID 10187 (a C shell process):

```
% pstat 10187
pid=10187 ppid=1 pgid=10187 sid=10187 state=3 Active
flags=40000022 execed pgrp asleep
signal trampoline = 0x77EA1F66,0x77EA1F8F nullapi =
0x77EA7A07,0x77EA7A4F
current syscall=sigsuspend()
IP=77f8224d SP=0088da1c BP=0088daf8 FL=00000246 AX=00000000 BX=00000001
CX=00000000 DX=00000000 DI=0088eddc SI=0088da78
CS=0000001b DS=00000023 ES=00000023 FS=00000038 GS=00000000 SS=00000023
```

Although it does not have the same functionality, the **truss** tool was inspired by the System V tool of the same name. The **truss** tool outputs information about system calls and signals.

For example, this output represents the system calls made on behalf of the **cat** tool:

```
% truss cat config.log
tracing pid 12491
null() null returned 0
open("config.log", 0x1) open returned 3
fstat(1, 0x1210650, 0x1200650) fstat ret: 0 dev: 0x0 ino: 0x00000000
```

```
read(3, 0x8228E0, 512) read returned 127 0x7F
write(1, 0x8228E0, 127) This file contains any messages produced by
compilers
while running configure, to aid debugging if configure makes a mistake.
write returned 127 0x7F
read(3, 0x8228E0, 512) read returned 0
close(3) close returned 0
close(1) close returned 0
exit(0) process exited with status 0
```

# Developing the Testing Tools and Tests

After developing the solution components, you need to test the code changes made as part of the development. The testing process helps identify and address potential issues prior to deployment. Testing spans the Developing and the Stabilizing Phases. It starts when you begin developing the solution and ends in the Stabilizing Phase, when the test team certifies that the solution components address the schedule and quality goals in the project plan. This also involves using the automated test tools and test scripts.

Figure 2.2 illustrates the positioning of testing activities within the phases of the MSF Process Model.

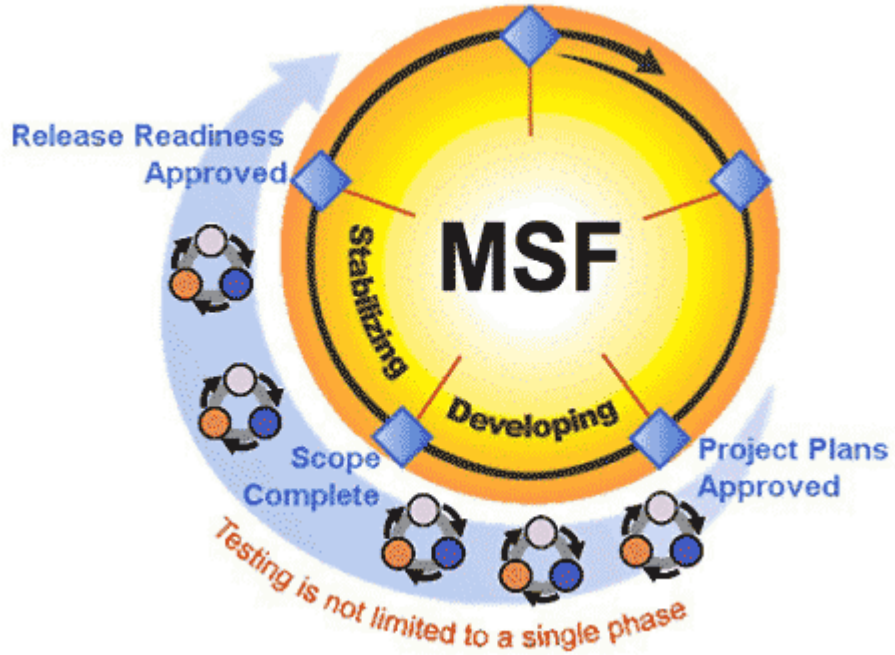


**Figure 2.2. MSF Process Model: Testing throughout the Developing and Stabilizing Phases**

This section discusses the unit testing activity that needs to be performed during the Developing Phase. The other necessary testing activities are discussed in Chapter 8, “Deployment Considerations and Testing Activities” and Chapter 9, “Stabilizing Phase” of this volume.

Testing in the Developing Phase is an integral part of the build cycle. It is not a stand-alone activity and is performed parallel with development. For instance, when building software, the development team designs, documents, and writes the code. Meanwhile, the test team designs and documents test specifications and test cases, writes automated scripts, and runs acceptance tests on components submitted for a formal round of testing.

The test team assesses the solution, makes a report on its overall quality and feature completeness, and certifies that the solution features, functions, and components address the project goals.

Testing in migration projects involving infrastructure services is focused on finding discrepancies between the behavior of the original application, as seen by its clients, and that of the newly migrated application. All discrepancies must be investigated and fixed. It is better to add any new functionality to a migrated application—or new capabilities to a migrated service—in a separate project, initiated after migration is complete.

In the Developing Phase, testing entails a code review of the application, followed by unit testing.

### *Unit Testing*

Unit testing is the process of verifying if a specific unit (which can be a class, a program, or a specific functionality) of the code is working according to its functional specifications. Unit testing also helps determine whether the specific unit will be capable of interacting with the other units as defined in the functional specifications.

Unit testing in a migration project is the process of finding discrepancies between the functionality and output of individual units in the Windows application and the original UNIX application. However, this might not always be the case; in some cases the design in Windows may differ from the UNIX design, thereby identifying units that are different from the UNIX units. Basic smoke testing, boundary conditions testing, and error testing are done based on the functional specification of the unit.

The test cases for unit testing include constraints on the inputs and outputs (pre-conditions and post-conditions), the state of the object (in case of a class), the interactions between methods, attributes of the object, and other units.

## Building the Solution

By this stage, the individual components of the solution have been developed and sufficiently tested in the Interix environment to satisfy the project requirements. This stage helps you build the solution with the developed and tested components, and then make the migrated application ready for internal release.

As a good practice, MSF recommends that teams working on development projects perform daily builds of their solution. In migration projects, on the other hand, you typically have to examine large bodies of existing code to understand what they are intended for and to make changes to this code. However, code changes can happen only after addressing porting issues, hence daily builds may not be required. The process of creating interim builds allows a team to find issues early in the development process, which shortens the development cycle and lowers the cost of the project. Note that these interim builds are not deployed in the live production environment. Only when the builds are thoroughly tested and stable are they ready for a limited pilot release to a subset of the production environment. Rigorous configuration management is essential to keeping builds in synch.

### *Interim Milestone: Internal Release*

Interim milestones help the team measure their progress in the actual building of the solution during the Developing Phase. Each internal release signifies a major step toward completion of the solution feature sets and achievement of the associated quality level. Depending on the complexity of the solution, any number of internal releases may be required. Each internal release represents a fully functional addition to the solution’s core feature set that is potentially ready to move on to the Stabilizing Phase. As each new release of the application is built, fewer bugs must be reported and triaged.

The subsequent chapters of this volume describe the necessary code changes required for the migration of the UNIX code to the Interix environment. You can use these instructions to develop the solution components in the Developing Phase.

# Chapter 3: Developing Phase: Process and Thread Management

This chapter discusses process and thread management in Microsoft® Interix and compares the implementation of these mechanisms in the UNIX and the Interix environment models. The chapter also provides workarounds for areas that are significantly different for both models. These areas include:

- Process management:
  - Creating a new process
  - Replacing a process image
  - Maintaining process hierarchy
  - Waiting for a child process
  - Managing process resource limits
  - Supporting process groups
  - Managing and scheduling the processes
- Thread management:
  - Creating new threads
  - Detaching a thread
  - Terminating a thread
  - Synchronizing threads
  - Associating thread attributes
  - Scheduling and prioritizing threads

After comparing these environments, you will be able to identify the core changes required in these areas during the migration of UNIX applications to the Interix environment. You will also learn about the equivalent Interix system calls for the UNIX system APIs.

## Process Management

The process models for the UNIX and Microsoft Windows® operating systems are very different. However, because these differences are hidden by the Interix subsystem, it is possible to migrate UNIX code to Interix with a few modifications in the process code.

The following sections discuss the similarities between UNIX and Interix process functions and highlight the modifications that must be made to the code for migration.

### *Creating a New Process*

In a UNIX environment, you create a new process using the **fork** function. The **fork** function creates a child process that is almost an exact copy of the parent process, thereby ensuring that the process environment for the child is the same as that for the parent.

Interix supports the UNIX application programming interfaces (APIs) for process creation—**fork(),** and **vfork()**. A code that uses these calls does not require any modifications to compile under Interix.

## Replacing a Process Image

A UNIX application replaces the executing image with that of another application using one of the **exec** functions.

Because Interix supports all six **exec** calls that are collectively known as **exec()**, **execl()**, **execle()**, **execlp()**, **execv()**, **execve()**, and **execvp()**, code that uses these calls does not need to be modified.

Interix supports the family of **setuid** and **setgid** APIs. The Interix **setuser()** API call is a faster and more secure replacement for **setuid/setgid** API calls. The **exec*_asuser()** APIs are deprecated and are currently wrappers to **setuser()**.

## Maintaining Process Hierarchy

In UNIX, processes have a parent-child relationship. This hierarchical arrangement is used to manage processes within applications. Interix maintains this process hierarchy and even tracks the parent-child relationship of Microsoft Win32® processes on the same system. The Interix **ps –efi** command displays processes and their relationship to each other. (The **-i** option is specific to Interix and shows the process hierarchy.) The **init** is the first process that runs when the Interix subsystem starts. It is similar to /etc/init on traditional UNIX systems, except that the Interix version does not use /etc/inittab because Interix runs only at level 2. The init process runs as process identifier 1 and always remains in the background when the system is running.

The following sample output of the **ps –efi** command shows Interix processes followed by Win32 processes:

```
UID PID PPID STIME TTY TIME CMD
...
joeuser 2049 1 Jun 4 n00 0:00.59 /bin/csh –l
joeuser 9545 2049 06:31:58 n00 0:00.01 ps –ef
...
+SYSTEM 8 0 Jun 4 S00 1:14.33 SystemProcess
+SYSTEM 152 8 Jun 4 S00 0:00.86 \SystemRoot\System32\smss.exe
+SYSTEM 176 152 Jun 4 S00 3:53.04 C:\WINNT\system32\csrss.exe C:
+SYSTEM 1040 152 Jun 4 S00 0:23.39 C:\WINNT\system32\psxss.exe C:
+SYSTEM 196 152 Jun 4 S00 0:24.47 C:\WINNT\system32\winlogon.exe
+SYSTEM 236 196 Jun 4 S00 0:48.41 C:\WINNT\system32\lsass.exe
+SYSTEM 224 196 Jun 4 S00 0:47.00 C:\WINNT\system32\services.ex
```

In some versions of UNIX, applications provide for automatic cleanup of child processes by calling **signal** as follows:

```
signal(SIGCHLD, SIG_IGN);
```

However, in the Portable Operating System Interface (POSIX) standard, the result of setting SIG_IGN as the signal handler for SIGCHLD is undefined. Therefore, you cannot use this method to provide for automatic cleanup of child processes in portable applications.

To clean up child processes in an Interix application, you need to define a signal handler for SIGCHLD and call **waitpid()** within that signal handler. The prototype of the **waitpid** function is describes as follows:

```
pid_t waitpid (pid_t wpid, int *status, int options)
```

The **waitpid** function suspends execution of its calling process until status information is available for any terminated child process. This function takes the following parameters:

- **Status**. Status of the child process.
- **Pid**. Specifies which child process to wait for.
- **Options**. A bitwise OR of flags to control the behavior of **waitpid**. The possible values of flags are WNOHANG and WUNTRACED.

A detailed explanation for the preceding is available in the Help manual of Windows Services for UNIX 3.5.

To avoid blocking within the signal handler, call **waitpid** with the first and third arguments set to -1 and WNOHANG, respectively. Because stopped child processes are not important in this case, add SA_NOCLDSTOP to sa_flags for the SIGCHLD signal.

## *Waiting for a Child Process*

Interix supports most of the wait-for-process-termination calls, including **wait**() and **waitpid()**. However, Interix does not support calls in the style of Berkley Software Distribution (BSD). When BSD-style wait calls are used, modify the code to use the suggested equivalents in Interix, which are listed in Table 3.1.

**Table 3.1. BSD-Style Wait Calls and Interix Equivalents**

| Function | Description | Suggested Interix Replacement |
|---|---|---|
| pid_t **wait3**(int *status, int options, struct rusage *rusage) | Waits for process termination. | pid_t **waitpid** (pid_t wpid, int *status, int options) |
| pid_t **wait3**(int *status, int options, struct rusage *rusage) | Waits for process termination. | **pid_t cpid = waitpid** (-1, *nstatus, options)<br>**getrusage** (cpid, *r_usage)<br>**waitpid** (pid, *status, options) |
| pid_t **wait4**(int *status, int *statusp,int options, struct rusage *rusage) | Waits for process termination. | pid_t **waitpid** (pid_t wpid, int *status, int options) |
| pid_t **wait4**(int *status, int *statusp, int options, struct rusage *rusage) | Waits for process termination. | int **getrusage** (int pid, struct rusage *r_usage) |

Functions supported by Interix are defined by the POSIX and the UNIX standards and are more portable than the forms that they replace.

Without requiring some additional steps in the ported application, combining **waitpid()** with **getrusage**() does not produce the same results as **wait3**() or **wait4**(). The idea is to capture **getrusage**(RUSAGE_CHILDREN, …) information before and after the child process is terminated and to compute the difference between the data contained in the two structures. To achieve accuracy, a child process can use **getrusage()** and communicate the same to the parent process. This method is more appropriate when there are multiple child processes.

When a program forks and the child finishes before the parent, the kernel still keeps the child process information; then the child process is said to be in a zombie state. It remains in a zombie state until it is cleaned up by its parent. In this state, the only resource it holds is a proc structure, which retains its exit status and resource usage information. The parent retrieves this information by calling **wait**, which also frees the proc structure.

## Managing Process Resource Limits

Interix supports the following three functions in UNIX:

- **getrlimit**. Returns the process resource limits.
- **getrusage**. Returns current usage.
- **setrlimit**. Sets new limits.

In addition, Interix supports the common limit names in UNIX listed in Table 3.2.

**Table 3.2. Process Resource Limit Names**

| Limit | Description |
|---|---|
| RLIMIT_CORE | 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, no core files are created. |
| RLIMIT_CPU | Maximum CPU time (in seconds) that 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 data segment. If the data segment grows larger than this value, the functions **brk**, **malloc**, and **sbrk** fail. |
| RLIMIT_FSIZE | 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 truncate call exceeds the limit, further attempts fail. |
| RLIMIT_NOFILE | Maximum value for a file descriptor, plus one. This limits the number of file descriptors a process can allocate. If more than RLIMIT_NOFILE files are allocated, functions allocating new file descriptors can fail and generate the error EMFILE. |
| RLIMIT_STACK | Maximum size (in bytes) of a process stack. The stack does not automatically grow past this limit. If a process tries to exceed the limit, the system generates the SIGSEGV error for the process. |
| RLIMIT_AS | Maximum size (in bytes) of total available memory for a process. If this limit is exceeded, the memory functions **brk**, **malloc**, **mmap**, and **sbrk** fail with **errno** set to ENOMEM, and automatic stack growth fails as described for RLIMIT_STACK. |

The other resource limit names listed in Table 3.3 are sometimes used in UNIX code and are unavailable in Interix. For these names, you need to modify the code to use the replacements that are also suggested in Table 3.3.

**Table 3.3. Process Resource Limit Names Not Available in Interix**

| Limit | Description | Suggested Interix Replacement |
|---|---|---|
| RLIMIT_MEMLOCK. | Maximum locked-in-memory address space (in bytes). | Interix has no mechanism to determine or enforce limits on this resource. |
| RLIMIT_NPROC | Maximum number of processes. | **sysconf**(_SC_CHILD_MAX) is the only Interix equivalent that provides programmatic information on process limits, but this is not an exact equivalent. |

| Limit | Description | Suggested Interix Replacement |
|---|---|---|
| RLIMIT_RSS | Maximum resident set size (in bytes) of address space in a process's address space (in bytes). | Interix has no mechanism to determine or enforce limits on this resource. |
| RLIMIT_VMEM | Maximum size (in bytes) of mapped address space in a process's mapped address space (in bytes). If this limit is exceeded, the **brk** and **mmap** functions fail with **errno** set to ENOMEM. In addition, the automatic stack growth fails as described for RLIMIT_STACK. | Interix has no mechanism to determine or enforce limits on this resource. |

## *Supporting Process Groups*

Functions in this category provide support for the management of processes as a group. Because the functions in this group are not supported by Interix, code must be modified to use the recommended replacement functions listed in Table 3.4.

**Table 3.4. Process Group Functions Not Supported by Interix**

| Function Name | Description | Suggested Interix Replacement |
|---|---|---|
| pid_t **getpgid**(0) | Gets process group ID of the calling process. | pid_t **getpgrp** (void) |
| pid_t **getpgid**(pid_t pid) | Gets process group ID for process PID. | No support or equivalent in Interix. It can be replaced with user-defined function **getpgid**. (See the description paragraph that follows the table.) |
| **Setpgrp()** | Sets process group ID of the calling process. | **setpgid(0,0)** |
| **tcgetsid** | Gets process group ID for session leader for the terminal indicated by the file descriptor. | struct utmpx ***getutxid** (const struct utmpx *id) |

As mentioned in the preceding table, Interix does not support the getpgid(pid) function, which returns the process group ID for a given process. You can obtain this information using /proc mechanism, which allows a program to retrieve a variety of information about any running process. An implementation of such a function is as follows:

```
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
extern int errno;
pid_t getpgid(pid_t pid)
{
   char procfile[25];
   char stat_rec[40];
   char inbuf[110];
```

```
char field1[10], field2[100];
FILE *in;
sprintf(procfile, "/proc/%d/stat", pid);
in = fopen(procfile, "r");
if (in == NULL)
{
errno = ESRCH; /* No such process */
return(-1);
   }
//Scan file for "pgid" entry
while(fgets(inbuf, sizeof(inbuf), in))
{
sscanf(inbuf, "%s\t%s\n", field1, field2);
if (strcmp(field1,"pgid") == 0)
   return( (pid_t) atoi(field2));
}
errno = ENOSYS; /* Function not implemented */
return(-1);
}
```

## *Managing and Scheduling the Processes*

The **getpriority**(), **setpriority**(), and **nice**() functions provide support for the scheduling and priority management of processes. These functions operate on a *nice* value, which is an integer in the range -20 to +19 where a *nice* of -20 means that the process has the highest priority.

Interix maps *nice* values to Windows process scheduling priorities according to the following rules:

- A *nice* value of 0 in Interix corresponds to the default Windows scheduling priority of 10.
- Positive *nice* values in Interix are applied as a reduction in Windows scheduling priority. For example, assigning a *nice* value of +4 to a process in Interix would result in the process being given a Windows scheduling priority of 6.
- Negative *nice* values in Interix are applied as an increase in Windows scheduling priority; a process *nice* value of -4 in Interix would cause the process to have a Windows scheduling priority of 14.

Regardless of the *nice* value, the lowest Windows priority applied by Interix to a process is 1, whereas the highest is 30. Microsoft recommends that no process be assigned a priority higher than 15—that is, a *nice* value of -5. The Interix subsystem itself runs at a Windows priority of 15. Setting a higher priority on any application yields unpredictable results. The preceding mapping of *nice* values with the Windows scheduling priority is again illustrated in Table 3.5.

**Table 3.5. *Nice* Values and Windows Scheduling Priority Mapping**

| ***Nice* Value** | **Windows Scheduling Priority** |
|---|---|
| -20 <= n <= -1 | 10 – n or 10 + abs(n) |
| 0 | 10 |
| 1 <= n <= 9 | 10 – n |
| 10 <= n <= 20 | 1 |

Any Interix process can lower the Windows priority of any process owned by the same user (increase its *nice* value). The effective user of a process must have the SE_INC_BASE_PRIORITY_NAME Windows privilege to increase the Windows scheduling priority of any process owned by the same user (decrease its *nice* value). In addition, the effective user of a process must have the SE_TCB_NAME Windows privilege to affect any process owned by any other user.

## *Terminating the Processes*

Interix supports most of the process-termination calls, including **exit**(), **_exit()**, and **kill()**.

The **exit()** function terminates the process. Before terminating the process, it calls the functions registered with the **atexit()** function, flushes the stream buffers, closes streams, and unlinks the temporary files.

The **_exit()** function also terminates the process, but it only performs the kernel cleanup of the process. The difference between them is that **exit()** also performs the cleanup of the user-mode constructs, whereas **_exit()** performs only kernel level cleanup.

The **kill()** function is used to send a termination signal to the process. This can be used to terminate a process or a group of processes. In addition to these functions, Interix also supports the **kill** command.

# Thread Management

The Interix subsystem supports much of the Pthread model and Pthread interfaces necessary for conformity to the IEEE Std1003.1-2001 standard as well as many of the Pthread, semaphore, mutex, and scheduling-specific APIs. The Interix environment also comes with updated libraries that support the thread-safe and new reentrant functions that this standard requires.

Although the Windows kernel provides much of the thread functionality and semantics required by Interix Pthreads, the function calls and syntax between the two threading models are very different.

Interix threads are implemented using the same mechanisms and functionality as Windows threads. This implies that Interix threads will share the same characteristics as Windows threads. These characteristics are listed as follows:

- **Scheduling Priorities**. Each thread can be assigned a scheduling priority in the range 1 to 31. The range 1 to 15 is known as the dynamic range and the range 16 to 31 is known as the real-time range.
- **Scheduling Policies and Algorithms**. The thread with the highest priority value is executed first.
- **System-Level Scoping**. All threads in the system are scheduled relative to each other.

Compiling Pthread programs on Interix is similar to any other UNIX platform. You must use the **–D_REENTRANT** option to ensure that all the Pthread definitions are visible at the preprocessing stage. You can use **-lpthread** as well, but that is unnecessary because all the Pthread support is incorporated into the default libraries **(libc.a** and **libpsxdll.a**).

**Note** The Interix version of Perl distributed with Windows Services for UNIX 3.5 was inadvertently built with an inappropriate implementation of Pthread support. For appropriate Pthread support, download a newer version of Perl from http://www.interopsystems.com/tools/warehouse.htm.

## *Creating New Threads*

Interix supports the **pthread_create** function of UNIX to create a new thread. This function has the following three arguments:

- A pointer to a data structure that describes the thread.
- An argument specifying the attributes of the thread (usually set to NULL indicating the default settings).
- The function that the thread will run.

The thread finishes execution with a **pthread_exit**, where it returns a string. The process can wait for the thread to complete using the **pthread_join** function. The simple UNIX example that follows creates a thread and waits for it to finish execution.

Code that uses these calls can be directly ported to Interix without any modification.

```
#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);
}
```

## Detaching a Thread

Interix supports the **pthread_detach** function of UNIX. A code that uses these calls does not require any modifications to compile under Interix.

## Terminating a Thread

Threads can terminate themselves by either returning from their thread start function or by explicitly calling the **pthread_exit**() routine. Usually, when a thread terminates, all other threads in the process continue to run. However, there is a special case— when the initial thread returns from the **main**() routine, the entire process terminates. To avoid this behavior, the initial thread can either avoid returning from the **main**() routine or call **pthread_exit**().

A thread can also be cancelled by any other thread (including itself) with the **pthread_cancel**() routine, as long as the caller knows the identity of the thread that is going to be cancelled. Each thread controls its “cancelability” state and type. The thread must cooperate and allow itself to be cancelled.

Interix supports the **pthread_exit**() and **pthread_cancel**() routines of UNIX. Code that uses these calls does not require any modifications to compile under Interix.

## Synchronizing Threads

Programs that use threads usually need to share data between the threads and need to perform various actions in a particular and coherent order across multiple threads. These operations require a mechanism to synchronize the activity of the threads. This synchronization is used to avoid race conditions and to wait for or signal when resources are available.

The Pthread model uses objects called mutexes, condition variables, and semaphores as the interthread synchronization mechanism.

The Interix Pthread implementation is built on the thread functionality already provided by the Windows kernel. Table 3.6 lists the Windows resources that are used in Interix.

**Table 3.6. Windows Resources Used in Interix**

| Pthread Object | Windows Kernel Object |
|---|---|
| Thread | Kernel thread object |
| Mutex | Kernel mutex object |
| Condition variable | Kernel event object |
| Semaphore | Kernel semaphore object |
| Read-write lock (rwlock) | Kernel rtl_resource object |

## Synchronization Using Mutexes

A Pthread mutex is an object that can be created, initialized, locked, unlocked, or destroyed. Interix supports the following UNIX routines for mutexes:

- **pthread_mutex_init**(). Use this function to initialize the mutex object.
- **pthread_mutex_lock**(). Use this function to lock the mutex object.
- **pthread_mutex_trylock**(). This function is identical to **pthread_mutex_lock**() except that if the mutex is already locked by the calling thread or any other thread, then this function returns immediately.
- **pthread_mutex_unlock**(). Use this function to unlock the mutex object.
- **pthread_mutex_destroy**(). Use this function to destroy the mutex object.

Code that uses these calls does not require any modifications to compile under Interix.

## Synchronization Using Condition Variables

A condition variable is used to communicate information about the state of shared data. It provides a mechanism to signal and wait for specific events related to data in a critical section being protected by a mutex.

Interix supports the following UNIX functions for condition variables:

- **pthread_cond_signal.** This function unblocks one or more threads that are blocked on the condition variable specified by condition, if any.
- **pthread_cond_broadcast.** This function unblocks all threads that are blocked on the condition variable specified by the condition argument.
- **pthread_cond_wait.** This function unlocks the mutex referenced by the mutex and blocks the calling thread on the condition variable.
- **pthread_cond_timedwait.** This function unlocks the mutex and blocks the calling thread on condition or until the system clock reaches or passes the absolute time.
- **pthread_cond_destroy.** This function destroys (uninitializes) the condition variable.

Code that uses these calls does not require any modifications to compile under Interix.

## Synchronization Using Semaphores

Interix supports the following UNIX functions for semaphores:

- **sem_init.** This function initializes the unnamed semaphore.
- **sem_destroy.** This function destroys the unnamed semaphore.
- **sem_post.** This function unlocks the semaphore.
- **sem_timedwait.** This function locks a semaphore with expiration time.
- **sem_trywait.** This function locks the semaphore only if the semaphore is not currently locked.
- **sem_wait.** This function locks the semaphore object.

Code that uses these calls does not require any modifications to compile under Interix.

## Associating Thread Attributes

A number of attributes are associated with threads in UNIX. Table 3.7 lists these threads along with their default values supported by Interix.

**Table 3.7. Thread Attributes and Default Values**

| fThread Attribute | Default Value Supported by Interix |
|---|---|
| detachstate | PTHREAD_CREATE_JOINABLE, PTHREAD_CREATE_DETACHED. |
| inheritsched | PTHREAD_INHERIT_SCHED, PTHREAD_EXPLICIT_SCHED. |
| schedparam | No default value. |
| schedpolicy | Although POSIX_THREAD_PRIORITY_SCHEDULING is defined, Interix only supports the SCHED_OTHER scheduling policy. It does not support the SCHED_RR or SCHED_FIFO policies. |
| Scope | PTHREAD_SCOPE_SYSTEM, PTHREAD_SCOPE_PROCESS<br>The only scheduling contention scope that Interix provides is the PTHREAD_SCOPE_SYSTEM. This means that all threads on the system are scheduled with respect to each other, regardless of the process that owns the thread. |
| Stackaddr | No default value. |
| Stacksize | No default value. |

## Scheduling and Prioritizing Threads

Interix threads are implemented using the same mechanisms and functionality as Windows threads. By relying on the Windows thread and process implementation, the Interix Pthread implementation has the following characteristics:

- Scheduling contention scope is system wide (PTHREAD_SCOPE_SYSTEM) because scheduling is per thread and not per process.
- The existing functions **nice**(), **getpriority**(), and **setpriority**() continue to handle *nice* values. These *nice* values have a range -20 to 19 and map directly to the Windows scheduling priorities in the range 30 to 1 as described in Table 3.5: *Nice* Values and Windows Scheduling Priority Mapping. Therefore, higher positive *nice* values result in less favorable scheduling priorities. The Windows scheduling priority represented by the UNIX process *nice* value is used as the value in the Windows process absolute priority.
- To access the Windows thread base-priority value, Interix uses the **sched_priority** member in the newly defined sched_param structure found in the file sched.h. The current Interix implementation maps the Pthread definition of priority to the Windows thread base-priority. Therefore, this member is only allowed the values in the range -2 to +2 when used with the following Pthread-specific functions:
  - **pthread_getschedparam()**
  - **pthread_setschedparam()**
  - **pthread_attr_setschedparam()**
  - **pthread_attr_getschedparam()**

- There are other Pthread-specific functions that deal with thread priority values. Functions such as **pthread_setschedprio()**, **sched_get_priority_max**(), and **sched_get_priority_min**() accept a priority argument. This argument is limited to the range of values between PRIO_MIN and PRIO_MAX. You can use the following statements at the beginning of the code to define the values of PRIO_MIN and PRIO_MAX:

  ```
  #define PRIO_MIN -2

  #define PRIO_MAX 2
  ```

- The combination of the Interix priority value of the thread with the Interix process *nice* value is used as the Windows thread current scheduling priority, which determines the actual scheduling priority.
- Changing the *nice* value of the process will also change the current scheduling priority in all the threads associated with that process.

**Note** The Interix implementation in Windows Services for UNIX 3.5 introduces an inconsistency because Pthread scheduling priority values are supposed to be used consistently for both threads and processes. The Windows scheduling algorithm uses more than five priority values. Programmers migrating Pthread applications must be wary of this in case the application makes assumptions between processes and thread priorities that may not be true in Interix.

# Chapter 4: Developing Phase: Memory and File Management

This chapter compares the implementation of memory and file management mechanisms in UNIX and Microsoft® Windows® Services for UNIX 3.5. In addition, this chapter discusses the functions available for memory and file management in UNIX and the corresponding application programming interfaces (APIs) available in Windows Services for UNIX 3.5.

Knowing the differences between the memory management and file management routines in the two environments will enable you to identify the memory-specific and file management-specific porting changes required for UNIX applications. You will also learn about the equivalent routines in the Interix environment for the UNIX system APIs in memory and file management.

## Memory Management

Microsoft Interix supports the majority of UNIX memory management calls; therefore, porting code by using memory management is generally straightforward. However, there are a few specific differences. This section discusses how to address these differences in your code.

### *Heap Management*

Interix supports the **alloca**() function, which allocates size bytes of space in the stack frame of the caller. However, the **alloca()** function is neither thread-safe nor async signal–safe; there might be performance issues because **alloca**() allocates memory on the stack frame instead of the heap.

Therefore, it is usually recommended that you use **malloc**(size_t size) and call **free**() because space is not automatically freed on return.

Interix does not support the platform-specific, memory-management functions listed in Table 4.1 and therefore alternative functions need to be used in the code.

**Table 4.1. Platform-Specific, Memory-Management Functions Not Supported by Interix**

| Function Name | Description | Suggested Interix Replacement |
|---|---|---|
| void **cfree**(void *) | Deallocates memory allocator. | **free**(void *ptr) |
| Int **getpagesize**(void) | Gets system page size. | Always returns 65536 (64 KB), regardless of the actual Windows page size.<br>The **getconf**(_SC_PAGE_SIZE) and **sysconf**(_SC_PAGE_SIZE) functions also return 65536 (64 KB) always. |
| void **mallocctl**(int cmd, long value) | MT hot memory allocator. | No support or equivalent in Interix. There may be open-source versions of other allocators that can be used. |
| int **mallopt**(int cmd, int value) | Provides for controls over the allocation algorithm. | No support or equivalent in Interix. The supported **malloc**() has no controllable options //mallopt. |

| Function Name | Description | Suggested Interix Replacement |
|---|---|---|
| void ***memalign**(size_t alignment, size_t size) | Allocates size of bytes on the specified alignment boundary. | No support or equivalent in Interix. |
| void * **valloc**(size_t size) | Equivalent to **memalign**(**sysconf**(_SC_ PAGESIZE),size). | void ***malloc**(size_t size) |
| **watchmalloc** | Debugs memory allocator. | No support or equivalent in Interix. |

## Memory-Mapped Files

Interix supports memory-mapped files by using the **mmap** function. The length of the mapped space (in bytes) is rounded up to the nearest multiple of **sysconf**(_SC_PAGE_SIZE). This means that the value returned by **sysconf**(_SC_PAGE_SIZE) or **sysconf**(_SC_PAGESIZE) is not the virtual-memory page size used by the system, but the value that is used by the **mmap** call. Code should work without modification unless it assumes the page sizes to be smaller than 64 kilobytes (KB). Most applications are written without any assumption regarding the page size.

## Shared Memory Management

Shared memory permits two or more processes to share a region of memory. Data present within the memory region is not copied as part of the communication process. Instead, the same physical area of the memory is accessed by both the client and the server. For this reason, shared memory performance is considered the best of all interprocess communication (IPC) methods.

Interix supports all of the System V IPC mechanisms, including the **shmat**, **shmctl**, **shmdt**, and **shmget** shared memory routines.

The **ipcs** and **ipcrm** command-line interfaces are also provided for the management of shared memory segments. The **ipcs** interface reports the status of IPC objects. The ipcrm interface removes an interprocess communication identifier, such as a shared memory segment.

## Synchronizing Access to Shared Resources

Code that uses shared memory must ensure that the processes accessing the shared memory are not attempting to access the shared memory resource simultaneously. This is particularly troublesome if one or both of the processes are writing to the same shared memory area. To address this problem, UNIX provides the semaphore object. There are two sets of functions for semaphores: The POSIX real-time extensions are used for thread synchronization, and the System V semaphores are commonly used for process synchronization.

Interix supports both POSIX real-time extensions and System V semaphores (all of them), including the shared memory routines **semctl**, **semget**, and **semop**.

The **ipcs** and **ipcrm** command-line interfaces are also provided for the management of semaphore objects. These interfaces perform the same actions as they do for the shared memory management described previously.

# File Management

The Interix file management system uses inodes to store administrative information about files and directories like UNIX. The inode is a structure that contains information about such things as the file size, its location, last access time, last modification time, and access permissions. Directories are also represented as files and have an associated inode. In addition to the file information, the inode also contains pointers to the data blocks of the file. An inode has 13 block addresses, of which the first 10 are direct block addresses for the first 10 data blocks of the file. The next addresses point to multiple-level address blocks to accommodate large files.

Interix file and data access (including security settings) differs somewhat from UNIX because of the underlying Windows input/output (I/O) system. Consequently, certain UNIX features are different or do not work in Interix.

## *Differences Between Interix and UNIX File I/O*

Interix does not support file I/O with memory caching turned off (O_DIRECT), but it supports the file I/O APIs (and their associated options) listed in Table 4.2.

**Table 4.2. File I/O APIs Supported by Interix**

| Function Name | Description |
|---|---|
| ssize_t **pread**(int fd, void *buf, size_t nbytes, off_t offset) | Reads from a file descriptor at a given offset. |
| ssize_t **pwrite**(int fd, void *buf, size_t nbytes, off_t offset) | Writes to a file descriptor at a given offset. |

## *The Interix ioctl() Function Implementation*

The **ioctl**() interface has many uses. The **ioctl**() function does not have a single standard. Its arguments, returns, and semantics vary according to the device driver. The call is used for operations that do not cleanly fit the UNIX stream I/O model.

The **ioctl**() interface in UNIX has historically been used to handle the following:

- File control (See the “File Control and **ioctl**()” section later in this chapter.)
- Socket control (See the “Socket Control and **ioctl**()” section later in this chapter.)
- Disk labels
- Magnetic tape control
- Terminal control (See the “Terminal Control and ioctl()” section later in this chapter.)

The disk label and magnetic tape I/O requests are not supported in the Interix environment.

The Windows Services for UNIX 3.5 API set contains many **ioctl**() operations, including terminal, file, and socket **ioctl**() support. The following sections explain these operations in more detail.

### Terminal Control and ioctl()

Interix supports almost all **ioctl**() requests for terminal control except a few, and some of these can be replaced with other system calls. The following are the only **ioctl**() requests that are not supported:

- TIOCNOTTY
- TIOCGETP
- TIOCSETP
- TIOCSETN
- TIOCSETC

- TIOCGETC
- TIOCLBIS
- TIOCLBIC
- TIOCLSET
- TIOCLGET
- TIOCSLTC
- TIOCGLTC
- TIOCGSID
- TIOCSSID

TIOCGETP and TIOCSETP can be replaced with POSIX Terminal I/O calls supported by Interix **cfgetispeed** and **cfsetispeed**. For more information on POSIX Terminal I/O calls, refer to "POSIX Terminal I/O" in Chapter 6, "Migrating the User Interface" of this volume.

The following three bits are not turned on in Interix, although they are held reserved with values.

- TIOCM_LE
- TIOCM_ST
- TIOCM_SR

**Note** More information on ioctl requests is available at usr/include/sys/ioctl.h.

## File Control and ioctl()

The following are the only **ioctl**() requests that are defined for file control in Interix:

- FIONREAD. To get the number of bytes available to read.
- FIONBIO. To set and unset nonblocking I/O.

The FIOCLEX and FIONCLEX requests (usually found in Filio.h) are not provided. You can replace them with the **fcntl**() FD_CLOEXEC request, as shown in the following example:

```
#ifndef __INTERIX
(void) ioctl(fd, FIOCLEX, NULL)
#else
(void) fcntl(fd, F_SETFL, fcntl(fd, F_GETFD) | FD_CLOEXEC));
#endif
```

## Socket Control and ioctl()

The only **ioctl**() request defined for sockets in Interix is SIOCATMARK. Other socket control requests are handled by using **fcntl**() or by using functions such as **setsockopt**().

**Note** Additional ioctls for sockets (for example SIOCGIFCONF and SIOCGIFFLAGS) are available using Interop Systems Porting Library.

More information on socket control and **ioctl()** is available at http://www.interix.com/tools/warehouse.htm.

# *File Security*

This section covers how files created in the two environments—namely Interix and Microsoft Win32® subsystem—differ in their security settings.

## Files Created in the Interix Environment

When you create a file in Interix and view it with **ls -l**, the following permissions and attributes apply:

- The file is owned by the user who created it.
- The file inherits its group from the group of the directory.

- Group names can contain spaces. For example, “domain users” is a valid group name. Although it is possible to create group names with spaces, if you do so, shell and **awk** scripts might not work properly because these types of scripts parse a group name as a single token.
- File permissions are dictated by the user mask.

Interix files are given three access control entries (ACEs) in Windows: one for the owner, one for the group, and one for the Everyone group, which represents everyone else. Interix permissions work as follows:

- The Interix read permission is represented by the Windows **Read** (R) permission.
- The Interix write permission is represented by the Windows **Write** (W) permission. If the read-only attribute of the file is set, Interix does not assign the write permission, regardless of the content of the ACEs. If you use the **chmod** command to assign write permission to a file that has the read-only attribute set, the read-only attribute is removed.
- The Interix execute permission is represented by the Windows **Execute** (X) permission.
- You can deny access when the Windows permission is too broad. For example, the owner should not be able to read a file with a mode of 077 (---rwxrwx). However, the Windows Everyone group gives the owner access to the file. In this case, Interix adds a deny permission to the Windows ACE for the owner to accurately represent the permissions.
- The owner of a file can change permissions on a file.
- The owner of a file can grant or deny the permission to others to take ownership of a file.
- Some special permissions, such as the **setuid** bit, are not represented through ACEs and are not visible through standard Win32 tools.
- Interix and Windows handle uppercase and lowercase letters of file names differently. Two files created under Interix that differ only in the use of case will be visible to Windows Explorer. However, access will only be given to the contents of one of the files when you try to open either from Windows Explorer. This can cause surprising behavior when interoperating between Interix and Win32.
- The **chmod** command can be used in the Interix environment to change file modes and set specific access levels for the owner, group, and everyone for a particular file. There is no equivalent of the **chattr** command in Interix.

**Note** Case-sensitivity in the Interix environment is the user choice selected during the installation of Windows Services for UNIX 3.5.

## Files Created in the Win32 Subsystem

A file created in the Win32 subsystem can have a number of ACEs associated with it. In addition, those ACEs might not fit neatly into the categories of user, group, and everyone else. The Interix tools “assemble” permission from the available UNIX ACEs in the following ways:

- For all Windows operating systems except Windows XP Professional, the user is the owner of the file unless the user is a member of the Administrators group. In this case, the Administrators group owns the file. For Windows XP Professional, the user who created the file is the default owner. To change the default owner to the Administrators group, use the Group Policy snap-in.
- The group for the file will be Windows Domain Users on a server or None on a workstation.
- If an owner has no specific ACE associated with it, the owner-permission bits are empty.
- If the owner is a group, group permissions are transferred to owner permissions and the group permissions are made empty.
- If the ACE used to determine the permissions of the owner does not have **Change Permissions** (P) or **Take Ownership** (O) permission, the **chown**, **chgrp**, and **chmod** commands might not work as expected.

**Note** When a file created in Win32 is viewed from an Interix application, the additional ACE information available is indicated by a flag being set. This flag appears as a plus sign (+) suffix, with the permissions visible by using **ls -l**.

In addition, the structure st_mode (the structure returned from **stat**) has a bit set indicating the additional ACE information: bit S_ADDACE in the file sys/stat.h. The prototype of stat function is as follows.

int stat (const char *path, struct stat *sb)

File Explorer should not be used to adjust the permissions on an Interix file or directory. File Explorer rearranges the order of the ACEs in the ACL, which can cause problems when viewing the permissions later from Interix.

# *Directory Operations*

Interix supports a subset of the routines used to access directory entries. Code that uses the calls listed in Table 4.3 does not require any modifications to compile under Interix.

**Table 4.3. Routines for Accessing Directory Entries Supported by Interix**

| Routine Name | Description |
|---|---|
| int **alphasort(**const void *d1, const void *d2**)** | Routine that can be used for the *compar* argument of the **scandir** routine to sort the array alphabetically. |
| int **scandir(**const char *dirname, struct dirent ***namelist, int (*select)(struct dirent *), int (*compar)(const void *, const void *)**)** | Scans a directory for matching entries. |

Interix does not support the calls and options listed in Table 4.4. Use the suggested Interix replacements instead.

**Table 4.4. Directory Operations Routines Not Supported by Interix**

| Function Name | Description | Suggested Interix Replacement |
|---|---|---|
| int **getdents**(int fd, struct dirent *dirp, int nbytes) | Gets directory entries and puts them in a file system-independent format. | struct dirent * **readdir**(DIR *dirp) |
| **int** getdirentries**(int fd, char *buf, int nbytes, long *basep)** | Gets directory entries in a file system-independent format. | struct dirent * **readdir** (DIR *dirp) |

## *Working Directory*

Interix does not support the routines used to obtain the current working directory. Instead, use the suggested Interix replacements listed in Table 4.5.

**Table 4.5. Working Directory Routines Not Supported by Interix**

| Function Name | Description | Suggested Interix Replacement |
|---|---|---|
| char * **get_current_dir_name** (void) | Gets absolute path of current working directory (define:__USE_GNU). | char ***getcwd**(char *buf, size_t size) |
| char ***getwd**(char* path_name) | Gets absolute path of current working directory | char ***getcwd**(char *buf, size_t size) |

| Function Name | Description | Suggested Interix Replacement |
|---|---|---|
| | (define:__USE_BSD). | |

The **getwd** API as defined on Berkeley Software Distribution (BSD) systems is particularly dangerous because it is vulnerable to buffer overrun attacks. Replace it with **getcwd** and a buffer of known size.

## *File System Operations in Interix*

File system operations in Interix differ in a number of ways from file system operations in UNIX. Some functions are not supported, such as **sync**, **sysfs**, and **ustat**. Others have different parameters or use a different set of options for UNIX. Table 4.6 lists the file system information functions that need to be replaced.

**Table 4.6. File System Information Functions Not Supported by Interix**

| Function | Description | Suggested Interix Replacement |
|---|---|---|
| int **statfs**(const char *path, struct statfs *buf)**,**<br>**int** fstatfs**(int fd, struct statfs *buf)** | Gets file system statistics. | int **statvfs**(const char *path, struct statvfs *buf)<br>int **fstatvfs**(int fildes, struct statvfs *buf) |
| void **sync**(void)<br>int **fsync**(int fd) | Writes all information in memory that should be on disk, including modified super blocks, modified inodes, and delayed block I/O. **fsync** synchronizes changes to a file. | int **fsync**(int fd) |
| int **sysfs**(int opcode, const char *filename) | Gets file system type information. | int **statvfs**(const char *path, struct statvfs *buf) |
| int **ustat**(dev_t dev,struct ustat *buf) | Gets file system statistics. | int **statvfs**(const char *path, struct statvfs *buf) |

The Windows NTFS file system is structured and implemented in such a way as to prevent the need for the sync function. NTFS uses a log-based mechanism to ensure that the file system metadata (the equivalent of super blocks and inodes) is updated in sync with changes to file data. In the event of a system failure, automated NTFS recovery guarantees that either the metadata corresponds to data appearing in files or the changes to the files never occur. Chkdsk is the tool on the Windows environment similar to the UNIX **fsck** command. This command starts and executes on the Windows environment to synchronize the file system in case of a power disruption. Windows NTFS is a journaled file system (JFS) that contains its own backup and recovery capability. It maintains a log, or a journal, of what activity has taken place in the main data areas of the disk. Interix does not have its own file system and uses the Windows NTFS file system for its operations.

When using the **statvfs** function in Interix, be aware that the **statfs** structure has some members that are different from those usually found in UNIX. Table 4.7 summarizes these differences. In general, references to the **statfs** structure can by replaced with references to the **statvfs** structure.

**Table 4.7. Differences Between UNIX statfs and Interix statvfs**

| UNIX statfs Structure | Interix statvfs Structure | Description |
|---|---|---|
| long f_type | unsigned long f_type | Type of file system. |
| long f_bsize | unsigned long f_bsize | Transfer block size. |
| long f_blocks | unsigned long f_blocks | Total data blocks in file system. |
| long f_bfree | unsigned long f_bfree | Free blocks in file system. |
| long f_bavail | unsigned long f_bavail | Free blocks available to all users except superusers. |
| long f_files | unsigned long f_files | Total file nodes in file system. (Currently returns 0.) |
| long f_ffree | unsigned long f_ffree | Free file nodes in file system. (Currently returns 0.) |
| fsid_t f_fsid | unsigned long f_fsid | File system ID. |
| long f_namelen | unsigned long f_namemax | Maximum length of file names. |
| long f_spare[6] | unsigned long f_flag | Bit mask of values describing the file system. |
| | unsigned long f_frsize | Fundamental block size. |
| | unsigned long f_favail | Total number of file serial numbers available to a process without the required privileges. (Currently returns 0.) |
| | unsigned long f_iosize | Optimal transfer block size. |
| | char f_mntonname [MNAMELEN+1] | Mountpoint for the file system. |
| | char f_mntfromname [MNAMELEN+1] | Mounted file system. |

When using **statvfs**, the file system types supported by Interix differ from those supported by most implementations of UNIX. The following two lists illustrate this by comparing a common subset of UNIX-supported file systems with those supported in Interix. It is rare for these differences to have an impact on the migration of an application.

### *Commonly Supported File Systems in UNIX*

- AFFS_SUPER_MAGIC 0xADFF
- EXT_SUPER_MAGIC 0x137D
- EXT2_OLD_SUPER_MAGIC 0xEF51
- EXT2_SUPER_MAGIC 0xEF53
- HPFS_SUPER_MAGIC 0xF995E849
- ISOFS_SUPER_MAGIC 0x9660
- MINIX_SUPER_MAGIC 0x137F /* original minix fs */
- MINIX_SUPER_MAGIC2 0x138F /* 30 char minix */
- MINIX2_SUPER_MAGIC 0x2468 /* minix V2 */
- MINIX2_SUPER_MAGIC2 0x2478 /* minix V2, 30 char names */

- MSDOS_SUPER_MAGIC 0x4d44
- NCP_SUPER_MAGIC 0x564c
- NFS_SUPER_MAGIC 0x6969
- PROC_SUPER_MAGIC 0x9fa0
- SMB_SUPER_MAGIC 0x517B
- XENIX_SUPER_MAGIC 0x012FF7B4
- SYSV4_SUPER_MAGIC 0x012FF7B5
- SYSV2_SUPER_MAGIC 0x012FF7B6
- COH_SUPER_MAGIC 0x012FF7B7
- UFS_MAGIC 0x00011954
- XFS_SUPER_MAGIC 0x58465342
- _XIAFS_SUPER_MAGIC 0x012FD16D

### *Supported File System Types in Interix*

- ST_FSTYPE_UNKNOWN 0 /* unknown */
- ST_FSTYPE_NTFS 1 /* NTFS */
- ST_FSTYPE_OFS 2 /* OFS-NT object FS */
- ST_FSTYPE_CDFS 3
- ST_FSTYPE_CDROM ST_FSTYPE_CDFS
- ST_FSTYPE_ISO9660 ST_FSTYPE_CDFS
- ST_FSTYPE_HPFS 5 /* OS2 HPFS */
- ST_FSTYPE_SAMBA 6 /* Samba FS */
- ST_FSTYPE_NFS 8 /* NFS */

## File System Mount Entry Management

Interix supports dynamically mounted file systems, but it provides no mechanisms to mount or dismount them. These operations must be performed through Win32 tools or APIs.

Mount tables are stored in different files on different implementations of UNIX. They are usually stored under the /etc directory and have names such as **mtab** and **fstab** or **mnttab** and **vfstab**. This is unlikely to affect the migration of an application. However, if the application uses dynamically mounted file systems, the code can be altered using the following options:

- Use the /net file system to refer to the resource.
- Use the **system**() API to invoke a Windows command-line tool that can mount a file system or network share.
- Change the application and its environment so that the file system required is always mounted.

On UNIX systems, additional disk space is made available at some particular point in the file hierarchy by mounting a new disk volume onto a directory or by replacing the directory with a symbolic link to some already mounted volume. The first operation can be performed in Windows through the use of the Windows Disk Management tool. This solution makes the additional space visible to Interix and Win32 applications. If your application environment does not require Win32 applications to see this additional space, you can instead replace the directory with a symbolic link to any directory on any other volume on the system.

## Gdbm Library

GNU dbm (**gdbm**) is a library of routines that manages data files that contain key/data pairs. Applications still use the **gdbm** database (that is, an indexed file storage system), which is good at storing relatively static indexed data.

The following APIs are used by an application that uses the **gdbm** database:

- **gdbm_close()**
- **gdbm_delete()**
- **gdbm_exists()**
- **gdbm_fdesc()**
- **gdbm_fetch()**
- **gdbm_firstkey()**
- **gdbm_nextkey()**
- **gdbm_open()**
- **gdbm_reorganize()**
- **gdbm_setopt()**
- **gdbm_store()**
- **gdbm_strerror()**
- **gdbm_sync()**

The following example of application code describes the usage of some these functions.

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

GDBM_FILE dbf;
datum key, nextkey, content;
int ret;

/* opening the database See gdbm(2) manual page for details. */
dbf = gdbm_open ("test.db", BLOCK_SIZE, GDBM_WRCREAT, 0755, NULL);
if (dbf == NULL)
{
      /* if the open fails, print a standard error and exit */
      perror ("error in creating db file: test.db\n");
      return 0;
}

/* create a record */
key.dptr = "SWEngineer";
key.dsize = strlen(key.dptr);
content.dptr = "Title: Software Engineer Salary: $3000";
content.dsize = strlen(content.dptr);

/* write the record to the database */
ret = gdbm_store(dbf, key, content, GDBM_INSERT);
```

```
if (ret != 0)
{
      perror ("error in writing to database: test.db");
      return 0;
}

/* make another record */
key.dptr = "Programmer";
key.dsize = strlen(key.dptr);
content.dptr = "Title: Programmer Salary: $2700";
content.dsize = strlen(content.dptr);

/* write it to the database */
ret = gdbm_store(dbf, key, content, GDBM_INSERT);
if (ret != 0)
{
      perror ("error in writing to database: test.db");
      return 0;
}

memset(&key,0,sizeof(datum));
/* iterate through the set of records */
key = gdbm_firstkey ( dbf );
key.dptr[key.dsize] ='\0';
while ( key.dptr )
{
      if (gdbm_exists(dbf, key))
      {
            memset(&content,0,sizeof(datum));
            content = gdbm_fetch(dbf, key);
            content.dptr[content.dsize] ='\0';
      if (!strcmp(key.dptr,"Programmer"))
            {
             gdbm_delete ( dbf, key );
            }
      }
      nextkey = gdbm_nextkey (dbf, key );
      key = nextkey;
      key.dptr[key.dsize] ='\0';
}

/* closing the database */
gdbm_close(dbf);
```

The **gdbm** library is available at

http://www.interopsystems.com/tools/warehouse.htm.

The GNUs **gdbm** has been ported to Windows Services for UNIX 3.5, and the routines work like the UNIX **dbm** routine.

# Chapter 5: Developing Phase: Infrastructure Services

This chapter discusses the infrastructure services, which consist of the following features:

- Security
- Error handlers
- Signals
- Interprocess communication
- Networking
- Daemons and services

This chapter describes the implementation of the preceding services in Interix and provides a detailed comparison with the corresponding implementation in UNIX. Using the information provided in this chapter, you can identify the incompatibilities for your applications in these areas and also learn about the suggested replacement mechanisms in the Interix environment.

## Security

The UNIX and Microsoft® Windows® security models are quite different. Some of these differences are covered in the ”Architectural Differences Between UNIX and Interix” section in Chapter 1, “Introduction to Windows Services for UNIX 3.5” of this volume. The following subsections cover the differences between the various security models and describe how to modify your code to operate in Windows.

### *File System Security*

Windows and UNIX both control the access to the files and directories.

In UNIX, the directory entry for each file or directory includes a bitmap of 12 bits, known as file-mode bits. Of these, three bits control access by the file's owner, three bits control access by the owner's primary group, and three control access by everyone else.

In Windows, files and directories on NTFS partitions are protected by a discretionary access control list (DACL) consisting of one or more access control entries (ACEs). Each entry assigns or denies permissions to a user or group, unlike with UNIX. However, the number of permissions that can be granted or denied is quite extensive and provides for a much finer degree of control over the access allowed to the user or group. In addition, ACEs can be added to the DACL for any number of users or groups, allowing the file's owner complete control over who can and cannot access the file.

Windows Services for UNIX 3.5 gives Windows users the ability to access files on UNIX-based servers using the network file system (NFS) protocol and, likewise, gives users of UNIX-based computers the ability to access files on Windows-based servers running Server for NFS. In addition, Interix provides a UNIX-like environment for Windows users, giving Interix users the ability to access and manage files on the Windows-based computer using UNIX tools. Because the same file, whether it is located on a Windows-based computer or a UNIX-based computer, can be accessed by both Windows and UNIX users, it is important to understand how UNIX file security and Windows file security interact through Windows Services for UNIX.

## User-Level Security

Server for NFS security is a combination of Windows security and the security protocols that protect NFS shares. Both Windows security and NFS security control access to files by individual users, based on the account the user uses to log on. In addition, NFS security controls access to shared directories by specific client computers. Server for NFS enforces both types of access control. With Server for NFS, you can control access by users and groups to NFS resources. Controlled access is automatically enabled if Server for NFS is running on a Microsoft Windows Server™ 2003-based computer and the functional level of your domain or forest is Windows Server 2003. Otherwise, you must install Server for NFS Authentication on the primary and backup domain controllers of all domains containing users who might require access. (If you do not use Server for NFS Authentication and the functional level of your domain is not the Windows Server 2003 family, all users will access NFS resources as anonymous users.) You must also install User Name Mapping on one computer in your network to associate Windows user accounts with UNIX user accounts.

## Process Level Security

Each Interix process has two process identifiers (PIDs). The reasons for the difference between the Interix PIDs and the Windows PIDs are as follows:

- When an Interix process calls the **exec()** function, the new executable program must appear to all Interix processes as if it were the same process. For example, the PID for the new program must be the same as the PID for the old program that called the **exec()** function.

  However, Windows requires that the new process have a different PID than the previous one. When Windows starts the new program, it treats the new program as a new process. There is a very brief interval where both the old and the new processes exist at the same time. The old Windows PID is still in use when the new program is created. Therefore, the old Windows PID cannot be assigned to the new program.
- While a Portable Operating System Interface (POSIX) process group exists, the Interix PID that corresponds to the POSIX group identifier (GID) cannot be reused.

  However, if a process with a particular Windows PID exits and then a new process is created, Windows can reuse that particular PID and assign it to the new process.

For situations in which programs need to use resources to which the current user does not have access, UNIX systems run the program as a specific user or group instead of as the current user or group. UNIX programs can use special permissions, called the **setuid** and **setgid** bits, on the executable file of the program.

According to the POSIX standard, the permissions for a file include bits to set a UID (**setuid**) and a GID (**setgid**). If either or both bits are set on a file and a process executes that file, the process runs with an identity based on the UID or GID of the file respectively.

When an Interix process executes a file that has the **setuid** or **setgid** bit set, Interix constructs local security tokens for the process with the privileges assigned to the owner (if **setuid** is set) or group (if **setgid** is set) of the file. Other computers on the network do not recognize these tokens because the tokens are local. Even if the file is owned by a member of the Domain Admins group, the process still does not have trusted access to other computers in the domain.

For example, if a process executes a program file that has its **setuid** bit set and the file is owned by a member of the Domain Admins group, and if that program attempts to change the password of a domain user, the attempt fails because the security tokens of the process are local and the program is not recognized by other systems in the domain. On the other hand, if the program attempts to change the password of a local user, the attempt succeeds because the owner of the file is a member of the Domain Admins group, which typically belongs to the Administrators group of the local computer.

On UNIX systems, the user and group owner of a program is often set to root, thereby allowing a nonroot user to run that program with root privileges. On Interix, the owner should be a Local Administrator or a member of the Domain Admins group.

To summarize, UNIX and Interix systems maintain at least two user and group IDs—the effective user ID and effective group ID, and the real user ID and real group ID. Most UNIX and Interix systems also support a saved set user ID and a saved set group ID.

**Note** As a security measure, Windows Services for UNIX 3.5 can be installed with the **setuid** capabilities disabled. Also, with Interix shell scripts, the **setuid** or **setgid** bits will be ignored. Any change in the file ownership will clear these bits so that the program cannot "accidentally" be run as the new owner.

**To enable the execution of files with setuid or setgid mode bits set**

1. Insert the Windows Services for UNIX CD into the CD-ROM drive.
2. At the command prompt, type:

   ```
   regini cd_drive:\setup\enablesetuid.ini
   ```

   where *cd_drive* is the drive letter assigned to the CD-ROM drive.

   For the changes to take effect, you must restart your computer.

**To restore default Interix behavior**

1. Insert the Windows Services for UNIX CD into the CD-ROM drive
2. At the command prompt, type:

   ```
   regini cd_drive:\setup\disablesetuid.ini
   ```

# Error Handlers

The Interix subsystem does not necessarily use the same error numbers that are used in traditional systems, hence you must always use the symbolic names defined in the errno.h file.

Almost all the system calls can return an error number in the external variable **errno**, which is defined in the errno.h file. When a system call detects an error, it returns an integer value indicating failure (usually -1) and sets the variable **errno** accordingly. Successful calls never set **errno**. After the **errno** is set, it remains the same until another error occurs. It should only be examined after an error.

When a system call returns -1, the calling function can interpret the failure and take action accordingly. Numerous system calls overload the meanings of these error numbers, so these meanings must be interpreted according to the type and circumstances of the call.

Interix uses POSIX UNIX error reporting from system calls by returning a non-zero integer, usually -1, and **err**, and assigning an integer value to variable **errno**. Refer to the Help manual of Windows Services for UNIX 3.5 for further information about **err**, **verr**, **errx**, **verrx**, **warn**, **vwarn**, **warnx**, **vwarnx**, **errno**, and **strerror**. The prototype of these functions is defined as follows.

```
void err (int eval, const char *fmt, ...)
void verr (int eval, const char *fmt, va_list args)
void errx (int eval, const char *fmt, ...)
void verrx (int eval, const char *fmt, va_list args)
void warn (const char *fmt, ...)
void vwarn (const char *fmt, va_list args)
void warnx (const char *fmt, ...)
void vwarnx (const char *fmt, va_list args)
char * strerror (int errnum)
```

It is up to the individual program (that makes a system call) to handle the error conditions from that call. For example, ksh assigns the built-in variable **$?** with the exit condition, and the value of **errno** is assigned to *$ERRNO*. The shell has some built-in error handling and is reported through **stderr**, but most of the error handling must be taken up by the programmer. You can get a string interpretation of the value of **errno** through **strerror**. Usage for other shells, commands, and programs will vary.

# Signals

Signals are software interrupts that catch or indicate different types of events. This section describes the various signals and signal-handling routines. It also lists platform-specific signal functions that are not supported in Interix.

**Table 5.1. POSIX-Supported Signals**

| Signal Name | Description | Default Action/Effect | Number |
|---|---|---|---|
| SIGABRT | Abnormal termination | Terminate process | 6 |
| SIGALRM | Time-out alarm | Terminate process | 14 |
| SIGBUS | Bus error | Terminate process | 10 |
| SIGCHLD | Change in status of child | Ignore | 18 |
| SIGCONT | Continues stopped process | Ignore | 25 |
| SIGFPE | Floating-point exception | Terminate process | 8 |
| SIGHUP | Hang up | Terminate process | 1 |
| SIGILL | Illegal hardware instruction | Terminate process | 4 |
| SIGINT | Terminal interrupt character | Terminate process | 2 |
| SIGIO | I/O completion outstanding | Ignore | 19 |
| SIGKILL | Termination | Terminate process (cannot be caught or ignored) | 9 |
| SIGPIPE | Write to pipe with no readers | Terminate process | 13 |
| SIGPOLL | Pollable event (Sys V)–synonym of SIGIO | Ignore | 22 |
| SIGPROF | Profiling timer alarm | Terminate process | 29 |
| SIGQUIT | Terminal quit character | Terminate process | 3 |
| SIGSEGV | Invalid memory reference | Terminate process | 11 |
| SIGSTOP | Stop process | Stop process (cannot be caught or ignored) | 23 |
| SIGSYS | Invalid system call | Terminate process | 12 |
| SIGTERM | Software termination | Terminate process | 15 |
| SIGTRAP | Trace trap | Terminate process | 5 |
| SIGTSTP | Terminal stop character | Stop process | 24 |
| SIGTTIN | Background read from control TTY | Stop process | 26 |
| SIGTTOU | Background write to control TTY | Stop process | 27 |
| SIGURG | Urgent condition on socket | Ignore | 21 |
| SIGUSR1 | User-defined signal | Terminate process | 16 |
| SIGUSR2 | User-defined signal | Terminate process | 17 |
| SIGVTALRM | Virtual time alarm | Terminate process | 28 |
| SIGXCPU | CPU time limit exceeded | Terminate process | 30 |

| Signal Name | Description | Default Action/Effect | Number |
|---|---|---|---|
| SIGXFSZ | File size limit exceeded | Terminate process | 31 |

Interix supports most of the signal-handling functions. However, it does not support some nonstandard, platform-specific implementations, such as **sigfpe** and signal handling for specific SIGFPE codes.

Table 5.2 lists the platform-specific functions that are not supported by Interix along with the Interix substitutes to be used (if they exist).

**Table 5.2. Platform-Specific Signal Functions Not Supported by Interix**

| Function name | Description | Suggested Interix replacement |
|---|---|---|
| **bsd_signal** | Simplified signal facilities. | Refer to sample code in "UNIX **bsd_signal** Code Replacement" immediately following this table. |
| **getcontext** | Gets current user context. | No support or equivalent in Interix. |
| **gsignal** | Software signals. | No support or equivalent in Interix. |
| **makecontext** | Manipulates user contexts. | No support or equivalent in Interix. |
| **psiginfo** | Software signals. | No support or equivalent in Interix. |
| **sig2str** | Translates the signal number *signum* to the signal name. | char ***strsignal**(int signal) |
| **sigaltstack** | Sets or gets signal alternative stack context. | No support or equivalent in Interix. |
| **sigfpe** | Handles signals for specific SIGFPE codes. | unsigned int _controlfp(unsigned int new, unsigned int mask) |
| **siggetmask** | Gets the current set of masked signals. | Use int **sigprocmask**(int how, const sigset_t *set, sigset_t *oset) |
| **siginterrupt** | Allows signals to interrupt functions. | Controlled by the SA_RESTART flag passed to **sigaction()**. |
| **sigsend** | Sends a signal to a process or a group of processes. | No support or equivalent in Interix. |
| **sigsendset** | Sends a signal to a process or a group of processes. | No support or equivalent in Interix. |
| **sigstack** | Sets and/or gets alternative signal stack context. | No support or equivalent in Interix. |
| **ssignal** | Software signals. | No support or equivalent in Interix. |
| **str2sig** | Translates the signal name *str* to a signal number. | Write a simple table lookup routine. (See Table.) |
| **Swapcontext** | Manipulates user contexts. | No support or equivalent in Interix. |
| **Psignal** | System signal messages. | Maps closely to **strsignal**. |
| **setcontext** | Sets current user context. | Maps closely to **setuser**. The prototype of the function is as follows:<br>int **setuser**(char *username, char *password, int flags). |

| Function name | Description | Suggested Interix replacement |
|---|---|---|
| **sys_siglist** | System signal messages. | Change the code to use strsignal() instead. |
| **sys_signame** | System signal messages. | Change the code to use strsignal() instead. |

## *UNIX bsd_signal Code Replacement*

Code that uses the **bsd_signal**() function should be implemented using other signal functions in Interix.

The **bsd_signal**(sig, func) function call can be implemented as follows:

```
#include <signal.h>
void (*bsd_signal(int sig, void (*func)(int)))(int)
{
  struct sigaction act, oact;

  act.sa_handler = func;
  act.sa_flags = SA_RESTART;
  sigemptyset(&act.sa_mask);
  sigaddset(&act.sa_mask, sig);
  if (sigaction(sig, &act, &oact) == -1)
     return(SIG_ERR);
  return(oact.sa_handler);
}
```

This code can support calls to the **bsd_signal** function in a migrated application. You can also use the code to replace **signal**() in Berkeley Software Distribution (BSD)-derived applications as long as the signal handler expects a single parameter of the **int** type. If the handler expects any other parameters, **signal**() must be modified to use **sigaction**().

**Additional Signal Functions**

Interix provides the following POSIX functions for manipulating the signal set:

```
int sigemptyset(sigset_t * set);
int sigfillset(sigset_t * set);
int sigaddset(sigset_t * set, int signo);
int sigdelset(sigset_t * set, int signo);
int sigismember(const sigset_t * set, int signo);
```

The following example depicts the manipulation of the signal mask using **sigpprocmask**():

```
int sigprocmask(int how, sigset_t * set, sigset_t *oset);
```

You can determine whether a signal is pending by using the **sigpending**() function, as shown in the following example:

```
int sigpending(sigset_t * set);
```

# Interprocess Communication

An operating system designed for multitasking or multiprocessing must provide mechanisms for communicating and sharing data between applications or interprocess communication (IPC). Some forms of IPC are designed for communication among processes running on the same computer, whereas other forms are designed for communicating across the network between different computers.

## *Pipes (Unnamed/Named, Half/Full Duplex)*

Interix supports all the various forms of IPC. The following forms are most familiar to UNIX developers:

- Anonymous pipes
- Named pipes (FIFOs)
- Shared memory
- System V message queues

### Anonymous Pipes

The primary use of pipes, which can be named or unnamed, is to communicate between related processes. They also have separate read and write file descriptors, which are created through a single function call. Process pipes are supported under Interix using the standard C run-time library. Interix supports all the pipe function calls, including **popen**, **pclose**, and **pipe**. There is no need to change any references to these calls in your code. Pipes are frequently used between UNIX processes to connect the standard output file descriptor of one process to the standard input file descriptor of a second process, causing the results of the first program to be treated as the input data for the second. This sequence of commands is called a pipeline.

You can use this mechanism to connect an Interix process to a Microsoft Win32® process that it creates. In nearly all cases, everything works without any change. Problems using pipes to communicate between Interix and Win32 processes generally fall into the following two categories:

- **Line termination character**. Interix defines a line as ending with the \n character; Win32 defines a line as ending with the \r\n sequence. Some applications are sensitive to the precise line termination sequence. The **flip** command can be used in the pipe to change line termination as necessary.
- **End Of File (EOF) handling when attempting serial use of a pipe**. After a Win32 process has closed a pipe to which it was writing, it is not possible for an Interix application to use that pipe serially. For example, this command will display only the contents of file1:

  ```
  (cmd.exe /c type file1; cat file2) | cat
  ```

  This problem can be solved by introducing a second pipeline using the **cat32** tool:

  ```
  (cmd.exe /c type file1; cat file2 | cat32) | cat
  ```

With unnamed pipes, a parent process creates a pipe to communicate with its child process; the child process inherits and uses this pipe.

### Named Pipes (FIFOs)

A named pipe, also referred to as first-in-first-out (FIFO), is a special type of pipe that is created in the file system but behaves like a process pipe. These are generally half-duplex pipes because they support only one-way communication. Full-duplex pipes are named pipes that allow two-way communication.

Interix supports the two function calls for creating a named pipe—**mknod** and **mkfifo**. If possible, it is better to use **mkfifo** to make FIFO special files because it is more portable. You do not need to modify code that uses these functions for compiling under Interix.

These named pipes are distinct from, and not interoperable with, the identically named Win32 interprocess communication mechanism, called Win32 named pipes. The only way to use Win32 named pipes to communicate between Interix and Win32 processes is through anonymous pipes, as desribed in the previous section.

There is a difference in the behavior of pipes in UNIX and Windows. UNIX uses the buffer concept to deal with programs that use pipes, whereas Windows uses the file system object. If the program has a tendency to acquire "back pressure," this difference becomes a major issue to contend with when migrating to Windows using Interix. This issue arises in sequences of programs connected by pipes when there is a finite capacity for the pipe in UNIX versus a Windows file system object that will almost never be exhausted (based on page-file size).

However, Interix exhibits the same behavior as UNIX because it adds a UNIX style buffer to the front end of the Windows pipe, which is a file system object. So change to the application is necessary, even if it relies on the presence of back pressure.

## Shared Memory

Shared memory permits two or more processes to share a region of memory. Shared memory performance is considered the best of all interprocess communication (IPC) methods because 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.

Interix supports all of the System V IPC mechanisms. The **shmat**, **shmctl**, **shmdt,** and **shmget** **mmap()** routines can also be used to share memory between Interix and Win32 processes.

The command-line interfaces— **ipcs** and **ipcrm—**are also provided to manage shared memory segments. The **ipcs** interface reports the status of IPC objects. The **ipcrm** interface removes an IPC identifier, such as a shared memory segment.

Windows does not support the standard System V IPC 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.

```
/* 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
	Producer).
	*/
	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 Producer 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);
}

/* Producer */
/* This program is a producer 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);
}

/* 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];
};
```

## *System V Message Queues*

Message queues are very similar to named pipes, but there is no need to open or close message queues. Interix supports all the message queue routines—**msgctl**, **msgget**, **msgsnd**, and **msgrcv**. Code that uses these functions does not need to be modified.

**Note** Interix does not support Microsoft Message Queuing; however, similar functionality can be implemented using System V message queues.

# Networking

The Interix SDK implementation uses the Windows network stack, through the use of Windows Sockets 2 (Winsock) to access the network. This means that TCP/IP sockets and all of the installed Winsock protocols are supported.

**Note** Additional information about Winsock is available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/winsock/winsock/windows_sockets_start_page_2.asp.

This section covers the TCP/IP protocols and tools, and how remote procedure call (RPC) and sockets use TCP/IP.

# TCP/IP Protocols and Tools

TCP/IP is an industry-standard suite of protocols designed for large Internet works spanning wide area network (WAN) links.

Windows Services for UNIX 3.5 provides two remote shell servers: the Windows-based Remote Shell service and the Interix rshd(1). If the Windows Remote Shell service is installed, it is enabled by default and the Interix **rshd** daemon is disabled by default in /etc/inetd.conf. The Windows and Interix versions of these servers should not be run at the same time because they will both try to access the same TCP/IP port, causing unpredictable results. To avoid problems, you should ensure that only the Windows or the Interix version of these servers is enabled at a time.

## Remote Procedure Call

Open Network Computing (ONC) remote procedure call (RPC) is an IPC technique that allows client and server software to communicate with each other. These routines allow C programs to make procedure calls on other computers across the network. First, the client calls a procedure to send a data packet to the server. On receipt of the packet, the server calls a dispatch routine to perform the requested service and then sends back a reply. Finally, the procedure call returns to the client.

Interix provides **rpc**()–library routines for RPCs. The following must be present in the list of includes:

```
#include <rpc/rpc.h>
```

ONC RPC is different and is not compatible with Windows because this protocol uses Distributed Computing Environment (DCE) RPC for a number of system services. However, there are third-party software packages that implement ONC RPC on the Windows side. Tools such as Interix and NuTCRACKER provide support for ONC RPC. Additionally, shareware tools are available to support ONC RPC on Windows 2000.

### *Displaying Current TCP/IP Connections*

Interix provides **netstat**, which shows protocol statistics and current TCP/IP network connections. It displays active TCP connections, ports on which the computer is receiving, Ethernet statistics, the Internet Protocol (IP) routing table, and IP statistics for the IP, Internet Control Message Protocol (ICMP), TCP, and User Datagram Protocol (UDP) protocols. Used without parameters, **netstat** displays active TCP connections.

### *Host Name to Address Translation*

Interix does not support the generic transport name-to-address translation routines. It supports all the **gethostby** routines except the reentrant versions (routines with the _r suffix). While Interix does not ship with the *_r for the gethostby* family of APIs, you can freely install the BIND 9 library from the Interop /Tools site to gain the *_r functionality.

Table 5.3 lists other host name translation get or set routines that are not supported by Interix.

**Table 5.3. Host-Name Translation Routines Not Supported by Interix**

| Function Name | Description | Suggested Interix Replacement |
|---|---|---|
| **getdomainname** | Gets the NIS domain name. | No equivalent in Interix. The principal Windows domain for the system can be obtained through **getpdomain**(). The prototype of the method is as follows: int **getpdomain** (char * buf, int bufsize). |
| **gethostid** | Gets the unique identifier of the current host. | No equivalent in Interix, but should be a very rare occurrence in any application except a network administration application. |
| **setdomainname** | Sets the NIS domain name. | No equivalent in Interix. |
| **sethostid** | Sets the unique identifier of the current host. | No equivalent in Interix, but should never need to be set. It is restricted to the root user account. |
| **sethostname** | Sets the name of the host computer. (This call is restricted to the superuser and is normally used only when the system is booted.) | No equivalent in Interix. |

## Sockets

Interix implements BSD-style socket interfaces, including **bind**(), **accept**(), and **connect**(). This implementation uses the Windows network stack through the use of Winsock to access the network. This means that TCP/IP sockets and all of the installed Winsock protocols are supported.

The exceptions to socket support in Interix are discussed in the following sections:

- Network Groups
- Network Socket Calls
- Transport Level Interface (XTI) Calls
- Select Functionality

### *Network Groups*

Interix does not support network group APIs. However, it supports network group designations in hosts.equiv and .rhosts files.

### *Network Socket Calls*

These functions constitute the BSD sockets library. Interix provides all the network functions in the main C library, libc. For compatibility, Interix includes an empty libsocket.a library.

Table 5.4 lists the socket calls that are not supported by Interix.

**Table 5.4. Socket Calls Not Supported by Interix**

| Function Name | Description | Suggested Interix Replacement |
|---|---|---|
| **cmsg** macros | Access ancillary data. | No equivalent in Interix. |
| **freehostent** | Removes IP node entry from linked list. | Currently no API support for IPv6. |
| **getipnodebyaddr** | Gets IP node entry. | Currently no API support for IPv6. |
| **getipnodebyname** | Gets IP node entry. | Currently no API support for IPv6. |
| **inet_ntop** | Processes network address structures. | Currently no API support for IPv6. |
| **inet_pton** | Creates a network address structure. | Currently no API support for IPv6. |
| **rcmd_af** | Returns a stream to a remote command and includes support for Ipv6. | Currently no API support for IPv6. |
| **recvmsg** | Receives a message from the socket. | No equivalent in Interix. |
| **rexec_af** | Returns a stream to a remote command and includes support for Ipv6. | Currently no API support for IPv6. |
| **rresvport_af** | **Returns a descriptor to a socket with an address in the privileged port space.** | Currently no API support for IPv6. |
| **sendmsg** | Sends a message to a socket. | No equivalent in Interix. |

The **recvmsg** and **sendmsg** socket functions are used in many network applications but are not supported by Interix version 3.5 and earlier. These functions are the only way to pass an open file descriptor from one running process to another running process. These are mostly used for data exchange. On Windows Services for UNIX 3.5, **recvfrom**() and **sendto**() APIs can be used to exchange the data between the processes.

Each reentrant interface performs the same operation as its non–re-entrant counterpart does. The only difference is the _r suffix. The reentrant interfaces use buffers supplied by the caller to store returned results, and they are safe for use in both single-threaded and multithreaded applications. If the application is not multithreaded, then the _r routines can be safely replaced by removing the _r suffix and the additional parameters.

**Note** BIND 9 for *_r routines are available at http://www.interopsystems.com/tools.

Table 5.5 lists the reentrant routines and their Interix replacements.

**Table 5.5. Interix Replacements for Reentrant Routines**

| Function Name | Description | Suggested Interix Replacement |
|---|---|---|
| **getnetbyaddr_r** | Searches for a network entry with the network address. | struct netent * **getnetbyaddr** (long net, int type) |
| **getnetbyname_r** | Searches for a network entry with specified name. | struct netent * **getnetbyname** (char *name) |
| **getnetent_r** | Enumerates network entries from the database. | struct netent * **getnetent** (void) |
| **getprotobyname_r** | Sequentially searches from the beginning of the file until a matching protocol name is found, or EOF is encountered. | struct protoent * **getprotobyname**(const char *name) |
| **getprotobynumber_r** | Sequentially searches from the beginning of the file until a matching protocol number is found, or EOF is encountered. | struct protoent * **getprotobynumber** (int proto) |
| **getprotoent_r** | Gets a matching protocol name. | struct protoent * **getprotoent** (void) |
| **getservbyname_r** | Returns a pointer to an object containing the information from a network services database. | struct servent ***getservbyname** (char *name, const char *proto) |
| **getservbyport_r** | Returns a pointer to an object containing the information from a network services database. | struct servent ***getservbyport** (int port, const char *proto) |
| **getservent_r** | Returns a pointer to an object containing the information from a network services database. | struct servent ***getservent**(void) |

## *Transport Level Interface (XTI) Calls*

The X/Open Transport Interface (XTI) APIs, defined by the X/Open Transport Interface specification of Open Group, define protocol-independent networking functions similar to those provided by the old SVR4 TLI (Transport Layer Interface) APIs. Earlier, XTI and TLI were primarily used as interfaces to the ISO OSI protocol family or to the STREAMS networking stack. In general, XTI should be replaced by the more standard BSD sockets interface.

Interix support for XTI is limited to the functions and features required to access the UDP Internet protocol. Interix does not support some extended calls, which are mainly used with “expedited data” and the management or configuration of variables and parameters.

Table 5.6 lists the TLI calls that are not supported by Interix.

**Table 5.6. Transport-Level Interface Calls Not Supported by Interix**

| Function Name | Description | Suggested Interix Replacement |
|---|---|---|
| **nlsgetcall** | Gets client data passed through the listener. | No equivalent in Interix. |
| **nlsprovider** | Gets the name of the transport provider. | No equivalent in Interix. |
| **nlsrequest** | Formats and sends a listener service request message. | No equivalent in Interix. |
| **t_rcvv** | Receives data or expedited data sent over a connection and puts the data into one or more noncontiguous buffers greater than or equal to. | No equivalent in Interix. |
| **t_rcvvudata** | Receives a data unit in one or more noncontiguous buffers. | No equivalent in Interix. |
| **t_sndv** | Sends data or expedited data from one or more noncontiguous buffers on a connection. | No equivalent in Interix. |
| **t_sndvudata** | Sends a data unit from one or more noncontiguous buffers. | No equivalent in Interix. |
| **t_sysconf** | Gets configurable XTI variables. | No equivalent in Interix. |

## *Select Functionality*

On UNIX **select** is limited to 512 FDs and 64 FDs on Windows by default.

You can fix this by defining FD_SETSIZE in winsock2.h or in the user-defined header file after including the winsock2.h header file:

#define FD_SETSIZE 512

On UNIX, **select** doesn't block on empty lists of fds. If you don't pass any socket fds, it blocks for the specified timeout. On the Interix environment, it returns immediately with an error.

# Daemons and Services

A daemon in UNIX is a process that runs in the background to provide service to other applications and does not require a user interface. A service on Windows is the equivalent of a UNIX daemon. Normally, a daemon is started when the system is booted and runs without supervision until the system is shut down. Similarly, Windows services enable you to create long-running executable applications that run in their own Windows sessions. These services can be automatically started when the computer boots to continue across the logon sessions, can be paused and restarted, and do not show any user interface. Services are ideal for use on a server or for long-running functionality that does not interfere with other users who are working on the same computer. It is possible to run services in the security context of a specific user account that is different from the logged-on user or the default computer account.

This section provides an overview of the UNIX daemons and Windows services, explains their similarities and differences, and how to use them in the Interix environment.

## *Daemons*

A UNIX daemon is a process that provides a specific service. The following are some examples:

- The **inetd** daemon listens for connections on certain Internet sockets to start other daemons.
- The **nfsd** daemon implements the user-level part of the NFS (directory/file sharing) services.
- The **syslogd** daemon provides system tools with support for system and kernel logging.

On traditional UNIX systems, a daemon is a process that runs for an extended period of time, but it does not have a controlling terminal. A Windows service is a background process that is similar to a daemon process. You can run daemons directly on the Interix subsystem, or you can port daemons to run as Windows services.

Many daemons use **setuid**() or **seteuid**() to run as a particular user. However, this does not work on Windows because of the way in which Windows security is structured. A daemon invoked by **inetd** does not have these restrictions. It inherits the environment of the **inetd** process. In this context, the term daemon refers to a daemon process invoked by an Interix process that runs in the context of the Interix subsystem. An Interix service is an Interix process that is tied to the Win32 execution environment by the psxrun.exe program.

The chief advantage of running a daemon as a service is that the service runs as if it is logged on as a Windows user. This allows you to control the privileges and permissions granted to the service. When a daemon is logged on with a domain account, it enables the service to access the Windows network resources. However, unlike traditional UNIX daemons, Windows services cannot use mechanisms such as **fork** and **exit** to create a background process. This might make it difficult to port a daemon to run as a service.

## *Cron Service*

Windows Services for UNIX provides the **cron** daemon that runs under the Interix subsystem. The **cron** daemon is used to execute scheduled commands. The **cron** daemon searches its spool directory (/var/spool/cron/tabs/) for crontab files that are named after fully qualified user names in the form domain+user and loads the crontab files it finds into memory. The **cron** daemon also searches for /etc/crontab that is in a different format. The **cron** daemon then wakes up every minute, examining all loaded crontab files and checking each command to see if it should be run in the current minute. The **cron** daemon requires the user to register a valid password before editing or running any crontab entry. The **cron** daemon uses this password to impersonate this user to execute any crontab entries submitted by this user. For **cron** to successfully execute another user's **cron** jobs, the user must have registered his or her password using *crontab –p* command. Without this password, **cron** cannot impersonate that user and will not execute the user's crontab commands.

A **crontab** file contains instructions to the **cron** daemon of the general form: *run this command at this time on this date*. Each user who has been assigned **cron** privileges has his or her own **crontab** file.

The Windows-based **Cron** service allows users to run commands at scheduled times, much like the UNIX **cron** daemon. Users run the **crontab** Windows command to schedule jobs.

**To start and stop the Windows-based Cron service**

1. Go to Control Panel.
2. Click **Administrative Tools**, and then **Services**.
3. In the list of services, right-click **Windows Cron Service**.
4. Click **Properties**. If the Startup type is disabled, select either **Manual** or **Automatic**.
5. Click **Apply**.
6. Click either **Start** or **Stop**.

## Remote Shell Service

The Windows-based Remote Shell Service allows you to use a remote computer to execute commands on the computer running the Remote Shell Service, much like the UNIX **rsh** daemon.

The Windows Remote Shell Service (**rshsvc.exe**) is automatically installed on Windows Server-level products when you perform a standard installation of Windows Services for UNIX 3.5 and is automatically selected for a custom installation on these products.

**rshsvc.exe** resides in the *%SFUDIR%\common directory*.

**To use the Windows Remote Shell Service**

1. Install the service from the Windows Services for UNIX 3.5 installation medium onto the target computer.
2. Enable the service to start automatically or manually, as appropriate on your computer.
3. Create a .rhosts file in the %windir%\system32\drivers\etc directory of the target system. The file must contain a list of computers and users who will be allowed to connect to the target system. Each line in this file should have the format as "*hostname username*", same as UNIX host files. The members of the local users group must be able to read this file. However, this file is not writable. The typical inherited permissions in the default directory should be appropriate.
4. Log on to the target system as the user who will connect to the system.
5. Run **rshpswd** to store the password of the user.
6. Connect from the originating system by using a simple command such as **set** to test the connectivity. For example, to connect to a system called example-srv, the command might read:

```
rsh example-srv set
```

**Note** Additional information on common configuration problems of the Remote Shell Service and their solutions is available at

http://www.microsoft.com/technet/interopmigration/unix/sfu/sfu35rsh.mspx.

The remote shell capability is inherently an insecure protocol. Use the Secure Shell (**ssh**) instead for a similar capability with greater security; this is available for both Windows and for the Interix subsystem.

**Note** You can download OpenSSH for Interix at

http://www.interopsystems.com/tools/warehouse.htm.

# The Interix Remote Shell Daemon

The Interix Remote Shell Daemon, **rshd**, runs as a daemon initiated by the **inetd** process and behaves in a more traditional UNIX manner. By default, **rshd** is disabled.

The Interix **rshd** uses a more traditional authentication mechanism, with both hosts.equiv and .rhosts supported. The format of each line in the hosts.equiv file used by Interix **rshd** should be as “hostname username”. Interix **rshd** supports a user-specific .rhosts file that is present in the home directory of the user. This file is checked only if a hosts.equiv match is not found. This file follows the same format as the hosts.equiv file.

As in traditional UNIX systems, the .rhosts file must reside in the home directory of the user. It must be a regular file owned by the user, and it must be writable only by the user. In addition, you should use only the Interix **vi** tool or other UNIX editor to create and edit the file. Windows editors can insert Windows text format line endings or other stray characters that will cause a failure. However, you can use the **flip** tool that converts text-file formats between POSIX and other formats like Microsoft MS-DOS®.

The following tools are supported:

- **regpwd**
- **rcp**
- **rlogin and rlogind**

## regpwd

The **regpwd** tool stores a copy of the password in a protected area accessible only by privileged processes. This password can then be used by privileged programs that impersonate the user, such as **rshd**, as well as the Windows Remote Shell Service (**rshsvc**).

## rcp

The UNIX remote copy, **rcp**, is supported by the Interix **rshd**. This tool copies files between computers.

## rlogin and rlogind

Interix also supports remote logins using the **rlogin** client and **rlogind** daemon. Remote logins do not require that a password be stored on the target computer. However, passwords are then passed in clear text over the network. On a traditional UNIX system, if all the necessary conditions are met, remote logins through **rlogin** do not require a password if *.rhosts* or *hosts.equiv* allows the remote user. Under Interix, if the default permissions and ownerships have not been changed and the SYSTEM account runs the Interix subsystem, then the behavior is as expected, except that **regpwd** must be used to store the password just as with **rshd**. In both traditional UNIX and Interix systems, if the remote logon cannot occur automatically, the remote user is prompted for the password of the target user. If the password is correctly provided, the user is logged on.

Keep the following in mind for the Interix **rshd** daemon:

- The Interix **rshd** daemon does not provide a log of all failures and successes, as does Windows Remote Shell Service (**rshsvc**). However, when started with the **–L** option, it will write verbose success messages to the **syslog.** The **syslogd** is not enabled by default in Windows Services for UNIX 3.5, so you will need to enable it.
- You must stop the Windows Remote Shell Service before you enable **rshd**.
- DNS failures between the client and server systems may cause the computer names in .rhosts or hosts.equiv to not be resolved correctly. Hence, as a good practice, use IP addresses during initial testing and setup. After everything is working, revert to using DNS resolvable names to avoid later configuration issues if IP addresses change.

**Note** To enable the Interix Remote Shell Daemon, edit the file /etc/inetd.conf by using the Interix **vi** or other Interix or UNIX editor.

Additional information on common configuration problems of the Interix Remote Shell Daemon and their resolutions is available at

http://www.microsoft.com/technet/interopmigration/unix/sfu/sfu35rsh.mspx.

## Porting a Daemon to Interix

This section discusses porting a UNIX daemon to Interix and calling that daemon from either **inetd** or another master daemon. Daemons ported in this way cannot be run as a Windows service. If a daemon needs to run as a Windows service, use the instructions in the "Porting a Daemon to Interix Service" section.

When you port a daemon from UNIX to Interix, the daemon must have the following features:

- The **daemon** call is an interface to allow a program to become a system daemon. This function causes the calling program to **fork**. The parent exits, and the child performs a **setsid**. This disassociates the process from its current process group, session, and controlling terminal. On successful completion of this call, the process is the session leader of a group in which it is the only member, and the session has no controlling terminal.
- The signal-handling routine (terminate in this case) is a common characteristic of a daemon process.The daemon can perform cleanup operations when it receives the SIGTERM signal (for example, when the system shuts down) by using the signal handling routine. To use the terminate handling routine, the daemon must call the **sigaction**() system call, which installs its signal handler for the SIGTERM signal. Another signal, SIGHUP, is often used to signal to the daemon that it should reinitialize or restart.

Porting this daemon is a simple process. All you need to do is recompile, relink, and execute the daemon from the command line.

## Porting a Daemon to Interix Service

The previous section discussed porting a UNIX daemon to Interix. However, an Interix daemon has some limitations. First, it can start other daemons, but individual daemons can be started through the **init** scripts that are part of startup of Interix only. Secondly, it is not integrated into the Windows service mechanisms. When a daemon is converted into a service, it can be managed in Windows from Control Panel (by clickin**g Administrative Tools**, and the**n Services**), as well as from the Interix command line.

Before going into more details on how to convert a daemon into a service, it is important to mention the following two commands that can help tie Interix processes to the Win32 execution environment:

- The **posix.exe** program starts an Interix process with a controlling terminal.
- The **psxrun.exe** program starts an Interix process without a controlling terminal.

In the case of a Windows service, the process must run without a controlling terminal. Run the **psxrun** program for such a case.

**Note** More details on **posix** and **psxrun** can be obtained from the Help manual of Windows Services for UNIX 3.5.

Interix services can be administered using **Services** in Control Panel or with the service tool provided with Interix.

The service tool is used to install the service as a particular user and to stop the service. The user name and password must be provided when the service is installed. If no user name is provided, the user name defaults to LocalSystem; this logs on the user as an administrator without access to the network. When a request to stop a particular service comes either from **Services** in Control Panel or from the service tool, **psxrun** sends the signal SIGTERM to the service.

## *Converting Daemon Code into Interix Service Code*

To convert daemon code into Interix service code, the daemon code needs the following modifications:

- Ensure that the service exits when it receives the SIGTERM signal. It must catch the SIGTERM signal, clean up, and shut down. Ideally, the service should not spend more than a few seconds in cleaning up; otherwise, the service can loose communications with the Services Control Manager.
- Change the code so that it does not **fork,** and the parent exits.

  If the parent process exits, **psxrun** treats the program as having exited, and the Windows Service Control Manager reports that the service was never successfully started.
- Do not call **setsid**( ) to create a new session. This does not work because of Windows Security. Use *#ifdef* to skip this code.
- Do not access network drives through drive letters from the daemon. Network drives are typically mounted on drive letters when a user logs on and get unmounted when that user logs out. A service program cannot depend on a given network drive mounted on a given drive letter. If a service uses net.exe to mount a network drive, the drive letter it uses becomes unavailable to interactive users, which may cause Winlogon.exe to display error messages. If a service must access a network drive, reserve specific drive letters for exclusive use by the system. It is suggested that you use the /net file system.

# Chapter 6: Developing Phase: Migrating the User Interface

This chapter describes how to migrate from a UNIX user interface (UI) to a Microsoft® Windows® user interface. As a majority of UNIX graphical interfaces are built on X Windows and Motif, once you know about the support available on Windows for X Windows and the differences between the UNIX user interface and the corresponding Windows UI, you can follow the steps required to migrate the user interface of X Windows, Motif, and POSIX applications to the Windows UI.

## X Windows Support on Windows Services for UNIX 3.5

The X Windows system is a portable, network-transparent Windows system that runs on many different computers. Any X Windows system consists of two distinct parts:the X server and one or more X clients. The server controls the display directly and is responsible for all input and output (I/O) through the keyboard, mouse, or display. The clients do not access the screen directly. Instead, they communicate with the server, which handles all I/O requests. It is the clients that do the "real" computing, such as running applications. Each client communicates with the server, causing the server to open one or more windows to handle the I/O requests for that client.

Windows Services for UNIX 3.5 includes many X Windows-based client programs that you can start from within Microsoft Interix. These programs, which include **xterm**, **xlsfonts**, and **xrdb**, are compiled using X11R5 and X11R6 libraries to run with the corresponding servers. Some of these programs are provided for only one version of the X Windows system, but most programs compiled for use on X11R5 servers are capable of running with X11R6 servers as well.

However, Windows Services for UNIX 3.5 does not include an X Windows-based server. You must first install and start an X Windows server program before using any X Windows-based client applications. In addition, you must have an X11 server running either locally or on a remote system before you can run any of the Interix X Windows-based clients on your computer. You must also set the DISPLAY variable in Interix as the name of the computer on which you want the X client output to be displayed. This would be either *localhost:0.0* if you have a local X11 server running, or the remote-system name, where an X server is running. When a shell is started with Interix, the environment variable DISPLAY is set to *localhost:0.0* by default.

Windows Services for UNIX 3.5 also supports the **xhost** tool, which is used to add and delete host names to the list allowed to make connections to the X server. The usage syntax of **xhost** is *xhost [[+-]name…]*, where name is the host name to be added to the list allowed to make connection to the X server. The **xhost +** command grants access to everyone, while the **xhost –** command restricts access to only those on the list. More details on the **xhost** program can be found in the Help manual of Windows Services for UNIX 3.5.

The display of X client applications depends upon how you have configured your X11 server and which window manager you are running. Most X11 servers running under Windows have the following two display modes:

- **Single-window mode.** One application window, the X root window, is open. All X clients exist in this window. You must use a window manager, such as **twm**(1X11R6). The **twm** window manager is one of the X Windows clients included with Interix. It can be run from the command line.
- **Multiple-window mode.** Each X client application is opened in its own window. Individual windows are controlled by the Microsoft Windows window manager or an X Windows window manager.

Most X servers that run with Windows have a default X Windows window manager in the multiple-window mode. Hence, you do not need to necessarily start an X Windows manager to run a client except when you need to use the manager with settings that are different from the default.

By default, Interix also sets up your PATH to include the directory */usr/X11R6/bin*, which is the location of the X11R6 client binaries. The *XAPPLRESDIR* environment variable, which is used by X clients to locate their resources, is set up for you as well.

## X Windows Programs Supported by Interix

Table 6.1 lists the X Windows programs that are supported by Interix.

**Table 6.1. X Windows Programs in Interix**

| X Windows Program | Description |
|---|---|
| **Xman*** | This tool is a manual page browser. |
| **Xedit*** | This tool opens a simple text editor for X Windows. |
| **Xbiff*** | This program displays a little image of a mailbox. When there is no mail, the flag on the mailbox is down. When mail arrives, the flag goes up and the mailbox makes a sound. |
| **Xcalc*** | This tool is a scientific calculator desktop accessory that can emulate a TI-30 or an HP-10C. |
| **Xeyes*** | This tool is a follow-the-mouse X Windows demo that watches what you do and reports to the Boss. |
| **xterm** | This tool is a terminal emulator for X Windows. |
| **xclock** | This program displays the time in analog or digital form. The time is continuously updated at a frequency that may be specified by the user. |
| **oclock** | This program displays the current time on an analog display. |
| **twm** | This is a window manager for X Windows. It provides title bars, shaped windows, several forms of icon management, user-defined macro functions, click-to-type and pointer-driven keyboard focus, and user-specified key and pointer button bindings. |

**Note** All X Windows programs marked with * are supported by X Windows System 11 Release 5 (X11R5). Therefore, to run such clients as **xman**, you need to add /usr/X11R5/bin to your PATH or give /usr/X11R5/bin/xman as the command.

Additional X Windows programs, such as **rxvt,** are available for downloading at http://www.interopsystems.com/tools/warehouse.htm.

## *Building X Windows Applications*

The Interix Software Development Kit (SDK) contains X11R5 and X11R6 libraries, header files, and various other tools specifically designed for building X Windows applications.

Many of the tools provided as part of the Interix SDK (**cpp**, **imake**(1X11R5), **xmkmf**(1X11R5), **mkdirhier**(1X11R5), and **makedepend**(1X11R5)) are necessary to build X11 software packages provided by third parties.

When you are building an application, you must include the appropriate X11 header directory as part of the object compilation as follows:

```
CFLAGS = -O -I/usr/X11R5/include
```

or

```
CFLAGS = -O -I/usr/X11R6/include
```

You must also include the libraries and place them after the object files as follows:

```
XLIBS = -lXaw -lXmu -lXt -lXext -lX11
```

For the X11 versions of functions to be linked, include general purpose libraries after the X11 libraries:

```
XLIBS = -lXaw -lXmu -lXt -lXext -lX11
```

You must include the -L directive (library location) before the object file:

```
XLIBDIR = -L/usr/X11R5/lib
```

or

```
XLIBDIR = -L/usr/X11R6/lib
```

Some applications use the **xrdb**(1X11R6) tool to create the application default files. When you build these applications, you must set your display and start the X11 server before you start the build process.

Many X Windows applications come with Imakefiles. You can create a makefile from an Imakefile using either **xmkmf**(1X11R5) or **imake**.

If you use **xmkmf** to build an application from an Imakefile, you will see the following output:

```
mv Makefile Makefile.bak
imake -DUseInstalled -I/usr/X11R6/lib/X11/config
```

You can use **imake** directly, but you might need to pass appropriate -D and -I options. You can type the previous command as an **imake** command:

```
imake -DUseInstalled -I/usr/X11R5/lib/X11/config
```

or

```
imake -DUseInstalled -I/usr/X11R6/lib/X11/config
```

After building the makefile, you can build the application with **make**.

**Note** To install the shared X11 libraries, during installation, install the GNU SDK component.

# Migrating Character-based User Interfaces

Not all UNIX interfaces are graphical. Character-based interfaces were the original mainstay of UNIX computing long before the graphical workstation was developed. The following two options are available for character-based interfaces:

- Migration to the Interix environment can take place with minimal change.
- Replacement of the character-based interface with the graphical interface (Windows-based or HTML).

A preliminary port to Portable Operating System Interface (POSIX) using POSIX terminal I/O ensures a smooth migration to Interix.

# POSIX Terminal I/O

The POSIX termios structure and a new set of access calls replace the two traditional terminal hardware interfaces, namely termio structures in System V and stty structures in Berkley Software Development (BSD).

The POSIX input/output (I/O) model is very similar to the System V model. The following two modes exist in the POSIX I/O:

- **Canonical mode.** Canonical input is line-based, like BSD cooked mode.
- **Noncanonical mode.** Noncanonical input is character-based, like BSD raw or cbreak mode.

The Interix subsystem includes a true, noncanonical mode, with support for **cc_c[VMIN]** and **cc_c[VTIME]**. The termios structure is defined in Termios.h, as shown in the following listing:

```
struct termios {
tcflag_t c_iflag; /* input mode */
tcflag_t c_oflag; /* output mode */
tcflag_t c_cflag; /* control mode */
tcflag_t c_lflag; /* local mode */
speed_t c_ispeed; /* input speed */
speed_t c_ospeed; /* output speed */
cc_t c_cc[NCCS]; /* control characters */
};
```

The Interix SDK extends the POSIX.1 set of flags for c_iflag to include IMAXBEL and VBELTIME. For c_cc, VMIN and VTIME do not have the same values as VEOF and VEOL. However, for a portable application, a developer should take into consideration that VMIN and VTIME can be identical to VEOF and VEOL on a POSIX.1 system.

Table 6.2 lists the 12 new functions that replace the terminal I/O **ioctl**() calls, which include **ioctl**(fd, TIOCSETP, buf) and **ioctl**(fd, TIOCGETP, buf) or **stty**() and **gtty**(). They were changed because the data type of the final argument, for terminal I/O **ioctl**() calls, depends on an action that makes type checking impossible.

**Table 6.2. Functions That Replace the Terminal I/O ioctl() Calls**

| Function | Description |
|---|---|
| int **tcgetattr** (int fd, struct termios *t) | Fetches attributes (termios structure). |
| int **tcsetattr** (int fd, int action, const struct termios *t) | Sets attributes (termios structure). |
| speed_t **cfgetispeed** (const struct termios *t) | Gets input speed. |
| speed_t **cfgetospeed** (const struct termios *t) | Gets output speed. |
| int **cfsetispeed** (struct termios *t, speed_t speed) | Sets input speed. |
| int **cfsetospeed** (struct termios *t, speed_t speed) | Sets output speed. |
| int **tcdrain** (int fd) | Waits for all output to be transmitted. |
| int **tcflow** (int fd, int action) | Suspends transmit or receive. |
| int **tcflush** (int fd, int queue_selector) | Flushes pending I/O. |

| Function | Description |
|---|---|
| int **tcsendbreak** (int fd, int len) | Sends BREAK character. |
| pid_t **tcgetpgrp** (int fd) | Gets foreground process group identifier (ID). |
| int **tcsetpgrp** (int fd, pid_t pgrp_id) | Sets foreground process group ID. |

For the window size, use the TIOCGWINSZ command for **ioctl**() with the winsize structure, which are both supported.

## *Porting Curses and Terminal Routines to Interix*

The Interix SDK libraries include libcurses, a port of the ncurses package, and libtermcap, the termcap routines. The Interix SDK also supports pseudoterminals. Porting such applications to Interix should be straightforward, but note the following:

- The curses routines make use of the terminfo database, stored in /usr/share/terminfo. This location is different from the location used in traditional systems. To link with the terminfo routines, link with the curses library.
- Interix supports both the BSD */dev/ptynn* and the System V */dev/ptmx* methods for opening the master side of a pseudoterminal. The System V method is slightly faster because the search for an available master device is handled by the subsystem. Currently, the Interix subsystem supports 265 ptys named */dev/ pty[p–zA–E][0–9a–f]* on the master side. The corresponding subordinate side names are */dev/tty[p–zA–E][0–9a–f].*
- When using /dev/ptmx, the subordinate (slave) **tty** name can be obtained with **ptsname**(). BSD-based **ioctl**() calls can be used with the pty master side.
- Provided that it is a session leader, a process without a controlling **tty** acquires a controlling terminal on the first **open**() call to a **tty**, unless NOCTTY is specified in the **open**() call.

Older, character-based terminal applications placed the cursor on the physical display screen based on the capabilities of the terminal. These capabilities were typically stored in the /etc/termcap file.

**Note** The mouse-interfacing features found in the ncurses source are not enabled for Interix. Interix does not support System V STREAMS.

# Porting OpenGL, Motif, and Xview Applications

The Interop Systems Web site provides a product with Motif 2.2.2 development environment for the Interix technology in Windows Services for UNIX 3.5, with static libraries (and documentation). The OpenGL Development Kit version 1.3 with libraries and APIs for Windows Services for UNIX 3.5 is also included in the Interop Systems Web site.

**Note** Additional information is available at http://www.interopsystems.com/Motif-OpenGL.htm.

The Xview toolkit is a toolkit for developing X11 programs and includes an X11 window manager for OpenLook behavior.

**Note** Additional information on downloading the Xview toolkit is available at http://www.interopsystems.com/tools/warehouse.htm.

# Chapter 7: Developing Phase: Functions to Change for Interix

This chapter describes the important functions not supported in Microsoft® Interix that need to be changed or removed before the code compiles under Interix. This chapter also describes the availability of these functions in the Interix environment and their suitable replacement mechanisms. The functions discussed in the chapter include:

- Math routines.
- Regular expressions.
- System and C library and other platform-specific application programming interfaces (APIs).

The following APIs are not supported by Interix and should not be used:

- Wide character-type APIs
- Multibyte character-type APIs (ISO/ANSI C and UNIX 98)
- Long-long character type (64-bit integer) APIs
- Message-handling APIs

Using the information in this chapter, you will be able to identify which modifications are required in your application code to ensure compatibility in the Interix environment. You will also be able to use suitable replacement mechanisms in such areas of your application as command line and shell APIs, string-related functions, and system and C library APIs.

**Note** Extended UNIX Code (EUC) characters are not supported by Interix and should not be used.

## Math Routines

There are two sets of mathematical routines that are not supported by Interix. The first is the IEEE floating-point environment control routines, listed as follows:

- **fpclass**
- **fpgetmask**
- **fpgetround**
- **fpgetsticky**
- **fpsetmask**
- **fpsetround**
- **fpsetsticky**

The second set of mathematical routines is the conversion routines, listed as follows:

- **decimal_to_double**
- **decimal_to_extended**
- **decimal_to_floating**
- **decimal_to_quadruple**
- **decimal_to_single**
- **double_to_decimal**
- **econvert**
- **extended_to_decimal**
- **fconvert**

- **file_to_decimal**
- **floating_to_decimal**
- **func_to_decimal**
- **gconvert**
- **qeconvert**
- **qfconvert**
- **qgconvert**
- **seconvert**
- **quadruple_to_decimal**
- **sfconvert**
- **sgconvert**
- **single_to_decimal**
- **string_to_decimal**

GMP is a portable library written in C for arbitrary precision arithmetic on integers, rational numbers, and floating-point numbers. You can use this library as the replacement mechanism for the previously described math routines. It aims to provide the fastest possible arithmetic for all applications that need higher precision than is directly supported by the basic C types.

**Note** You can download the GMP Library at http://interopsystems.com/InteropToolworks.htm.

# Regular Expressions

Interix does not support some of the regular expression function calls. However, Table 7.1 lists functions that can be used to replace them.

**Table 7.1. Regular Expression Function Calls Not Supported by Interix**

| Function Name | Description | Suggested Interix Replacement |
|---|---|---|
| **re_comp** | Compiles and executes a regular expression and returns a character pointer to NULL on success. | int **regcomp** (regex_t *preg, const char *pattern, int cflags);<br>int **regexec** (const regex_t *preg, const char *string, size_t nmatch, regmatch_t pmatch[], int eflags); |
| **re_exec** | Compiles and executes a regular expression and returns an integer of 0 or 1 on success. | int **regcomp** (regex_t *preg, const char *pattern, int cflags);<br>int **regexec** (const regex_t *preg, const char *string, size_t nmatch, regmatch_t pmatch[], int eflags); |
| **Regcmp** | Compiles a regular expression and returns a pointer to a compiled form. | int **regcomp** (regex_t *preg, const char *pattern, int cflags); |
| **regex** | Executes a regular expression. | int **regexec** (const regex_t *preg, const char *string, size_t nmatch, regmatch_t pmatch[], int eflags); |

# System/C Library and Miscellaneous APIs

There are many specialized and platform-specific APIs under this category. They are as follows:

- Command-line and shell APIs
- String-manipulation functions
- BSD string and bit functions
- Time-handling APIs
- Other system/C library functions
- Kernel calls

The kernel calls are not supported by Interix and should not be used.

## *Command-Line and Shell APIs*

Interix does not support calls to obtain information about legal user shells from the /etc/shells file, nor does it support the implementation of the **popt** command-line parser.

**To replace the UNIX functions with the Interix functions**

1. Add to the /etc/shells file any additional legal shells that have been ported, such as /bin/bash. The /etc/shells file contains the legal Interix shells /bin/csh, /bin/ksh, /bin/sh, /bin/tcsh.
2. Write functions named **endusershell**, **getusershell**, and **setusershell**, as listed in Table 7.2.

**Table 7.2. Functions to Implement the Command-Line and Shell APIs in Interix**

| Function name | Description | Suggested Interix Replacement |
|---|---|---|
| **endusershell** | Closes the file of legal user shells (/etc/shells). | Create **endusershell** routine to wrap the standard **close**() file routine. The prototype of the routine is as follows:<br>int **close** (int d) |
| **getusershell** | Gets legal user shells from /etc/shells. | Create **getusershell** routine to wrap the standard **read**() file routine.<br>The prototype of the routine is as follows:<br>ssize_t **read** (int d, void *buf, size_t nbytes) |
| **popt** | Parses command-line options. | Use **getopt** function**.**<br>The prototype of the routine is as follows:<br>int **getopt** (int argc, char * const *argv, const char *optstring) |
| **setusershell** | Rewinds the file of legal user shells (/etc/shells). | Create **setusershell** routine to wrap the standard **lseek**() file routine to set /etc/shells file back to beginning. The prototype of the routine is as follows:<br>off_t **lseek** (int fd, off_t offset, int whence) |

## String-Manipulation Functions

Interix supports most of the standard string-handling functions. Interix does not support the **atoq**, **memmem**, **stpcpy**, **stpncpy**, **strfmon**, **strfry**, **strnlen**, and **strtows** functions. Use the replacements listed in Table 7.3 for these functions.

**Table 7.3. String-Handling Functions Not Supported by Interix**

| Function name | Description | Suggested Interix Replacement |
|---|---|---|
| **atoq** | Converts string to a long long. | **strtoq()** |
| **memmem** | Finds the start of the first occurrence of a substring in the memory area. | char * **strstr** (const char *big, const char *little) |
| **stpcpy** | Copies the source string, including the terminating \0 character, to the destination. | char * **strcpy** (char *dst, const char *src) |
| **stpncpy** | Copies at most *num* characters from the source string, including the terminating \0 character, to the destination. | char * **strncpy** (char *dst, const char *src, size_t len) |
| **strfmon** | Formats specified amount according to the format specification and places the result in a character array of size *max*. | int **sprintf** (char *str, const char *format ...) |
| **strfry** | Randomizes the contents of the string by using **rand**() to randomly swap characters in the string. | No equivalent in Interix. Customized function can be written using **rand**(). |
| **strnlen** | Returns the number of characters in the string, not including the terminating \0 character, but at most *maxien*. | size_t **strlen** (const char *s) |

## BSD String and Bit Functions

String interfaces specified by ANSI/ISO C and found only in the Single UNIX Specification are in string.h. The contents of the strings.h file are listed in Table 7.4.

**Table 7.4. String Interfaces in the strings.h File**

| Function name | Description |
|---|---|
| **bcmp** | Compares two strings. |
| **bcopy** | Copies at most *n* characters from **src** string to **dest**. |
| **bzero** | Places **len** value of 0 bytes in the string. |
| **ffs** | Finds the first bit set (beginning with the least significant bit) and returns the index of that bit. |

All the functions mentioned in the preceding table are supported in the Interix environment.

The Interix SDK (software development kit) also supports the BSD 4.4 **strsep()** and **strcasestr()** routines.

## Time-Handling APIs

The time functions that are not supported by Interix and that cannot be implemented by some other Interix API or set of API calls are **adjtime**, **adjtimex**, **ntp_adjtime**, and **tzsetwall**.

**Note** You can download the **rdate** tool at http://www.interopsystems.com/tools/warehouse.htm.

The **rdate** function sets the date of the system from a remote Network Time Protocol (NTP) server or host.

## Other System/C Library Functions

Interix does not support some of the system/C library functions. Table 7.5 lists these functions and their suggested replacements.

**Table 7.5. System/C Library Functions Not Supported by Interix**

| Function Name | Description | Suggested Interix Replacement |
|---|---|---|
| **on_exit** | Registers a function to be called at usual program termination. | int **atexit**(void (*function)(void)) |
| **quotactl** | Manipulates disk quotas. | No support or equivalent in Interix. |
| **stime** | Sets system time and date. | int **settimeofday** (struct timeval *tp, void *tzp) |

# Chapter 8: Developing Phase: Deployment Considerations and Testing Activities

This chapter discusses the key deployment considerations that need to be made in deploying the Microsoft® Windows® Services for UNIX 3.5 migrated application before closing the Developing Phase. This chapter also discusses various testing activities that you need to carry out in the Developing Phase. This chapter will help you identify the activities and milestones required to complete the Developing Phase.

## Deployment Considerations

The following are the key deployment considerations that need to be made during the Developing Phase to ensure smooth deployment in the Deploying Phase:

- Process environment
- Migration of scripts
- Database connectivity
- Deploying the application
- Interoperability with Windows Services for UNIX 3.5
- Monitoring and supporting the applications

The process for deploying the migrated application is discussed in detail in Volume 5, *Deploy-Operate* of this guide.

### *Process Environment*

Some of the key elements in the process environment that are different for UNIX and Interix are:

- Environment variables.
- Temporary files.
- Computer information.
- Logging system messages.

This section discusses each of these key elements and explains how to implement them in Interix.

#### Environment Variables

An environment block is a block of memory allocated within the process address space. Each block contains a set of name-value pairs. All UNIX variants support process environment blocks. The particular differences between Interix and other UNIX variants depend on the UNIX variant being ported to Interix. For example, some UNIX variants do not support either the **setenv** or the **unsetenv** function calls, whereas Interix does.

There are usually no issues in porting calls to environment variable functions in Interix. However, when porting System V Interface Definition (SVID) code, instead of the process environment being defined as a third argument to **main**(), it is defined as extern char **environ. To modify the environment for the current process, use the **getenv**() and **putenv**() functions. To modify the environment so that it passes to a child process, use the **getenv**(), **setenv**(), and **putenv**() functions, or build a new environment and pass it to the child using the envp argument of the **exec**() function.

**Note** The **putenv** and **setenv** functions are only available if *_ALL_SOURCE* is defined and set to 1.

## Temporary Files

Interix supports all standard and common functions that create temporary files. You do not need to modify the code to migrate these functions to Interix.

## Computer Information

Interix supports functions that obtain information about the computer on which the application is executed. Typically, the ported code does not require any modifications.

The computer information includes:

- Host name
- Operating system name
- Network name of the computer
- Release level of the operating system
- Version number of the operating system
- Hardware platform name

This information can be obtained by using the **uname -a** Interix shell command. Note that the **uname** command and application programming interface (API) return information about the installed version of Interix, and not the version of the host Windows operating system. You can get information about the Windows operating system version by adding the Interix-specific **-H** option to the **uname -a** command.

For example:

```
$ uname -a
Interix JOE-GX 3.0 SP-7.0.1701.1 x86
Intel_x86_Family15_Model1_Stepping2
$ uname -aH
Windows JOE-GX 5.1 SP0 x86 Intel_x86_Family15_Model1_Stepping2
```

## Logging System Messages

Interix provides the standard UNIX **syslogd** daemon to store and redirect log messages from applications and system services. The configuration file for **syslog** is located in */etc/syslog.conf*. The Interix **syslogd** daemon handles only those Interix processes that are designed to use the **syslog** API. It does not handle log messages from the Win32® subsystem. If **syslogd** is not running, all the messages intended for **syslogd** are appended to the file */var/adm/log/logger*.

The **syslog**, **vsyslog**, **openlog**, **closelog**, and **setlogmask** function calls are supported by Interix with the same set of severity levels, including:

- LOG_ALERT
- LOG_CRIT
- LOG_DEBUG
- LOG_EMERG
- LOG_ERR
- LOG_INFO

- LOG_NOTICE
- LOG_WARNING

The following superset of facility indicators is also supported:

- LOG_AUTH
- LOG_CRON
- LOG_DAEMON
- LOG_KERN
- LOG_LOCAL(0-7)
- LOG_LPR
- LOG_MAIL
- LOG_NEWS
- LOG_USER
- LOG_UUCP

It is not necessary to modify a code that uses **syslog** calls on Interix.

**Note** The **syslog** daemon is not started with the default installation. You can refer to the startup instructions on the **syslogd** manual page.

# *Migration of Scripts*

This section describes the process of porting UNIX shell scripts to the Interix environment. The porting process follows these steps:

1. Evaluating script migration tasks.
2. Planning for management of platform differences.
3. Evaluating source and target environments.

For information about each of these steps, refer to Volume 1: *Plan* of the *UNIX Custom Application Migration Guide*.

Scripts fall into the following two basic categories:

- Shell scripts, such as Korn and C shell.
- Scripting language scripts, such as Perl, Tcl, and Python.

Shell and scripting language scripts tend to be more portable than compiled languages such as C and C++. A scripting language such as Perl handles most of the platform specifics. However, the original developer may have used easier or faster platform-specific features, or may not have taken cross-platform compatibility into consideration at all.

A large number of UNIX commands are available with an Interix installation. Many UNIX shell scripts run under Interix without conversion because Interix provides both the Korn and Tenex C shells.

**Note** More information on porting shell scripts is available at http://www.microsoft.com/windows2000/docs/portingshellscripts.doc.

## Porting UNIX Shell Scripts to Interix

There are only two significant differences in porting a shell script from an open-system implementation of UNIX (such as System V4 or BSD) to Interix. First, by default, Interix stores binaries in one of the following three directories:

- /bin
- /usr/contrib
- /usr/local/bin.

For example, Perl is installed in one of these directories. Second, although Interix has a standard UNIX file hierarchy and a single-rooted file system with the forward slash (/) as the base of the installation regardless of the Windows drive or directory, absolute paths can be different. Absolute paths normally do not need to be converted because you can handle most situations by adding symbolic links. For example, /usr/ucb can be linked to /usr/contrib/bin, and /usr/local/bin can be linked to /usr/contrib/bin.

Additional considerations include:

- Port scripts that set up either local or environment variables.
- The Interix C shell initialization process executes the /etc/csh.cshrc and /etc/csh.login files before the .cshrc and .login files in the home directory of the user.
- Be aware of the current limits of Interix shell parameters so that you can take the appropriate action. These parameters and their current limits are:
  - Maximum length of $path ($*PATH*) variable = ARG_MAX (normally not a problem).
  - Maximum (shell) command length = ARG_MAX (normally not a problem).
  - Maximum (shell) environment size = ARG_MAX.
  - Maximum length of command arguments, that is, length of arguments for **exec()** in bytes, including environ data (ARG_MAX) = 1048576.
  - Maximum length of file path (*PATH*_MAX) = 512.
  - Maximum length of file name (NAME_MAX) = 255 (normally not a problem).
- Modify any scripts that rely on information from /etc/passwd or /etc/group (for example, a script that uses **grep** to find a user name) to use other techniques, such as Win32 ADSI scripts, to obtain information about a user. Examples include:
  - Calls to Interix **getpwent**(), **setpwent**(), **getgrent**(), and **setgrent**() APIs.
  - Win32 ADSI scripts.
  - Win32 **net user** commands.

# *Database Connectivity*

This section contains information about Open Database Connectivity (ODBC) and accessing databases from Windows Services for UNIX 3.5.

## Open Database Connectivity

Applications use the ODBC interface to access data from the database. This allows applications to access the database management systems (DBMS) using Structured Query Language (SQL) as a standard. ODBC permits applications to access different databases and therefore permits interoperability. Application end users can then add ODBC database drivers to link the application to their choice of DBMS.

ODBC database drivers are dynamic-link libraries on Windows and shared objects on UNIX. These drivers allow an application to access one or more data sources. ODBC provides a standard interface to allow application developers and vendors of database drivers to exchange data between applications and data sources.

The ODBC architecture has the following four components:

- **Application.** Processes and calls ODBC functions to submit SQL statements and retrieve results.
- **Driver manager.** Loads drivers for the application.
- **Driver.** Processes ODBC function calls, submits SQL requests to a specific data source, and returns the results to the application.
- **Data source.** Consists of the data, its associated operating system, DBMS, and the network platform (if any) used to access the DBMS.

## Accessing Databases from Windows Services for UNIX 3.5

UNIX applications are usually given database connectivity using ODBC drivers. Windows Services for UNIX 3.5 has no built-in libraries or interfaces for accessing relational databases stored on other platforms. However, there are several packages listed in Table 8.1 that can reduce this problem.

**Table 8.1. Packages Available for Database Connectivity**

| Name | Description |
|---|---|
| **iODBC** | A popular open source ODBC driver manager. |
| **unixODBC** | A popular open source ODBC driver manager.<br>This also serves as the ODBC-ODBC bridge client from Easysoft, allowing access to any ODBC driver in Windows. |
| **FreeTDS** | An open source implementation of the Tabular Data Stream (TDS) protocol used to access Microsoft SQL Server™ and Sybase databases, including dblib, ctlib, and an ODBC driver. |
| **Perl** | One of the most popular scripting languages, complete with the DBI database interface module, DBD::ODBC, for connecting through ODBC, and DBD::Sybase for connecting to any Sybase or SQL Server using FreeTDS. |

All of these applications and libraries are built with support for threads enabled by default. Some of these libraries have been ported to Interix and are available at

http://www.interopsystems.com/tools/warehouse.htm.

### *iODBC*

Independent Open DataBase Connectivity (iODBC) is an Open Source platform-independent implementation of both the ODBC and X/Open specifications. iODBC possesses the capability to develop applications independent of a back-end database engine, operating system, and (for the most part) programming language. Although ODBC and iODBC are both “C –based” APIs, there are numerous cross-language hooks and bridges from such languages as C++, Java, Perl, Python, and TCL.

**Note** Additional information is available at http://www.iodbc.org/.

### *unixODBC*

The unixODBC Driver Manager is used in the binary builds of all the Windows Services for UNIX 3.5 database tools. It is also the client for the ODBC-ODBC bridge of Easysoft. With the ODBC-ODBC bridge, you can access any database that has a Windows ODBC driver. One license of ODBC-ODBC bridge of Easysoft, which incorporates the unixODBC driver manager, gives Windows Services for UNIX or Interix users a universal ODBC solution.

For example, consider a situation in which your Windows Services for UNIX or Interix application needs access to a remote Oracle database on a UNIX-based or Linux-based computer. Oracle does not provide OCI support for Interix. However, there are Oracle drivers available for Windows, so any Windows-based computer can act as both the client and the gateway. The ODBC-ODBC Bridge Server (a commercial product) of Easysoft is installed on the gateway, and the ODBC-ODBC client (unixODBC) is installed on the computer of the end user. In fact, the client and the gateway can be the same physical server. To access another database (for example, Microsoft SQL Server™, Access, or Excel®), just configure the appropriate data source on the gateway. No action is required on the UNIX system.

Figure 8.1 illustrates the ODBC-ODBC bridge.

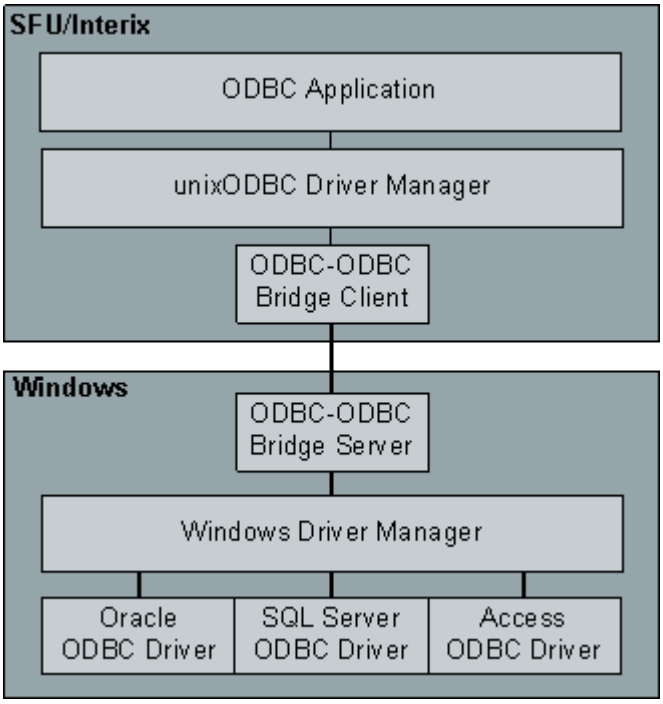


**Figure 8.1. ODBC-ODBC bridge**

For a detailed format of the odbc.ini file, refer to http://www.interopsystems.com/tools/db.htm#odbc_config.

Additional information about the ODBC-ODBC bridge is available at http://www.unixodbc.org/ and http://www.easysoft.com.

### *FreeTDS*

The TDS protocol is used to communicate with either Sybase or SQL Server databases. FreeTDS is an open source implementation of this protocol and contains several APIs that use it. The /usr/local/etc/freetds.conf configuration file must contain entries for your databases. Use these entries as templates for each server type. FreeTDS supports a number of different ways of setting up the configuration file; the method known as the "DBC-Combined" is strongly recommended.

**Note** Additional information about the DBC-Combined method is available at http://www.freetds.org/userguide/odbcombo.htm.

The configuration file can be overridden with the FREETDSCONF environment variable.

**Note** Additional information about the FREETDSCONF environment variable is available at http://www.freetds.org.

### *Perl*

Perl is especially complex to build correctly under Windows Services for UNIX 3.5. There are several pitfalls that can deter an unwary or inexperienced developer.

The best approach for building Perl under Windows Services for UNIX is to use the edited version of the config.sh and Policy.sh supplied files, and then running the following to rebuild the makefiles.

Run **gmake** to build a new binary:

```
$ ./Configure –der
```

Almost all the tests run by **gmake** test run correctly. However, there are a number of special cases for Windows Services for UNIX (for example, when the privileged user does not have UID 0 in Windows Services for UNIX) when the **gmake** test may hang after all the tests have completed successfully.

**Note** You can download Perl at http://www.interopsystems.com/tools/warehouse.htm.

# *Deploying the Application*

This section explains the process of deploying your migrated application, including Interix applications, into a Windows environment.

The standard method of deploying applications in a Win32 environment is to use the Microsoft Windows Installer service. This section provides information about the Windows Installer service and some of the tools that you can use to package your application.

When you deploy applications in the Interix environment, you can use the standard UNIX tools available in Interix for file transfer, remote configuration, and scripting. If you used the proprietary UNIX software management tools to manage the application before migration, it is unlikely that these tools will be available in Interix. If you use standard UNIX tools, such as the **tar** or **cpio** archiving commands and shell scripts for application management, migration of your deployment tools is relatively easy.

Interix applications reside either on the desktop or on the application servers. In the latter case, you may have to rely on the networked file systems to access the executable files of your application. Because of the differences in how the UNIX and Windows network file systems operate, you must take specific action on some migrated applications.

## Tools for Deploying Interix Applications

When migrating applications from UNIX to Interix, one of the main benefits is the similarity of the two platforms. This makes the process of migrating an application to Interix relatively easy. The similarity between the two platforms also extends to the tools available. If you have created your own deployment scripts using these tools, migrating the deployment tools to Interix is a straightforward process.

However, if you have used deployment tools specific to your UNIX distribution, you must either port these to Interix or write scripts using the available tools.

### *Berkeley Remote Shell Commands (r Commands)*

Distributing your application into the Interix environment is likely to call for the transfer of files over the network. You can use the **rsh**, **rcp**, **ssh, scp,** or **ftp** commands for the transfer of files. These commands are available in Interix.

**Note** The **ssh** and **scp** commands are not included in Interix. You can download these commands at http://www.interopsystems.com/tools/warehouse.htm.

### *Scripts*

Interix provides a wide range of scripting languages and tools for the creation of deployment tools. When you install Windows Services for UNIX 3.5, the UNIX Perl tool is installed as a part of the standard installation process. This allows Perl scripts to run on the Interix subsystem.

**Note** You can download Perl 5.8.3 at http://www.interopsystems.com/tools/warehouse.htm.

You can install ActiveState ActivePerl 5.6 in a custom installation process. This allows you to run Windows-based Perl scripts on the server.

Other languages, such as Ruby, Python, or Tcl/Tk, can also be used.

**Note** You can download these languages at http://www.interopsystems.com/tools/warehouse.htm.

Interix also includes C and Korn shell scripts.

**Note** You can download the bash and zsh shell scripts at http://www.interopsystems.com/tools/warehouse.htm.

# Deploying Interix Applications

You can use the approach described in the next section to deploy your applications on Interix.

## *Deploying Interix Applications by Pushing Them to the Desktop*

The push application delivery mechanism in UNIX environments is implemented with scripted **ftp** or **rcp** commands that copy the application binaries to a computer running UNIX. The application is then activated by pointing to a symbolic link at the new application binary. It is also common to use **rdist** and **rsync** to deploy the application across multiple computers. The **rdist** tool is supported by Windows Services for UNIX 3.5.

**Note** The **rsync** tool can be downloaded at http://www.interopsystems.com/tools/warehouse.htm.

From a management standpoint, remotely managed computers contain a selection of management scripts (Perl, C, or Korn shell) that can be invoked remotely to initiate an application image deployment, enumerate performance metrics, or run an audit for security.

## *Using the r Commands for Remote Management in Interix*

You can remotely manage Interix systems with the Berkeley **r** commands such as **rsh** and **rcp**. Before you use these commands for remote administration, configure the Interix daemon **rshd** to permit remote access between computers. This is because **rcp** uses a remote shell at the remote computer (source or destination of the copy) to start a shell to launch the **rcp** process with the appropriate arguments.

For example, when **rcp** is used to manage the system when Interix is the source of the directory copy, use the following:

```
rcp -r <Interix hostname>:ProgDir <destination hostname>:ProgDir
```

Interix has the following two processes associated with the **rcp** command in this case:

```
user1 9281 1025 20:41:48 - 0:00.03 sh -c rcp -r -f ProgDir
user1 9345 9281 20:41:48 - 0:00.26 rcp -r -f ProgDir
```

When **rcp** is used to manage the system when Interix is the destination (sink) of the directory copy, use the following:

```
rcp -r <source hostname>:ProgDir <Interix hostname>:ProgDir
```

Interix has the following two processes associated with the **rcp** command in this case:

```
user1 9729 1025 20:41:48 - 0:00.03 sh -c rcp -r -t ProgDir
user1 9793 9729 20:41:48 - 0:00.26 rcp -r -t ProgDir
```

Notice that a different argument is passed to **rcp** depending on whether it is the source (**-f**) or the destination (**-t**) of the copy.

**To automate the configuring of the host equivalent files**

1. Create the necessary configuration files and set appropriate permissions, as follows:

   /etc/hosts.equiv.

   For earlier versions of Interix 3.0, you can use the following command:

   `$HOME/.password`

   From Interix 3.0 onward, the **regpwd** tool is provided to store the password securely into the registry and is accessible only to the privileged processes.

   `$HOME/.rhosts`

2. Set appropriate permissions, as follows:

```
#!/bin/csh
#
# Does the hosts.equiv file exist?
# If not, create an empty one.
#
if ( ! -f "/etc/hosts.equiv" ) then
  #
  # create /etc/hosts.equiv
  #
  touch /etc/hosts.equiv

  #
  # Set the permissions on /etc/hosts.equiv
  #
  chmod 755 /etc/hosts.equiv

  #
  # Set the owner of /etc/hosts.equiv
  #
  chown +Administrators /etc/hosts.equiv
endif

if ( $?HOME) then

  #
  # See whether the users .rhosts file exists.
  # If not, then create it.
  #
  if ( ! -f "$HOME/.rhosts" ) then
    touch $HOME/.rhosts
    chmod 600 $HOME/.rhosts
  endif
```

```
endif
```

**Note** Use of the r commands may be restricted for security reasons. In this case, you need to modify the preceding script to reflect your security policies.

SSH is available for Interix as a secure alternative and can be downloaded at http://www.interopsystems.com/tools/warehouse.htm.

### *Installing MSI Packages Remotely with Interix rsh*

To remotely launch Win32-based applications from an Interix shell, remember that there are no stdin/stdout/stderr file handles on which to read and write. This is because the remote process is connected to pseudoterminals that do not have any corresponding Win32 object. Therefore, Win32-based applications must be wrapped so that they are provided with stdin/stdout/stderr file handles. The following code shows the remote execution of the **ipconfig.exe** command:

```
$ rsh remotesystem "/dev/fs/C/WINNT/system32/ipconfig.exe < /dev/null
2>&1 | cat"
```

You can also execute the Win32 command shell (**cmd.exe**) commands. The following depicts the execution of a remote **dir** command:

```
$ rsh remotesystem "/dev/fs/C/WINNT/system32/CMD.EXE /c dir < /dev/null
2>&1 |cat"
```

When you pass Win32 paths with Interix shells, backward slashes (\) used in the path specification must be escaped with another backslash (\). For example, to execute the MSI package installer from an Interix Korn shell, double backslashes (\\) must be passed in the paths to msiexec.exe. The following example shows this for the installation of the Windows® 2000 support tools (for example, *<drive>:\SUPPORT\TOOLS\2000 RKST.MSI*):

$ /dev/fs/C/WINNT/system32/msiexec.exe /I <drive>:\\SUPPORT\\TOOLS\\2000RKST.MSI /Lv C:\\Temp\\msi.log /qn.

The remote installation of this MSI package is interesting because **rsh** requires that Win32 path backslashes be escaped, and the command path that is passed over the remote shell must also have its backslashes (\) escaped. The following **rsh** command will install the Windows 2000 Support Tools remotely with an **rsh** command:

```
$ rsh remotesystem "/dev/fs/C/WINNT/system32/msiexec.exe /I
<drive>:\\\\SUPPORT\\\\TOOLS\\\\2000RKST.MSI /Lv C:\\\\Temp\\\\msi.log
/qn < /dev/null 2>&1 | cat"
```

(where *<drive>* is a drive letter available on the remote system.)

## Code Modification

To add or isolate code that implements Interix-specific features, surround it with the following code:

```
#ifdef __INTERIX, such as in the following example:
#ifdef __INTERIX
(void) fcntl(fd, F_SETFL, fcntl(fd, F_GETFD) | FD_CLOEXEC));
#else /* __INTERIX */
(void) ioctl(fd, FIOCLEX, NULL);
#endif /* __INTERIX */
```

The __INTERIX macro is automatically defined to be true by the **c89** compiler interface and by the **gcc** compiler. The _POSIX_ macro is defined to be true by **c89**. (The _POSIX_ macro is a reserved symbol and should not be used or modified.) The **c89** interface also passes the /Za option to the Microsoft Visual C++® compiler, which defines __STDC__, unless -N nostdc is specified. When -N nostdc is defined, the ANSI-only mode in the Visual C++ compiler is disabled and Microsoft extensions are allowed. The default mode is ANSI C mode.

The **cc** compiler interface defines the symbols __ INTERIX and UNIX to be true. (The UNIX macro is defined because many applications intended to be compiled on multiple platforms use this macro to call out features found on UNIX systems.) The cc interface also passes the /Ze option, which enables language extensions.

The Interix header files are structured to align with the Single UNIX Specification. For example, string and memory functions that occur in Portable Operating System Interface (POSIX).1 are in string.h. String and memory functions that are in the Single UNIX Specification, but not in POSIX.1, are in strings.h.

The include files are also structured to restrict the API namespace. If the macro _POSIX_SOURCE value is defined as 1 before the first header file is included, the program is restricted to the POSIX namespace. The program contains only those APIs that are specified in the POSIX standards. This can be restrictive.

To get all of the APIs provided with the Interix Software Development Kit (SDK), define the _ALL_SOURCE value as 1 before the first header file is included, as shown in the following example:

```
#define _ALL_SOURCE 1
#include <unistd.h>
```

You can also define _ALL_SOURCE in makefile with –D _ALL_SOURCE in the compiler flags section of the makefile. Then use the compiler flags in the compilation commands in the makefile.

This simplifies porting of the source code because this macro exposes all the define statements and prototypes in the requested header files.

If the source is not defined, the default is the more restrictive _POSIX_SOURCE.

## Packaging and Archiving Tools

Table 8.2 lists the archiving and packaging facilities used for UNIX applications and supported by Interix.

**Table 8.2. Archiving and Packaging Facilities**

| Archiving and Packaging Facilities | Description | Interix Support |
|---|---|---|
| **tar** | The tape archive program, which uses a variety of formats. It is one of the most popular formats. | Interix supports this tool. |
| **cpio** | Copies files into or out of a **cpio** or **tar** archive. It was intended to replace **tar**. The **cpio** archives can also be unpacked by **pax**. | Interix supports this tool. |
| **pax** | The POSIX.2 standard archiver. It reads, writes, and lists the members of an archive file (including **tar**-format archives). It also copies directory hierarchies. | Interix supports this tool. |
| **ar** | Creates and maintains groups of files that are combined into an archive. It is not usually used for source archives, but is almost always used for object file archives (static object libraries). | Interix supports this tool. |

| Archiving and Packaging Facilities | Description | Interix Support |
|---|---|---|
| **pkg_add** | Standard installation tool familiar to BSD users and very similar to the SUN and SV tool (of the same name). | This tool is available for downloading at http://www.interopsystems.com/tools/pkg_install.htm. |
| **rpm** | Powerful command-line–driven package management system capable of installing, uninstalling, verifying, and updating packages. | Interix has a set of tools to perform these activities:<br>• **pkg_add**. To install a package.<br>• **pkg_info**. To view status of installed packages.<br>• **pkg_delete**. To delete a package.<br>• **pkg_update**. To update or install new packages. |

Table 8.3 lists the compression formats that are used for UNIX applications.

**Table 8.3. Compression Formats for UNIX Applications**

| Compression Formats | Description | Interix Support |
|---|---|---|
| compress | Creates a compressed file with the adaptive Lempel-Ziv coding to reduce the size of files (typically 70 percent smaller than the original file). | Interix supports this tool. |
| zip/gzip | Algorithms combine a version of Lempel-Ziv coding (LZ77) with another version of Huffman coding in what is often called string compression. | Interix supports this tool. |
| pack | Compresses files using Huffman coding. | Interix supports this tool. |
| uncompress, zcat | Extracts compressed files. | Interix supports this tool. |
| gunzip | Decompresses files created through **compress**, **zip**, **gzip**, or **pack**. The detection of the input format is automatic. | Interix supports this tool. |
| unpack, pcat | Restores files compressed by **pack**. | Interix supports this tool. |
| bzip2 | Compresses files using the Burrows-Wheeler block sorting text compression algorithm and Huffman coding. | You can download this compression format at http://www.interopsystems.com/tools/warehouse.htm. |

Table 8.4 lists the common suffixes for archived and compressed file names.

**Table 8.4. Archived/Compressed File Suffixes**

| Suffix | Format | Description |
|---|---|---|
| .a | **ar** | Created by and extracted with **ar**. |
| .cpio | **cpio** | Created by and extracted with **cpio**. |
| .gz | **gzip** | Created by gzip and extracted with **gunzip**. |
| .tar | **tar** | Created by and extracted with **tar**. |
| .tgz | **tar, gzip** | A tar file that has been compressed with **gzip**. |
| .Z | **compressed** | Compressed by **compress**.<br>Uncompressed with **uncompress**, **zcat**, or **gunzip**. |
| .z | **pack**<br>**or**<br>**gzip** | Compressed by **pack** and extracted with **pcat**.<br>Compressed by **gzip** and extracted with **gunzip**. |
| .zip | **zip** | Compressed by **zip** and extracted with **unzip** or compressed by **pkzip** and extracted with **pkunzip**. |
| .bz2 | **bzip2** | Compressed by **bzip2** and extracted with **bunzip2** or **bzip2 –d**. |
| .tbz | **tar, bzip2** | A tar file compressed with **bzip2**. |

## Using Libraries

Microsoft linkers usually use the **LIB** environment variable to specify alternate search locations for libraries. The Interix **cc** and **c89** tools ignore the initial value of **LIB** to avoid conflicts with Windows tools. You can add additional libraries to the search path using the **-L** option, which you can specify multiple times on the command line.

You can also specify the C library, the lex, the math, and the yacc libraries with the traditional operands **-lc**, **-ll**, **-lm**, and **–ly**, respectively.

Many compilers use the **INCLUDE** environment variable to specify alternate search locations for header files. To avoid conflict with Windows tools, the **c89** tool ignores the initial value of **INCLUDE**. You can add additional directories to the search path with the **-I** option, which you can specify multiple times on the command line.

## Configuring the System

UNIX users generally configure the system by editing the configuration files with any of the available text editors. Many UNIX users and system administrators like the fact that much of the configuration for UNIX is stored in text files. The advantage is that you do not need to learn to use a large set of configuration tools. Familiarity with an editor and a scripting language serves the purpose. The disadvantage is that the information in the files comes in various formats, so you must familiarize yourself with the formats to change the settings. To manage a network, UNIX system administrators often employ scripts to reduce repetition and error. In addition, administrators can use the Network Information Service (NIS) to centralize the management of many standard configuration files. Although many versions of UNIX have GUI management tools, such tools are usually specific to each version of UNIX.

## *Startup Scripts and Logon/Logoff Scripts*

In UNIX, scripts are used during the startup to invoke most of the system and user processes. Such scripts include special scripts written by the systems manager, in addition to all the system services (such as networking and printing). The kernel starts **init,** a special process of UNIX that starts all other services and processes. It is configured through the /etc/inittab file. For BSD-style systems, **init** runs various rc scripts to configure services; and for System V–style systems, **init** runs scripts under the /etc/rc.d directory. Configuration of the characteristics of any service is carried out within /etc/inittab and the rc scripts.

The Interix subsystem supports the **init** tool. It is the first process that runs when the Interix subsystem starts. It is similar to /etc/init on traditional UNIX systems. However, the Interix version does not use /etc/inittab because Interix runs only at level 2.

When **init** starts, it executes all scripts in /etc/rc2.d in alphabetical order. The /etc/rc2.d directory contains symbolic links to the actual scripts located in /etc/init.d instead of the scripts themselves. The scripts are typically used to start and stop Interix daemons or to perform other tasks required when the system initializes or shuts down. The administrator can change the names of symbolic links in /etc/rc2.d to do the following:

- Control the order in which tools are run or daemons are started or stopped.
- Control whether a tool is run or a daemon is started or stopped.

## *Using inetd*

Windows Services for UNIX 3.5 provides **inetd,** which behaves like the UNIX **inetd**. Hence, no changes are required for a code that uses **inetd**.

The Interix **inetd** daemon is started by the **init** process and runs in the security context of the local Administrator. The **init** process is started automatically when the Interix subsystem starts. The services it starts (like **telnet**) are disabled by default.

After uncommenting the services you want **inetd** to run, send a SIGHUP signal to the **inetd** process. To start **inetd,** you must be the administrator because there are special privileges that some services will need and only the administrator has them (such as "root" on other UNIX systems).

Additional information is available at

http://www.microsoft.com/technet/interopmigration/unix/sfu/intdrutil.mspx.

**Note** The **inetd** program in Interix has an extra **-L** option, which is used to set the time lockout period to seconds. This is a security feature to mitigate denial-of-service (DoS) attacks. When the invocation rate of a service is exceeded (1000, by default), the service becomes unavailable for the time interval that you set. The default value is 180 seconds and the minimum value is 30 seconds.

The inetd.conf file is the configuration file for the **inetd** daemon. The file contains a list of Internet-related services that **inetd** can invoke when it receives a request from an Internet client.

However, the Windows system has one networking environment in common with Win32 and Interix applications. Therefore, it is not possible to have the Interix **telnetd** and the Win32 Telnet Service listening on the same standard telnet port at the same time. Hence, services that attempt to open standard ports used by other enabled Windows services like telnet should not be enabled in inetd.conf.

**Note** The user entry in inetd.conf should contain the user name of the user under whose account the server should run. This allows for servers to be given less permission than root. This column was originally added in UNIX to enhance security. On Interix, this field is always set to the string NULL and is ignored as it was not implemented in the current versions of Interix.

## Installation

Default installation of Windows Services for UNIX 3.5 does not install all useful tools. For example, if you plan to only use the network file system (NFS) portion, you may not want to install Interix, but you should still install user name mapping. If you want to develop UNIX code, you can install both the Interix SDK and Interix GNU SDK.

## Administration

The Domain Name System (DNS) is the hierarchical, distributed database. It stores information for mapping Internet host names to IP addresses and vice versa, mail routing information, and other data used by Internet applications. Clients look up information in the DNS by calling a resolver library, which sends queries to one or more name servers and interprets the responses.

If you face DNS errors, Interix has resolver routines that provide access to the Internet DNS. The resolv.conf configuration file contains information that is read by the resolver routines when they are invoked by a process for the first time. The file is designed to be read by users and contains a list of keywords with values that provide various types of resolver information. For example, you can check whether the /etc/resolv.conf file is configured to point to the right DNS.

During installation, the file is configured based on the information it gets from the system. However, if there is one DNS for external and one DNS for internal, then it might get configured to the wrong DNS. The information in resolv.conf can become incorrect because it is only set during the Windows Services for UNIX installation. Furthermore, for non-English Windows systems, the Windows Services for UNIX script used at installation puts incorrect information in resolv.conf. The updated BIND 9 tool can be used to configure and work with DHCP and non-English systems by ignoring the contents of the resolv.conf file. The BIND 9 software distribution contains both a name server and a resolver library.

**Note** Additional information is available at the Microsoft home page for Windows Services for UNIX 3.5 at http://www.microsoft.com/windows/sfu/default.asp.

Microsoft support for Windows Services for UNIX 3.5 is available at

http://support.microsoft.com/default.aspx.

You can access a newsgroup for discussion on Windows Services for UNIX 3.5 at

http://communities2.microsoft.com/communities/newsgroups/en-us/default.aspx?dg=microsoft.public.servicesforunix.general.

Answers to frequently asked questions and problems are available at the Interop Systems Web site at

http://www.interopsystems.com.

This site also provides a list of tools that can be downloaded to obtain added functionality.

**Note** Help is also available from the UNIX tools discussion forum at

http://www.interopsystems.com/tools/default.aspx.

A lot of information on possible errors is also documented in the Windows Services for UNIX 3.5 Help. You can also refer to the **man** pages.

**Notes:**

1. The event log helps fix administrative problems. Interix provides the syslogd, which is a system message logger. You can see the log file /var/adm/log/logger.
2. syslogd is not started automatically when the Interix subsystem starts. If you need to enable this service, remove the comment characters from the following lines in /etc/init.d/syslog and then start the service with the command:

   ```
   /etc/init.d/syslog start,
   ```

   or restart the computer.

   ```
   #  ${SYSLOGD}
   #  [ $? = 0 ] && echo "syslogd started"
   ```
3. The sendmail tool, although included with Windows Services for UNIX 3.5, is not supported as a full message transfer agent (MTA) by Microsoft.

# *Interoperability with Windows Services for UNIX 3.5*

This section discusses the scenario where Interix and Win32/64 and .NET might be required to interoperate with each other. The details are covered under the following sections:

- Running Win32-based Programs
- Encapsulating an Interix Application from a Win32 COM Object

## Running Win32-based Programs

The Interix subsystem extends the POSIX subsystem so that you can run a Win32-based, character-based user interface (CUI) and Win32-based graphical user interface (GUI) programs.

The Interix environment ships with the **runwin32** command, which simplifies the running of Win32 binaries. Shell scripts to invoke the standard built-in Windows command-line programs and CMD.exe are also provided in the /usr/contrib/win32/bin directory.

You can easily add the standard Windows command-line programs to your environment. The /usr/contrib/win32/bin directory is already added to your *PATH*.

The Interix ksh shell has been enhanced to run case-sensitive searches for Win32 programs as well.

After a Win32-based application is started, it interacts with the Win32-based subsystem, so there is no problem with the data generated inside the application. For example, in a file selection box, the path names are displayed in the Win32 format.

### *Interactions Between the Subsystems*

When you run Win32-based programs from the Interix command line, keep the following in mind:

- You must specify the Win32-based program either with a complete path name or by adding the Win32-based programs to your *PATH*. You need to do this only if ***PATH*_WINDOWS** is not set appropriately. The path name is case sensitive.
- The portion of the command line that specifies the Win32-based program must be in the Interix format because it is handled by the Interix subsystem. The portion of the command line passed to the Win32-based program must be in the format that the Win32-based program supports.
- If the Win32-based program makes use of environment variables, they must be in a format that is supported by a Win32-based program.
- If other Interix programs use the environment variables, they must be converted back to a format that the Interix programs can use.
- Running a Win32-based program involves the interaction of the two different permission models: the Interix POSIX model and the Win32 ACL model.

  The return value or exit status of a Win32-based program may not have any significance.
- You cannot run a Win32 interpreter program directly with a #! line in a shell script. Instead, you must use *#!/bin/sh* or *#!/bin/ksh* . The ActiveState Perl is an exception. It is possible to have *#!/Perl/bin/Perl.exe* as the first line of the shell script, but note that this will work only from the Interix environment. **cmd.exe** does not support this construct.
- The Win32-based program may expect any text files to be in the Win32 format (end-of-line marked by **CR-LF**) instead of the POSIX format (end-of-line marked by **LF**).

  You can use the **flip**(1) tool, which is a file interchange program that converts text-file formats between POSIX and other formats (such as MS-DOS® and Apple Macintosh). The **flip**(1) tool converts lines ending with carriage-return (CR) and linefeed (LF) (MS-DOS) or just carriage-return (Apple) to lines ending with just linefeed, or vice versa.
- When creating pipelines with Win32 commands, you may need to use the **cat32.exe** program.

**Note** For more information on cat32, refer to the Help pages of Windows Services for UNIX 3.5 or manual page of cat32 in Interix environment.

### *Adding Win32-based Programs to Your PATH*

You can run such programs as **ATTRIB.exe**, **CACLS.exe,** and **REGEDT32.exe** from the Interix shells. You can also redirect the input and output of these programs.

You can include the Win32 commands to your environment by adding the appropriate directories to your *PATH*. For example:

```
"${PATH}:/dev/fs/C/WINNT/system32"
```

**Note** Add the pathname in the POSIX file name format and not the Win32 format. Note that the case of the directory must exactly match that of the file system.

When you run a Win32-based program, the Interix subsystem converts your *PATH* variable back to the Win32 format so that the Win32-based program has the current *PATH*. This is the only environment variable that is converted for you; all others must be handled.

**Note** Typing the entire file name in the correct case can be difficult. To shorten the frequently used commands, you can use aliases, links, or a shell script.

### *Path Names*

The **unixpath2win** tool converts a UNIX path name to a Win32 path name. The **winpath2unix** tool converts a Win32 path name to a POSIX path name.

For example, you can easily convert a path from one format to another. To convert the \\inxsrv\public path to an Interix format, use the following code:

```
$ winpath2unix '\\inxsrv\publics'
/net/inxsrv/publics
$ winpath2unix 'C:\WINDOWS\system32'
/dev/fs/C/WINDOWS/system32
```

To convert the path back to Win32 format, use the following code:

```
$ unixpath2win /net/inxsrv/publics
\\inxsrv\publics
$ unixpath2win /dev/fs/C/WINDOWS/system32
C:\WINDOWS\system32
```

### *Environment Variables*

Environment variables are used to store information required by several tools or applications. Often, this information is a path name or a set of command options.

You can change an environment variable in a shell script without fear of conflicts because the environment of the shell script ends when the script does. Conflicts can arise when you change the environment variable in your current environment or when an Interix tool later in the script needs the value of the same environment variable.

The usual practice is to convert the environment variable before calling the Win32-based program, and then convert it back to the UNIX format after the Win32-based program exits. If the environment variable contains a directory, you can use the **unixpath2win** and **winpath2unix** tools.

### *PATH_WINDOWS*

The Interix **ksh** shell has been enhanced to support Windows style path searching. Using the *PATH*_WINDOWS environment variable, it is possible to find **.exe**, **.bat**, and other files at the **CMD.exe** prompt. You can use this variable to specify the directories that require a searching and suffix matching mechanism that is not case sensitive.

The Interix profile file sets the *PATH_WINDOWS* environment variable in which you can specify a suffix matching order.

## *Inherited Environment Variables*

The environment of your Interix shell session is built by both the Interix subsystem and your startup files. When the Interix subsystem starts, it converts the Win32 environment in the following ways:

- All environment variable names are converted to all uppercase—for example, *Path* becomes *PATH*.
- The contents of the *PATH* environment variable are converted from the Win32 format to the POSIX format.
- A new *HOME* environment variable is built from the *HOMEDRIVE* and *HOMEPATH* variables. If you already have *HOME* defined in your system control box, it is ignored.

The global startup file */etc/profile* adjusts the environment in the following ways:

- The old *PATH* is stored in *PATH_ORIG* and a new *PATH* is constructed.
- *TMPDIR* is set to the first of *$OPENNT_TMPDIR*, *$TMPDIR*, *$TMP*, or *$TEMP* that actually exists. TMP and TEMP are not converted because they may be used by Win32-based programs.
- **TERM**, **TERMCAP**, **EDITOR**, **VISUAL**, and **FCEDIT** are set.
- **SHELL** is unset.

## *Redirecting Standard Input, Output, and Error*

Win32-based programs can accept input redirected from the standard in (as in, `< file`). You can also redirect their standard output (with, `> file`) and their standard error.

Win32-based programs can also be used in command pipelines. However, Win32-based programs are not required to behave as expected (in terms of the three standard file streams). To provide a behavior that is more robust, when piping to and from Win32-based programs, Interix includes a Win32 tool, **cat32.exe**, which is just a "better behaved" filter. If you experience problems with a particular Win32-based program in a pipeline, try inserting **cat32** into the pipeline. For example:

```
$ net.exe users | cat32 | more
```

## *Useful Tools*

Two tools provided with Interix make it easier to handle the interaction between the two environments:

- **runwin32**. The **runwin32** tool runs a Windows command. The cmd can be any file with a .exe extension found in the directories *$WINDIR* (typically, */dev/fs/C/WINDOWS* on Windows XP and Windows Server 2003 or later*)*, *$WINDIR/system32*, or any built-in command of **CMD.exe**. You can run the Win32 commands directly from the Interix shells, but using **runwin32** eliminates the need to change your *PATH* environment variable to include the Windows or system32 directories present in your computer.
- **wvisible.** The **wvisible** tool returns true if the current window station is visible and returns false if it is not. The term "window station" is peculiar to Windows. Every Win32 process is associated with a window station object. If the window station is visible, Win32 windows are displayed on the screen of the computer, and the user can interact with these windows using the keyboard and mouse. If the window station is not visible, Win32 windows are invisible and noninteractive.

## Encapsulating an Interix Application from a Win32 COM Object

The Component Object Model (COM) is a binary-level specification that allows you to build applications using interchangeable components. You can write the language-independent components in C++, Java, C, or the Korn shell because the specification is binary. Client applications that make use of COM are largely Win32-based (although there is nothing in COM itself that restricts it to a Windows environment). Applications that use COM to bind components together can upgrade or customize those components dynamically. This section describes a method for encapsulating a UNIX application in a Win32 DLL (as a COM object).

For example, consider an application that writes to standard output. You must define a COM interface to pass this information. The encapsulating DLL is written to invoke the Interix application (your UNIX code that is ported to Interix) with the correct command-line options. The routine interprets the output of the application and passes it to the client application.

Defining the interface and writing the DLL are not trivial, but the basic concept is simple. On an Interix system, the Win32 world can invoke an Interix application as arguments to the **POSIX.exe** program. **POSIX.exe** serves a variety of purposes in the Interix environment, but its primary role is to serve as the access mechanism for the Win32 subsystem of Windows NT® to start an Interix process.

Figure 8.2 shows the architecture of the completed system.

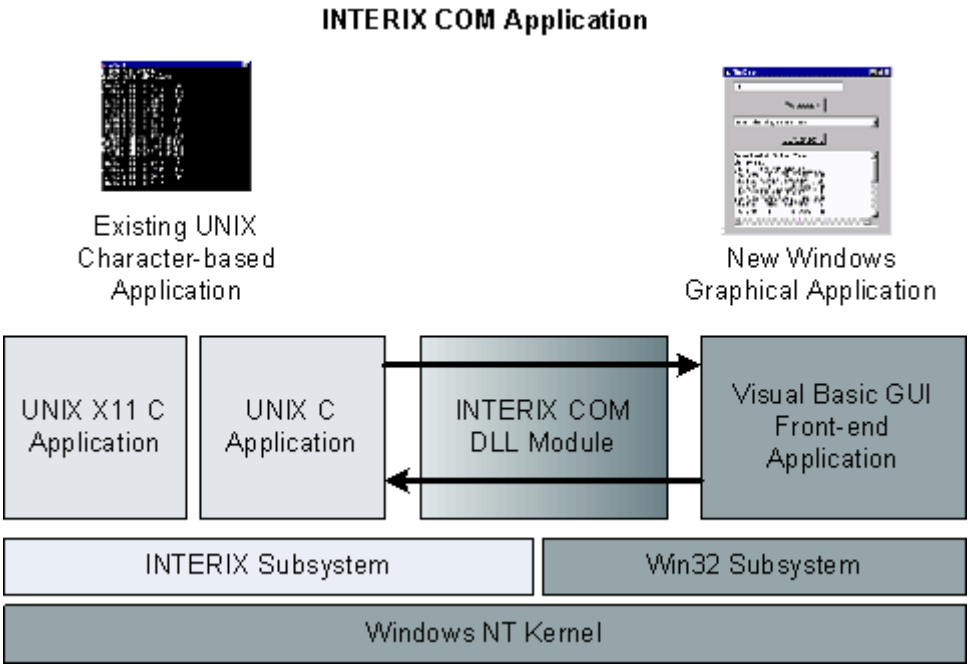


**Figure 8.2. Architecture of Interix COM application**

The client makes a call to the Interix COM DLL module. The DLL invokes **POSIX.exe** (and thereby, the Interix subsystem) to run the Interix C application. The DLL captures the output and passes it back to the Microsoft Visual Basic® GUI front-end application. The DLL does not have exclusive use of the application, and it can be invoked by another Interix user (for instance, logged on over telnet) while it is being called by the Visual Basic GUI front-end application.

To build a DLL that wraps a UNIX application, you need to think like both a UNIX programmer and a COM programmer. The UNIX programmer has to think about getting the command to produce the correct output. The COM programmer has to think about capturing that output. The difficult part is to get the **POSIX.exe** command line correct.

**To build the Interix COM DLL module**

1. Build the Interix application.
2. Define the COM interface.
3. Implement the interface in the DLL. The basic idea is to invoke POSIX.exe to run the Interix application, capture the standard output, and pass that data. For this, you need to get the application command line correct for POSIX.exe. Command-line quoting, if any, must be handled carefully.
4. Build the DLL.

**Note** Additional information about building the DLL is available at http://www.microsoft.com/technet/interopmigration/unix/sfu/intrxcom.mspx.

## Interacting with a Win32 Application Using Memory-Mapped Files

Memory mapping is the technique of making a part of the address space appear to contain a file or device so that ordinary memory accesses act on it.

Memory mapping uses the same mechanism as used by virtual memory to "trap" accesses to parts of the address space so that data from the file or device can be paged in (and other parts paged out) before the access is completed.

An Interix application can interact with the Win32 application using the same memory-mapped file. The following example shows how the Win32 program and the Windows Services for UNIX 3.5 program communicate with each other using a memory-mapped file. Compile both the programs and execute one with argument 1 and the other with argument 2.

Following is a Windows Services for UNIX example:

```
#include <fcntl.h>
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include <sys/mman.h>

#define BufferSize 256

char *Messages[][2]={
      /* Talk of '1' */            /* Talk of '2' */
 {"Hello '2', How are you?",       "Not bad '1', how about
yourself?"},
 {"What's up '2'?",                "Not much '1'"},
 {"It could be a lot worse!",      "Yup!"},
 {"Well, I guess that is that, '2'!",     "I guess so '1'..."},
 {0,0}
};

typedef struct ShareData_tag{
 char P1_speaks;  /* 'T'=true, 'F'=false */
 char P2_speaks;  /* 'T'=true, 'F'=false */
 char Mess_p1_to_p2[BufferSize];
 char Mess_p2_to_p1[BufferSize];
```

```
} SHARED_DATA;

SHARED_DATA SharedDataInit={
 'F','F',{0},{0}
};

const char *pMemMapFileName="/dev/fs/C/common.mem";


int main( int argc, char *argv[]){
 int Step=0, fd, Done=0;
 SHARED_DATA *pSharedData;
 void *pMem;

 if( argc<2){
  printf("Please invoke with either '1' or '2'\n");
  exit(1);
 }

 if( -1 != (fd=open( pMemMapFileName, O_RDWR | O_CREAT, 0777 ))){
            /* mmap seems to fail without this write on empty file */
  write( fd, &SharedDataInit, sizeof(SHARED_DATA));

  pMem=mmap( 0, sizeof(SHARED_DATA), PROT_READ | PROT_WRITE,
MAP_SHARED, fd, 0);
  if( (void*)-1 != pMem){
   pSharedData=(SHARED_DATA *)pMem;
   while( !Done){

            /************************
             * Personality of '1':
             * - Will talk if both are silent
             * - Will stop talking if both are talking
             */
    if( '1'==argv[1][0]){
     if( 'F'==pSharedData->P1_speaks){
      if( 'F'==pSharedData->P2_speaks){
       strcpy( pSharedData->Mess_p1_to_p2, Messages[Step][0]);
       pSharedData->P1_speaks='T';

      }
     }
     else{
```

```
      if( 'T'==pSharedData->P2_speaks){
       printf("[1,%d] P2 says: %s\n",Step,pSharedData->Mess_p2_to_p1);
       pSharedData->P1_speaks='F';
       Step++;
       if( !Messages[Step][0]) Done=1;
      }
     }
    }
            /************************
             * Personality of '2':
             * - Will talk if other was talking and it was silent
             * - Will stop talking if other is talking
             */
    else{
     if( 'F'==pSharedData->P2_speaks){
      if( 'T'==pSharedData->P1_speaks){
       printf("[2,%d] P1 says: %s\n",Step,pSharedData->Mess_p1_to_p2);
       strcpy( pSharedData->Mess_p2_to_p1, Messages[Step][1]);
       pSharedData->P2_speaks='T';
       Step++;
       if( !Messages[Step][0]) Done=1;
      }
     }
     else{
      if( 'F'==pSharedData->P1_speaks){
       pSharedData->P2_speaks='F';
      }
     }
    }
    sleep(1);
   }

            /* Wait for other process to finish */
   sleep(2);
   pSharedData->P1_speaks='F'; /* For next time we use the same file */
   pSharedData->P2_speaks='F';
   munmap( pSharedData, sizeof(SHARED_DATA));
  }
  else{
   perror( "Could not map memory\n");
  }
  close( fd);
 }
```

```
 else perror( "Could not create memory mapping file\n");
}
```
(Source File: InteropSFU-UAMV2C8.01.c)

Win32 example:

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

#define BufferSize 256

char *Messages[][2]={
      /* Talk of '1' */             /* Talk of '2' */
 {"Hello '2', How are you?",        "Not bad '1', how about
yourself?"},
 {"What's up '2'?",                 "Not much '1'"},
 {"It could be a lot worse!",       "Yup!"},
 {"Well, I guess that is that, '2'!",     "I guess so '1'..."},
 {0,0}
};

typedef struct ShareData_tag{
 char P1_speaks;  /* 'T'=true, 'F'=false */
 char P2_speaks;  /* 'T'=true, 'F'=false */
 char Mess_p1_to_p2[BufferSize];
 char Mess_p2_to_p1[BufferSize];
} SHARED_DATA;

SHARED_DATA SharedDataInit={
 'F','F',{0},{0}
};


const char *pMemMapFileName="C:\\common.mem";


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

 int Step=0, Done=0;

 SHARED_DATA *pSharedData;
      HANDLE hFile;
```

```
      HANDLE hMem;

 if( argc<2){
  printf("Please invoke with either '1' or '2'\n");
  exit(1);
 }

#ifdef XYZ
 if( -1 != (fd=open( pMemMapFileName, O_RDWR | O_CREAT, 0777 ))){

  write( fd, &SharedDataInit, sizeof(SHARED_DATA));

  pMem=mmap( 0, sizeof(SHARED_DATA), PROT_READ | PROT_WRITE,
MAP_SHARED, fd, 0);
  if( (void*)-1 != pMem){

   pSharedData=(SHARED_DATA *)pMem;
#endif
/*************************************************************/
 if (( hFile = CreateFile (pMemMapFileName /* szTmpFile */,
            GENERIC_WRITE | GENERIC_READ,
            FILE_SHARE_WRITE | FILE_SHARE_READ,
            NULL,
            OPEN_ALWAYS /* CREATE_ALWAYS */,
            FILE_FLAG_NO_BUFFERING | FILE_FLAG_WRITE_THROUGH,
                              NULL)) == INVALID_HANDLE_VALUE){
  printf("Could not create file %s\n",pMemMapFileName /* szTmpFile */);
  exit(1);
 }

                        /* Create file mapping. */
 if (!( hMem = CreateFileMapping ( hFile,
              NULL,
              PAGE_READWRITE,
              0,
              sizeof(SHARED_DATA),
              NULL))){

  if ( hFile){
   CloseHandle ( hFile);
   hFile = NULL;
  }
  printf("Mapping file failed\n");
```

```
  exit(1);
 }
 if( (pSharedData = (SHARED_DATA *)MapViewOfFile( hMem,
                 FILE_MAP_WRITE,
                 0,
                 0,
                         sizeof(SHARED_DATA)))){

  *pSharedData=SharedDataInit;
  FlushViewOfFile( pSharedData, sizeof(SHARED_DATA));

/*************************************************************/
   while( !Done){

            /************************
             * Personality of '1':
             * - Will talk if both are silent
             * - Will stop talking if both are talking
             */
    if( '1'==argv[1][0]){
     if( 'F'==pSharedData->P1_speaks){
      if( 'F'==pSharedData->P2_speaks){
       strcpy( pSharedData->Mess_p1_to_p2, Messages[Step][0]);
       pSharedData->P1_speaks='T';
       FlushViewOfFile( pSharedData, sizeof(SHARED_DATA));
      }
     }
     else{
      if( 'T'==pSharedData->P2_speaks){
       printf("[1,%d] P2 says: %s\n",Step,pSharedData->Mess_p2_to_p1);
       pSharedData->P1_speaks='F';
       FlushViewOfFile( pSharedData, sizeof(SHARED_DATA));
       Step++;
       if( !Messages[Step][0]) Done=1;
      }
     }
    }
            /************************
             * Personality of '2':
             * - Will talk if other was talking and it was silent
             * - Will stop talking if other is talking
             */
    else{
```

```
     if( 'F'==pSharedData->P2_speaks){
      if( 'T'==pSharedData->P1_speaks){
       printf("[2,%d] P1 says: %s\n",Step,pSharedData->Mess_p1_to_p2);
       strcpy( pSharedData->Mess_p2_to_p1, Messages[Step][1]);
       pSharedData->P2_speaks='T';
       FlushViewOfFile( pSharedData, sizeof(SHARED_DATA));
       Step++;
       if( !Messages[Step][0]) Done=1;
      }
     }
     else{
      if( 'F'==pSharedData->P1_speaks){
       pSharedData->P2_speaks='F';
       FlushViewOfFile( pSharedData, sizeof(SHARED_DATA));
      }
     }
    }
    Sleep(1000);
   }

            /* Wait for other process to finish */
   Sleep(2000);

/*************************************************************/
  UnmapViewOfFile ( pSharedData);
 }
 else printf("Could not obtain mem pointer\n");

 if( hMem){
  CloseHandle ( hMem);
  hMem=NULL;
 }
 if ( hFile){
  CloseHandle ( hFile);
  hFile = NULL;
 }
/*************************************************************/
#ifdef XYZ
   pSharedData->P1_speaks='F'; /* For next time we use the same file */
   pSharedData->P2_speaks='F';
   munmap( pSharedData, sizeof(SHARED_DATA));
  }
  else{
```

```
   perror( "Could not map memory\n");
  }
  close( fd);
 }
 else perror( "Could not create memory mapping file\n");
#endif
}
```

(Source File: InteropWin-UAMV2C8.021.cpp)

**Note** FlushViewOfFile writes to the disk a byte range, within a mapped view of a file. Flushing a range of a mapped view causes any dirty pages within that range to be written to the disk. Dirty pages are those whose contents have changed since the file view was mapped. Because of caching implementation of the file system support for memory-mapped files, this call may not be required. However, there is no harm in having this call other than when high performance is a critical factor.

# *Monitoring and Supporting the Applications*

System and application administrators usually want to use tools and scripts to manage migrated applications. With migrations to Interix, there is an option to also migrate any of the management tools that were used in the UNIX environment.

This section discusses the options available for managing and supporting migrated applications. In this case, Windows Services for UNIX 3.5 provides the tools that are necessary to port Perl and UNIX shell scripts to Win32, including those employing standard UNIX tools such as **awk**, **sed**, and **grep**.

## Deploying Interix Applications Using the Berkeley "r" Commands

The script copies the updated image of an application remotely to an Interix computer and modifies the startup symbolic link of the application to point to the new application. This installation script is copied (using **rcp**) to the target computer, and then executed using an **rsh** to copy a specified application directory to the target computer (where this script is executed). The following checks must be performed:

- Whether the platform is Interix.
- What the current application version is, if any.
- Whether there is enough disk space at the target computer.

The arguments to this script include:

- The directory or file on the target computer where the application will be installed.
- Disk space (in kilobytes) needed for the file or directory in which the application will be installed.
- The directory or file that must be removed. This must be different from the second argument.
- The source computer on which the installation directory of the application exists.
- The path of application on the source computer.

If you specify **-f**, the application directory or file is removed if it already exists, and then the installation is performed. If this option is not specified and the application directory or the file exists, it is not removed and the script results in error.

Following is an example of such a script with the relevant error messages:

**Script for Remote Deployment of Applications in Interix**

```
#!/bin/ksh
#
# ------------------------ Start of Functions-------------------------
--
CHECK_DISK_SPACE ()
{
 TARGET_DIR="$1"
 SPACE_REQ="$2"
 AVAILFILE1="$3"
 AVAILFILE2="$4"

 if [ ! -d "$TARGET_DIR" ]; then
  return 2
 fi

 cd "$TARGET_DIR"
 let AVAILSPACE=`/bin/df -k . | /bin/tail -1 | /bin/awk '{print $4}'`
 TARGET_DIRFILESYSTEM=`/bin/df -k . | /bin/tail -1 | /bin/awk '{print
$1}'`

 if [ -a "$AVAILFILE1" ]; then
  cd `dirname "$AVAILFILE1"`
  AVAILFILE1FILESYSTEM=`/bin/df -k . | /bin/tail -1 | /bin/awk '{print
$1}'`
  if [ "$AVAILFILE1FILESYSTEM" = "$TARGET_DIRFILESYSTEM" ]; then
   let AVAILSPACE="$AVAILSPACE"+`/bin/du -skx "$AVAILFILE1" 2>/dev/null
| /bin/awk '{print $1}'`
  fi
 fi

 if [ -a "$AVAILFILE2" ]; then
  cd `dirname "$AVAILFILE2"`
  AVAILFILE2FILESYSTEM=`/bin/df -k . | /bin/tail -1 | /bin/awk '{print
$1}'`
  if [ "$AVAILFILE2FILESYSTEM" = "$TARGET_DIRFILESYSTEM" ]; then
   let AVAILSPACE="$AVAILSPACE"+`/bin/du -skx "$AVAILFILE2" 2>/dev/null
| /bin/awk '{print $1}'`
  fi
 fi

 if [ "$AVAILSPACE" -lt "$SPACE_REQ" ]; then
  return 9
```

```
 fi

 return 0
}
# ------------------------- End of Functions --------------------------
-
if [ `uname` != "Interix" ]; then
 echo "The operating system is not Interix."
 exit 1
fi

FORCE="n"
if [ "$6" = "-f" ]; then
 FORCE="y"
fi

#dir/file where application will be installed
INSTALL_LOC="$1"

#disk space (in KB) needed in `dirname $INSTALL_LOC`
SPACE_REQ="$2"

#dir/file to be removed (should not be same as $INSTALL_LOC)
REMOVE_LOC="$3"

if [ -a "$INSTALL_LOC" -a "$FORCE" = "n" ]; then
 echo "The product (same or different level) is already installed."
 exit 2
fi

TARGET_DIR=`dirname "$INSTALL_LOC"`
CHECK_DISK_SPACE "$TARGET_DIR" "$SPACE_REQ" "$INSTALL_LOC"
"$REMOVE_LOC"
FLAG="$?"

if [ "$FLAG" = "9" ]; then
 echo "Not enough disk space is available."
 exit 3
elif [ "$FLAG" != "0" ]; then
 echo "The available disk space cannot be ascertained."
 exit 4
fi
```

```
rm -rf "$INSTALL_LOC"
rm -rf "$REMOVE_LOC"

#Source machine dns name where the application's installation directory
exists
SOURCE_HOST="$4"

#Directory path of application on source machine
SOURCE_LOC="$5"
rcp –r "$SOURCE_HOST":"$SOURCE_LOC" "$INSTALL_LOC"
```

## Using the Remote Deployment Script

You will be able to deploy your application to a remote computer by using the following instructions. The remote computer is called *targetmachine* in these instructions.

**To deploy your applications to a remote computer**

1. Copy the script install.ksh to target computer using rcp.
2. Execute the install script as follows:

   $ rsh *targetmachine* /dev/fs/C/tmp/install.ksh **/dev/fs/C/tmp/test1 200 /dev/fs/C/appdir/app1** ***sourcehost*** /dev/fs/E/SFU/app1

   Where the arguments are as follows:

   - **/dev/fs/C/tmp/test1**

     The directory or file in which application will be installed on the target computer.
   - **200**

     The disk space (in KB) needed for the file or directory in which the application will be installed.
   - **/dev/fs/C/appdir/app1**

     The directory or file to be removed (must be different from the second argument).
   - ***sourcehost***

     The source computer in which the installation directory of the application exists.
   - **/dev/fs/E/SFU/app1**

     The directory path of application on the source computer.

   If you attempt to run the script twice, the following is displayed:

   $ rsh *targetmachine* /dev/fs/C/tmp/install.ksh **/dev/fs/C/tmp/test1 200 /dev/fs/C/appdir/app1** ***sourcehost*** /dev/fs/E/SFU/app1

   $ "The product (same or different level) is already installed."

   This is because the application already exists on the target computer. If you use the **-f** option, the following is displayed because installation would have occurred twice:

   $ rsh *targetmachine* /dev/fs/C/tmp/install.ksh **/dev/fs/C/tmp/test1 200 /dev/fs/C/appdir/app1** ***sourcehost*** /dev/fs/E/SFU/app1 –f

# Testing Activities

This section discusses the testing activities designed to identify and address potential solution issues prior to deployment. Testing starts when you begin developing the solution and ends when the testing team certifies that the solution components meet the schedule and quality goals established in the project plan.

Testing in migration projects involving infrastructure services is focused on finding discrepancies between the behavior of the original application, as seen by its clients, and the behavior of the newly migrated application. All discrepancies must be investigated and fixed.

In the Developing Phase, the testing team executes the test plans for acceptance tests on the application submitted for a formal round of testing on the test environment. The testing team assesses the solution, makes a report on its overall quality and feature completeness, and certifies that the solution features, functions, and components address the project goals.

The inputs required for the Developing Phase include:

- Functional specifications document.
- A feature-complete application, which has been unit tested.

The documents that are used during the Developing Phase include:

- **Test plan**. The test plan is prepared during the Planning Phase. It should describe in detail everything that the test team, the program management team, and the development team must know about the testing to be done.
- **Test specification**. The test specification conveys the entire scope of testing required for a set of functionality and defines individual test cases sufficiently for the testers. It also specifies the deliverables and the readiness criteria.
- **Test environment**. The test environment is an exact replica of the live environment; it is used to test the application under realistic environments. It also describes the software, hardware, and tools required for testing purposes.
- **Test data**. The test data is a set of data for testing the application. Test data is usually a diverse set of data that helps test the application under different conditions.
- **Test report**. The test report is an error report of the tests done. It includes a description of the errors that occurred, steps to reproduce the errors, severity of the errors, and names of the developers who are responsible for fixing them.

  The test report is updated during the Stabilizing Phase and is also one of the outputs of this phase, along with the tested and stabilized application.

The key deliverables of the Developing Phase include:

- Application ready to be deployed on the production environment.
- Application source code.
- Project documentation and user manual.
- Test plan, test specification, and test reports.
- Release notes.
- Other project-related documents.

Testing behind with a code review of the application and unit testing. In the Developing Phase, the application is subjected to various tests. The test plan organizes the testing process into the following different elements:

- Code component testing
- Integration testing
- Database testing
- Security testing
- Management testing

You can test the migrated application in all the scenarios using a defined testing strategy. Although each test has a different purpose, together they verify that all system elements are properly integrated and perform their allocated functions.

## *Code Component Testing*

A component may be a class or a group of closely related classes performing a similar task. Component testing is the next step after unit testing. Component testing is the process of verifying a software component with respect to its design and functional specifications.

Component testing in a migration project is the process of finding the discrepancies between the functionality and output of components in the Windows application and the original UNIX application. Basic smoke testing, boundary conditions, and error test cases are written based on the functional specification of the component.

The code component testing round tests the components for:

- Functionality.
- Input and output, interactions within and with other components.
- Stress testing.
- Performance.

The test cases for component testing cover, either directly or indirectly, constraints on their inputs and outputs (pre-conditions and post-conditions), the state of the object, interactions between methods, attributes of the object, and other components. The code component testing requires the following inputs:

- **Test plan and specification**. It provides the test cases.
- **System requirements**. These are used to determine the required behaviors for individual domain-level classes. The use case model is also used to determine which parts of a component must be tested for vulnerabilities.
- **Specifications of the component**. The specifications are used to build the functional test cases. Information on the component inputs, outputs, and interactions with other components can be derived from here.
- **Design document**. The actual implementation of the design provides the information necessary to construct the structural and interaction test cases.

Components must also be stress tested. Stress testing is the process of loading the component to the defined and undefined limits. Each component must be stressed under a load to ensure that it performs well within a reasonable performance limit.

System CPU and memory usage per component can also be measured and monitored to know the performance of individual components. For this, you can use tools such as the Windows Performance Monitor. For more information, refer to the "Testing and Optimization Tools" section in Chapter 9, “Stabilizing Phase” of this volume.

## Integration Testing

Integration testing involves testing the application as a whole, with all the components of the application put together. Component testing is done during the testing performed in the Developing Phase. Integration testing is the process of verifying the application with respect to the behavior of components in the integrated application, interaction with other components, and the functional specifications of the application as a whole. Integration testing in a migration project is the process of finding discrepancies in the interaction between components and the behavior of components in the Windows application and the original UNIX application.

Integration testing tests the components for:

- Functionality, behavior of the application as a whole and the individual components after integration.
- Input and output, interactions within and with other components.
- Response to various types of stresses.
- Performance.

Test cases for integration testing directly or indirectly include functionality of the components, constraints on their inputs and outputs (pre- and post-conditions), the state of the object, interactions between components, attributes of the object, and other components. The application must also be stress tested. Inputs required for integration testing include:

- **Test plan**. It provides the details of testing the application.
- **Test specification**. It is used to determine the required behaviors for individual domain-level classes. The use case model is also used to determine which parts of the application must be tested for vulnerabilities.

Stress testing must also be performed. Stress testing is the process of loading the application to the defined and undefined limits to ensure that it performs well within a reasonable performance limit.

System testing is also performed after completion of integration testing. System testing is the process of ensuring that the integrated application is compatible with all platforms and to test against its requirements. The system CPU and memory usage for the application can also be measured and monitored to determine their performance. For this, you can use tools such as the Windows Performance Monitor.

**Note** More information on these tools is available at "Testing and Optimization Tools" in Chapter 9, “Stabilizing Phase” of this volume.

## Database Testing

The database component is a critical piece of any data-enabled application. In a migration project, the database may be the same or may have been replaced by another database. In both cases, data must be migrated to the respective database on Windows. Testing of a migrated database includes testing of:

- Migrated procedural code.
- Data integration with heterogeneous data sources (if applicable).
- Customized data transformations and extraction.

Database testing also involves testing at the data access layer, which is the point at which your application communicates with the database. Database testing in a migration project involves:

- Testing the data and the structure and design of the migrated database objects.
- Testing the procedures and functions related to database access.

- Security testing, which tests the database to guarantee proper authentication and authorization so that only the users with the appropriate authority access the database. The database administrator must establish different security settings for each user in the test environment.
- Testing of data access layer.
- Performance testing of data access layer.
- Manageability testing of the database.

An application maintains the following three databases, which are replicas of each other:

- **Development database**. This is where most of the testing is carried out.
- **Deployment database (or integration database)**. This is where the tests are run prior to deployment to ensure that the local database changes are applied.
- **Live database**. This has the live data and cannot be used for testing.

Database testing is done on the development database during development, and the integrated application is tested using the deployment database.

## *Security Testing*

Security is about controlling access to a variety of resources, such as application components, data, and hardware. Security testing is performed on the application to ensure that only the users with the appropriate authority are able to use the applicable features of the application. Security testing also involves testing the application from the point of providing the same security features and measures that were provide by the original application.

To ensure that the application is secure, most security measures are based on the following four concepts:

- **Authentication**. This is the process of confirming the identity of the users, which is one layer of security control. Before an application can authorize access to a resource, it must confirm the identity of the requestor.
- **Authorization**. It is the process of verifying that an authenticated party has the permission to access a particular resource, which is the layer of security control following the authentication layer.
- **Data protection**. It is the process of providing data confidentiality, integrity, and nonrepudiability. Encrypting the data provides data confidentiality. Data integrity is achieved through the use of hash algorithms, digital signatures, and message authentication codes. Message authentication codes (MACs) are used by technologies such as SSL/TLS to verify that data has not been altered while in transit.
- **Auditing**. It is the process of logging and monitoring events that occur in a system and which are of interest to security.

  **Note** For more information, refer to "Event Logging" on the TechNet Web site at http://technet2.microsoft.com/WindowsServer/en/Library/0473658c-693d-4a06-b95b-ebe8a76648a91033.mspx.

The systems engineer establishes different security settings for each user in the test environment. Network security testing is performed to guarantee that the network is secure from unauthorized users. To minimize the risks associated with unchecked errors on the system, you should know the user context in which system processes run, keeping to a minimum the privileges that these accounts have, and log their access to these accounts. Active monitoring can be accomplished using the Windows Performance Monitor for real-time feedback.

All security settings and security features of the application must be documented properly.

**Notes**

More information about security testing is available at

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vsent7/html/vxcontestingforsecurability.asp.

More information on how to make your code secure is available at

http://msdn.microsoft.com/security/securecode/.

More information on secure coding guidelines for the .NET Framework is available at

http://msdn.microsoft.com/security/securecode/bestpractices/default.aspx?pull=/library/en-us/dnnetsec/html/seccodeguide.asp.

# *Management Testing*

Testing for manageability involves testing the deployment, maintenance, and monitoring technologies that you have incorporated into your migrated application.

Following are some important testing recommendations to verify that you have developed a manageable application:

- **Test Windows Management Instrumentation (WMI)**. WMI can provide important information about your application and the resources it uses. During the design of your application, you made certain decisions about the types of WMI information that must be provided. These might include server and network configurations, event log error messages, CPU consumption, available disk space, network traffic, application settings, and many other application messages. You must test every source of information and be certain you can monitor each one.

  **Note** More information on usage of WMI in applications is available at

  http://msdn.microsoft.com/library/default.asp?url=/library/en-us/wmisdk/wmi/wmi_reference.asp.

- **Test Network Load Balancing (NLB) and cluster configuration**. You can use Application Center 2000 clustering to add a front-end or back-end server while the application is still running. After installing new server hardware on the network, use your monitoring console to replicate the application image and start the server. The new server should automatically begin sharing some of the workload. You can set up the Application Center 2000 Performance Monitor (PerfMon) to track multiple front-end Web servers. After setting up PerfMon, make some requests to generate traffic. PerfMon will show you that there is an increase in traffic in the back-end servers and that the workload is evenly spread across the front-end computers.

  **Note** More information about Application Center 2000 is available at

  http://www.microsoft.com/applicationcenter/.

- **Test change control procedures**. An important part of application management is the handling of both scheduled and emergency maintenance changes. Test and validate all of the change control procedures including the automated and manual processes.

  It is especially important to test all people-based procedures to ensure that the necessary communication, authority, and skills are available to support an error-free change control process.

  **Note** More information on this is available on the MSDN Web site at

  http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vsent7/html/vxcontestingformanageability.asp.

## Interim Milestone: Internal Release *n*

The project needs interim milestones to help the team measure their progress in the actual building of the solution during the Developing Phase. Each internal release signifies a major step toward the completion of the solution feature sets and achievement of the associated quality level. Depending on the complexity of the solution, a number of internal releases may be required. Each internal release represents a fully functional addition to the solution's core feature set, indicating that it is potentially ready to move on to the Stabilizing Phase.

## Closing the Developing Phase

Closing the Developing Phase requires completing a milestone approval process. The team documents the results of different tasks that it has performed in this phase and obtains a sign-off on the completion of development from the stakeholders (including the customer).

### *Key Milestone: Scope Complete*

The Developing Phase culminates in the Scope Complete Milestone. At this milestone, all features are complete and the solution is ready for external testing and stabilization. This milestone is the opportunity for customers and users, operations and support personnel, and key project stakeholders to evaluate the solution and identify any remaining issues that must be addressed before beginning the transition to stabilization and, ultimately, to release.

Key stakeholders, typically, representatives of each team role and any important customer representatives who are not on the project team, signal their approval of the milestone by signing or initialing a document stating that the milestone is complete. The sign-off document becomes a project deliverable and is archived for future reference.

Now the team must shift its focus to verify that the quality of the solution meets the acceptance criteria for release readiness. The next phase, the Stabilizing Phase, describes the activities (for example, user acceptance testing (UAT), regression testing, and conducting the pilot) required to achieve these objectives.

# Chapter 9: Stabilizing Phase

This chapter describes the suggested strategy for stabilizing an application that has been migrated from UNIX to the Microsoft® Windows® operating system. The Stabilizing Phase involves testing the application for the expected functionality and improving the quality of the application to meet the acceptance criteria set for the project.

This chapter also describes the objectives of testing in the Stabilizing Phase. It introduces testing processes and methodologies that can be used to test applications with different architectures. You need to test the applications to verify that they meet the expected functionality and acceptance criteria set for the project. Various tools that you can use to test applications are also discussed here. The information in this chapter will enable you to choose the most appropriate tools for testing your application.

## Goals for the Stabilizing Phase

This section describes the primary goals that you need to accomplish in the Stabilizing Phase. This section acquaints you with the major tasks to be performed and the deliverables expected from the Stabilizing Phase.

The primary goal of the Stabilizing Phase is to improve the quality of the solution so that it meets the acceptance criteria and can be released into the production environment. During this phase, the team tests the feature-complete migrated application. At this point in the Stabilizing Phase, the applications are subjected to various tests such as user acceptance testing (UAT), regression testing, and bug tracking based on the application's requirements. The build must demonstrate that it reaches the defined quality and performance levels and is ready for full production deployment.

Testing during the Stabilizing Phase is an extension of the testing that was conducted during the development of the application in the Developing Phase. Testing in the Stabilizing Phase tests usage and operation of the application under realistic conditions. Test plans include testing the functionality in the migrated application and making a comparison of the migrated application's functionality with that provided by the original application. Test plans also must include test cases for testing the new features added to the application.

After a build is stabilized, the solution is deployed. This phase culminates with the Release Readiness Approved Milestone, indicating that the team and customer agree that all the outstanding issues have been addressed.

## *Major Tasks and Deliverables*

Table 9.1 describes the major tasks that must be completed during the Stabilizing Phase and lists the processes and roles responsible for achieving them.

**Table 9.1. Major Stabilizing Phase Tasks and Owners**

| Major Tasks | Owners |
|---|---|
| **Testing the solution**<br>The team executes the test cases that were created during the Planning Phase and enhanced and tested during the Developing Phase. Testing includes comparing the test results of the parent application and the migrated application as well as testing the applications from different perspectives. | Test |
| **Resolving defects**<br>The team triages the defects identified and resolves them. New tests are developed to reproduce issues reported from other sources. The new test cases are integrated into the test suite. | Development, Test |
| **Conducting the solution pilot**<br>This task involves setting up the deployment environment and the migrated application on the staging area to test the application before it is deployed. The team moves a solution pilot from the development area to a staging area to test the solution with the actual users and real scenarios. It also includes testing the solution in a live environment. The solution pilot is conducted before starting the Deploying Phase. | Release Management |
| **Closing the Stabilizing Phase**<br>The team documents the results of the tasks performed in this phase and solicits management approval at the Release Readiness Approved Milestone meeting. | Project |

Table 9.2 lists the tasks described in Table 9.1 and considers the tasks from the perspective of the team roles. The primary team roles directing the Stabilizing Phase are Test and Release Management.

**Table 9.2. Role Cluster Focuses and Responsibilities in Stabilizing Phase**

| Role Cluster | Focus and Responsibility |
|---|---|
| Product Management | Execute communications plan and launch test phase. |
| Program Management | Track project and bug triage. |
| Release Management | Preparation for deployment of the application and setting up the production environment. |
| Development | Bug triage and resolution, code optimization, and hardware or service reconfiguration. |
| User Experience | Stabilization of user documentation and training materials. |
| Test | Generate build and triage plan.<br>Track test schedule.<br>Review bugs entered in the bug-tracking tool and monitor their status during triage meeting.<br>Generate weekly status reports.<br>Escalate issues that are blocking progress, review impact analysis, and generate change management document.<br>Ensure that the appropriate level of testing is achieved for a particular release.<br>Lead the actual Build Acceptance Test (BAT) execution.<br>Execute test cases and generate test report. |

# Testing the Solution

This section describes the testing activities that are performed in the Stabilizing Phase. In the Stabilizing Phase, testing is performed not only on individual components of the solution, but on the solution as a whole, because all features and functions of the solution are now complete, and all solution elements have been built. The testing that began during the Developing Phase according to the test plan created during the Planning Phase continues with further testing, tracking, documentation, and reporting activities. This mainly involves UAT and regression testing as explained in the next subsections in detail.

## User Acceptance Testing (UAT)

The Stabilizing Phase emphasizes UAT to ensure that the migrated solution meets the business needs. UAT is performed on a collection of business functions in a production environment after the completion of functional testing. This is the final stage in the testing process before the system is accepted for operational use. It involves testing the system with data supplied by the actual user or customer instead of the simulated data developed as part of the testing process. The result of UAT confirms that the solution meets the overall user requirements and determines the release readiness status of the system. Running a pilot for a select set of users helps to identify areas where users have trouble understanding, learning, and using the solution.

For migration projects, UAT involves testing the migrated application and identifying the defects. These defects are addressed and regression testing is conducted for each fixed defect to ensure the fix doesn't break any other functionality of the migrated application. The UAT Summary confirms that the solution meets the customer's acceptance criteria, thereby assisting in customer acceptance of the solution.

## Regression Testing

Regression testing refers to retesting previously tested components and functionality of the system to ensure that they function properly even after a change has been made to parts of the system. For migration projects, this is the most important class of tests. As defects are discovered in a component, modifications should be made to correct them. This may require retesting of other components or the entire solution in the testing process.

The regression test helps in the following areas:

To ensure that no new problems are introduced and that the operational performance is not degraded because of the modifications.

To ensure that the effects of the changes are transparent to other areas of the application and other components that interact with the application.

To modify the original test data and test cases from other levels of testing.

# Resolving the Solution Defects

In order to resolve defects, they must be reproduced and tested in the test environment. Each reproduced defect in the test environment should be tracked with its status and severity. An important aspect of such tests involves test tracking and test reporting. Test tracking and reporting occurs at frequent intervals during the Developing and Stabilizing Phases. During the Stabilizing Phase, this reporting is driven by the bug count. Regular communication of the test status to the team and other key stakeholders ensures that the project runs smoothly. After fixing the defects, test cases and test data should be updated and integrated with the test suite.

## Bug Convergence

Bug convergence is the point at which the team makes visible progress against the active bug count. At bug convergence, the rate of bugs resolved exceeds the rate of bugs found, thus the actual number of active bugs decreases. After bug convergence, the number of bugs should continue to decrease until the zero bug bounce task, as explained in the next sections.

## Interim Milestone: Bug Convergence

Bug convergence tells the team that most of the bugs have been addressed and the rate of bugs resolved is higher than the rate of new bugs found. This can be considered as the interim milestone and the migrated application can be taken for zero bug bounce verification.

# *Zero Bug Bounce*

Zero bug bounce is the point in the project when development finally catches up to testing and there are no active bugs for the moment. After zero bug bounce, the number of bugs should continue to decrease until the product is sufficiently stable for the team to build the first release candidate.

## Interim Milestone: Zero Bug Bounce

Achieving zero bug bounce is a clear sign that the solution is near to being considered a stable release candidate.

# *Release Candidates*

After the first achievement of zero bug bounce, a series of release candidates are prepared for release to the pilot group. Each release is marked as an interim milestone.

Guidelines for declaring a build a release candidate include the following:

- Each release candidate has all the required elements to qualify for release to production.
- The test period that follows determines whether a release candidate is ready to release to production or if the team must generate a new release candidate with appropriate fixes.
- Testing the release candidates, carried out internally by the team, requires highly focused and intensive efforts and concentrates heavily on discovering critical bugs.

## Interim Milestone: Release Candidate

As each new release candidate is built, there should be fewer bugs reported, classified, and resolved. Each release candidate marks significant progress in the team's approach toward deployment. With each new candidate, the team must focus on maintaining tight control over quality.

## Interim Milestone: Preproduction Test Complete

Eventually, a release candidate is prepared that contains no defects. After this has occurred, no defects should be found within the isolated staging environment. At this stage, all testing that can be done before putting the migrated component into production has been completed.

# Conducting the Solution Pilot

This section describes the best practices to be adopted for conducting a pilot of the migrated application. This section provides you with information regarding various points to be considered while conducting a pilot and deciding the next steps to take after the pilot.

A pilot release is a deployment into a subset of the live production environment or user group. During the pilot, the team tests as much of the entire solution as possible in a true production environment. Depending on the context of the project, the pilot can take various forms:

- In an enterprise, a pilot can be a group of users or a set of servers in a data center.
- For migration projects, the pilot might involve testing the most demanding application or database that is being migrated with a sophisticated group of users that can provide helpful feedback.

The common element in all the piloting scenarios is testing under live conditions. The pilot is not complete until the team ensures that the solution is viable in the production environment and that the solution is ready for deployment.

Some of the best practices that should be followed while conducting a pilot are:

- Before beginning a pilot, the team and the pilot participants must clearly identify and agree upon the success criteria of the pilot. These should map back to the success criteria for the development effort.
- Any issues identified during the pilot must be resolved either by further development, by documenting resolutions and workarounds for the installation team and production support staff, or by incorporating them as supplemental material in training or help documentation.
- Before the pilot is started, a support structure and an issue-resolution process must be in place. This may require the support staff getting trained in the application area that is being piloted.
- To identify any issues and confirm that the deployment process will work, it is necessary to implement a trial run or a rehearsal of all the elements of the deployment prior to the actual deployment.

After you collect and evaluate the pilot data, a corresponding strategy should be selected based on the findings from the analysis of pilot data. The next strategy could be one of the following:

- **Stagger forward.** Deploy a new release to the pilot group.
- **Roll back.** Execute the rollback plan and revert the pilot group to the stable state they had before the pilot started.
- **Suspend.** Suspend the entire pilot.
- **Fix and continue.** If you find an issue during the pilot, fix the issue and continue with the next steps.
- **Proceed.** Advance to the Deploying Phase.

After the pilot has been completed, the pilot team must prepare a report detailing each lesson learned and how new information was incorporated and issues were resolved.

## Interim Milestone: Pilot Complete

This milestone signifies that the pilot has been successfully completed and that the team is ready to proceed to the Deploying Phase.

# Closing the Stabilizing Phase—Release Readiness Approved

The Stabilizing Phase culminates with the Release Readiness Approved Milestone. The team builds a release candidate (with all the major defects fixed) that satisfies the necessary quality policy of the organization. All rounds of testing must be completed, meaning that all test plans have been executed and test cases satisfied before the migrated component can be moved into the production environment. Then the release is approved with a formal sign-off marking that the Release Readiness Approved Milestone has been reached.

Key stakeholders, typically representatives of each team role and any important customer representatives who are not on the project team, signal their approval of the milestone by signing or initialing a document stating that the solution is complete and approved for release. The sign-off document becomes a project deliverable and is archived for future reference.

The performance of the application following deployment in the production environment is a key criterion in indicating a successful application migration. The following sections will help you to optimize the performance of the application and the tools following deployment.

# Tuning

This section discusses tuning of the solution in detail, including how to performance-tune the migrated application, and scaling up and scaling out of the application. In addition, the section discusses multiprocessor considerations for applications and network utilizations. You can use this information to identify the parameters that affect application performances and the steps to consider in the scalability of the applications.

## *Performance Tuning*

Performance management starts with the gathering of a data baseline that indicates what system performance should look like. After establishing a baseline, it is used to evaluate the performance of the application. Performance problems typically do not become apparent until the application is placed under an increased load.

Measuring the performance of an application when placed under ever increasing loads determines the scalability of that application. When the performance begins to fall below the stated minimum performance requirements, you have reached the limit of scalability of the application. For more information about scaling, refer to the "Scaling Up and Scaling Out" section later in this chapter.

Performance tuning can be carried out in the following ways:

- Tuning the computer hardware by adding more memory, updating CPUs, adding disk controllers, or upgrading network controllers. This is the most efficient way and helps performance-tune the application as well.
- Application rearchitecture to remove bottlenecks such as poor threading and looping and checking for other loops that use too much CPU time. This step also helps considerably in performance tuning.
- Operating system parameter tuning. This involves adjusting the amount of page store and tweaking network stack parameters.
- Tuning of the configurations on a database server, application server, or Web server.

In UNIX, performance is monitored using a type of kernel-level instrumentation, along with rudimentary tools for monitoring the CPU, disk, and memory usage.
Windows Server™ 2003 is designed such that it exposes a great deal of performance data. Tools like Windows Performance Monitor (PerfMon) can be used to export detailed information about the processor, memory, disk, and network usage. Performance Monitor support is integrated throughout Windows. Administrators can gather a variety of performance data from many computers simultaneously.

UNIX kernels tend to have many configurable parameters that can be fine-tuned for specific applications. By contrast, the Windows kernel is largely self-tuned. The virtual memory, thread scheduling, and I/O subsystems all dynamically adjust their resource usage and priority to maximize throughput. The difference between these two approaches is that the UNIX approach is to tweak kernel parameters for maximum advantage in the benchmark, even if those tweaks affect the real-world performance, while the Windows approach is to let the kernel tune itself for whatever load is placed on it.

**Notes**

More information on improving performance and writing faster managed code is available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndotnet/html/fastmanagedcode.asp.

More information on writing high-performance managed applications is available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndotnet/html/highperfmanagedapps.asp.

## *Scaling Up and Scaling Out*

Scalability is a measure of how easy it is to modify the application infrastructure and architecture to meet variances in utilization. As with other application capabilities, the decisions you make during the design and early coding phases largely dictate the scalability of your application.

Application scalability requires a balanced partnership between two distinct domains: software and hardware.

Scaling up is achieving scalability with the use of better, faster, and more expensive hardware to move the processing capacity limit from one part of the computer to another. Scaling up includes adding more memory, adding more or faster processors, or just migrating the application to a more powerful, single computer. Typically, this method allows for an increase in capacity without requiring changes to source code. However, adding CPUs may not add performance in a linear fashion. Instead, the performance gain curve slowly tapers off as each additional processor is added.

Scaling out distributes the processing load across more than one server by dedicating several computers to common tasks. In this scenario, the fault tolerance of the application can be increased. Scaling out also presents a greater management challenge because of the increased number of computers.

Developers and administrators use a variety of load-balancing techniques to scale out with the Windows platform. Load balancing allows an application to scale out across a cluster of servers, making it easy to add capacity by adding replicated servers. It provides redundancy, giving the site failover capabilities so that it remains available to users even if one or more servers fail or are taken down.

Scaling out provides a method of scalability that is not hampered by hardware limitations. Each additional server provides a near linear increase in scalability.

The key to successfully scaling out of an application is location transparency. If any of the application code depends on knowing which server is running the code, location transparency has not been achieved and scaling out will be difficult. This situation requires code changes to scale out an application from one server to many, which is seldom an economical option. If you design the application with location transparency in mind, scaling out becomes an easier task.

**Notes**

More information on scaling is available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vsent7/html/vxconmanageabilityoverview.asp.

Microsoft Application Center 2000 reduces the complexity and the cost of scaling out. More information on Application Center 2000 is available at http://www.microsoft.com/applicationcenter/default.mspx.

More information on scaling network-aware applications is available at http://msdn.microsoft.com/library/default.asp?url=/msdnmag/issues/1000/Winsock/toc.asp.

# Multiprocessor Considerations

Application performance improves by having multiple processors perform the same task. You can distribute the processing load across the several processors.

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 with a faster computer, a more efficient algorithm, and by improving the implementation or using more processors. You can improve the performance with the help of tuning techniques as well.

Using additional processors can mean taking advantage of an SMP computer or by using distributed computing with multiple networked computers. However, adding CPUs does not add performance in a linear fashion. Instead, the performance gain curve slowly tapers off as each additional processor is added. The characteristics of this behavior depend on how the application is designed. For computers with SMP configurations, each additional processor incurs system overhead. After you have upgraded each hardware component to its maximum capacity, you will eventually reach the real limit of the processing capacity of the computer. At that point, the next step is to move to another computer.

Multiprocessor optimization can be achieved by making use of threads.

**Note** More information on multiprocessor optimizations is available at http://msdn.microsoft.com/msdnmag/issues/01/08/Concur/.

# Network Utilizations

Network resources, such as available bandwidth and latency, must be predicted and managed on computers and devices throughout the network.

Optimal network utilization is achieved with cooperation among end nodes, switches, routers, and wide area network (WAN) links through which data must pass. There are tools that help analyze network traffic, provide network statistics and packet information, and thereby better use the network by analyzing areas of congestion.

Quality of Service (QoS), an industry-wide initiative, achieves a more efficient use of network resources by differentiating between data subsets. Windows 2000 implements QoS by including a number of components that can cooperate with one another.

**Notes**

More information on QOS on Windows is available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/qos/qos/qos_start_page.asp.

Network Monitor captures network traffic for display and analysis. More information on Network Monitor is available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/netmon/netmon/network_monitor.asp.

Network Probe is another tool for traffic-level network monitoring and for analysis and visualization. More information on Network Probe is available at http://www.objectplanet.com/probe/.

# Testing and Optimization Tools

The following are some of the commonly used tools for Windows Services for UNIX 3.5.

## *Monitoring Tools*

- **netstat.** Allows you to track the state of the socket ports. This tool is a part of Windows.

## *Testing and Debugging Tools*

- **Electric Fence.** A **malloc** debugger and bounds checker. It uses the virtual memory hardware of your system to detect when software overruns the boundaries of a **malloc** buffer. It also detects any accesses of memory released by **free**.
- **hexdump.** Gives an ASCII, decimal, hexadecimal, and octal dump. The **hexdump** command can be used to display the contents of a binary file or a file that contains unprintable characters.
- **nm.** Used to examine binary files (including libraries, compiled object modules, shared-object files, and stand-alone executables) and to display the contents of those files or the meta information stored in them.
- **ulimit.** Used to display or control the resources available to a process.
- **xev.** Prints contents of X Windows events. It is useful for seeing what causes events to occur and to display the information that they contain.
- **truss.** A run-time system call tracker. It follows the detailed history of system calls. It is useful for narrowing down errors before starting a debugger.
- **pstat.** Displays detailed process information and provides detailed run-time information per process. This tool is a part of Windows Services for UNIX.
- **ps.** Provides run-time information about processes and their interrelationships. This tool is a part of Windows Services for UNIX.
- **Objdump.** Displays information about a binary/object that can be useful for tracking down run-time problems (that is, shared library dependencies). This tool is a part of Windows Services for UNIX.
- **pstruct.** Dumps information about C structures, such as offsets, and actual member sizes. Good for checking alignment problems. This tool is a part of Windows Services for UNIX 3.5.
- **expect.** Scripts user activity and responses interactively to programs.

# Other Commonly Used Tools

This section lists other commonly used tools that are useful in testing and monitoring applications.

## Monitoring Tools

- **Diskmon**. This tool captures all hard disk activity or acts such as a software disk activity light in your system tray. This tool is available for download at http://www.sysinternals.com/ntw2k/freeware/diskmon.shtml.
- **Filemon**. This monitoring tool allows you to view all file system activity in real-time. This tool works on all versions of Windows NT, Windows 2000, Windows 2003, and Windows XP. It also works with the Windows XP 64-bit edition. This tool is available for download at http://www.sysinternals.com/ntw2k/source/filemon.shtml.
- **PMon**. This is a Windows NT GUI/device driver program that monitors process and thread creation and deletion, as well as context swaps if it is running on a multiprocessing or checked kernel. This tool is available for download at http://www.sysinternals.com/ntw2k/freeware/pmon.shtml.
- **Portmon**. You can monitor serial and parallel port activity with this advanced monitoring tool. It knows about all standard serial and parallel IOCTLs and even shows you a portion of the data being sent and received. This tool is available for download at http://www.sysinternals.com/ntw2k/freeware/portmon.shtml.
- **Regmon**. This monitoring tool allows you to view all registry activity in real-time. This tool is available for download at http://www.sysinternals.com/ntw2k/source/regmon.shtml.
- **TCPView**. You can view all the open TCP and UDP endpoints. TCPView even displays the name of the process that owns each endpoint. This tool is available for download at http://www.sysinternals.com/ntw2k/source/tcpview.shtml.
- **Task Manager**. Task Manager provides run-time information on processes. The Task Manager tool is available as part of Windows.

## Testing Tools

- **WinRunner**. Winrunner helps in GUI capture and playback testing for Windows applications. More information on WinRunner is available at http://www.mercury.com/us/products/quality-center/functional-testing/winrunner/.
- **Silktest**. Silktest is an object-oriented software testing tool for Windows applications. More information on Silktest is available at http://www.segue.com/products/functional-regressional-testing/silktest.asp.
- **LoadRunner**. LoadRunner is an automated client/server system testing tool that provides performance testing, load testing, and system tuning for multiuser applications. More information on LoadRunner is available at http://www.mercury.com/us/products/performance-center/loadrunner/.
- **Rational Robot Automated Test**. Rational Robot Automated Test provides automated functional, regression, and smoke tests for e-applications. More information on Rational Robot is available at http://www-306.ibm.com/software/rational/.
- **Microsoft Application Center Test**. Designed to stress test Web servers and analyze performance and scalability problems with Web applications, including Active Server Pages (ASP) and the components they use. It simulates a large group of users by opening multiple connections to the server and rapidly sending HTTP requests. More information on Microsoft Application Center Test is available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/act/htm/actml_main.asp.

## Source Test Tools

- **gdb.** Used to view the current activities inside another program while it executes or to view the state of another program with which you can monitor the processes and threads of an application at the moment of the application's crash.
- **Purify**. Purify is a run-time error and memory leak detector. More information on Purify is available at http://www-306.ibm.com/software/sw-bycategory.

**Tools for Win64:**

- **VTune Performance Analyzer**. Intel VTune Analyzers help locate and remove software performance bottlenecks by collecting, analyzing, and displaying performance data from the system-wide level down to the source level. More information on VTune Performance Analyzer is available at http://www.intel.com/software/products/vtune/.
- **Lint.** A source code (C language) checker. Lint highlights possible problem areas with code successfully compiled by **cc**. Additional information is available at http://www.pdc.kth.se/training/Tutor/Basics/lint/index-frame.html.

# Further Reading

For more information, refer to:

- "Testing Software Patterns" on the MSDN Web site at http://msdn.microsoft.com/practices/guidetype/Guides/default.aspx?pull=/library/en-us/dnpag/html/tsp.asp.

# Index