UNIX Code Migration Guide

UNIX Application Migration Guide

Chapter 8: Preparing for Migration

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

October 2002

Applies to:
Microsoft® Windows®
UNIX applications

The *patterns & practices* team has decided to archive this content to allow us to streamline our latest content offerings on our main site and keep it focused on the newest, most relevant content. However, we will continue to make this content available because it is still of interest to some of our users.
We offer this content as-is, without warranty that it is still technically accurate as some of the material is undoubtedly outdated. Note that the content may contain URLs that were valid when originally published, but now link to sites or pages that no longer exist.

Summary: Chapter 8: Preparing for Migration deals with a more detailed analysis of steps that should be taken prior to code conversion. This chapter covers such areas as standards compliance and script migration and introduces best practices that will ease the actual migration. (15 printed pages)

Contents

Introduction

Prior to migrating your source code, there are a number of pre-migration steps that you should take. This chapter looks at these steps and gives guidance on how you should carry them out.

The subjects covered in this chapter are:

- Preparing the code for migration

These sections show you how you can prepare your code for migration in order to reduce and simplify the changes necessary to migrate the code.

- Migrating scripts

  Most UNIX based applications make extensive use of scripts. Migration of your scripts prior to migrating your code means that you are immediately able to test the application once the developers have compiled it on the target platform.

The material in this chapter applies to migrations to both the Microsoft® Win32® and Microsoft Interix environments.

Preparing the Code for Migration

Programmers often write code for specific operating systems and then port it to others by using a combination of conditional compilation (**#ifdef** preprocessor commands) or wrapper functions. To port platform-specific code, read more about these steps in the following sections:

- Defining the target environment
- Analyzing the application to be migrated
- Making the source POSIX-compliant
- Introducing and rationalizing abstraction layers
- Using Compile-Time Macros

Defining the Target Environment

A UNIX developer preparing to migrate an application to the Microsoft Windows® operating system has the same choices as the developer of a Win32-based application. That is, the Win32 developer must first decide the style of the application. Will it be a text based application or will it be a Windows GUI-based application?

The UNIX developer follows a similar decision process. Will the user invoke the application from the command line and create output files? Will the application be a filter that uses stdin and stdout for input and output? Or will the application need graphical capabilities based on X-Windows or another GUI?

Analyzing the Application to Be Migrated

Before migration, the developer analyzes the application at a high level to determine the most appropriate migration approach. (For more information on this analysis, see Chapter 4, "Assessment and Analysis.") After beginning the migration, the developer can conduct a more detailed analysis to determine how to migrate the code and scripts.

While the code is still on the UNIX platform, the developer can perform a preliminary analysis to identify any areas of the code that should be treated with special care.

ANSI C/C++, FORTRAN applications

Migrated (ported) code needs to run on a different platform. The developer needs to look at the impediments to this, such as:

- Differences in compliance to language standards between the UNIX implementations and Windows; examples include:
  - Lack of compliance with adopted scoping rules involving local variable declaration within **for** loops
  - Standard Template Library (STL) implementation
- Differences between configuration and build tools; examples include:
  - Microsoft Visual Studio® interfaces run only on Microsoft Windows® (lack of UNIX implementations prevent Visual Studio from being a cross-platform IDE/build tool).
  - The **nmake** command does not conform to either the Berkeley **make** or GNU **gmake** commands.
  - UNIX systems use the **autoconf** or **automake** scripts to automatically create makefiles, the Windows equivalent would be the generation of a nmake file from the Visual Studio IDE.
- Lack of support for a compiler, lack of FORTRAN 90/95 support.

The steps in creating the Windows development environment as described in Chapter 7 should help with the differences between configuration and build tools. Verify the conversion to a Visual Studio-based project from the UNIX application's makefile-based build. In particular, be sure that:

- The same flags are passed to the compiler (as a general rule, convert **-**<*anything*> to /<anything>).
- The necessary #defines are the same.
- The header file and library file paths needed are properly set for the Visual Studio environment.

Because the Visual Studio compiler options are very different from their UNIX/GNU equivalents, migration tools that convert between them across platforms are useful.

ANSI C/C++ with cross-platform library applications

In addition to the concerns discussed in the previous section, there are a few additional provisions when running ANSI C/C++ applications with cross-platform library applications:

- Analyze and verify that the cross-platform library has the same set of function definitions, return types, argument types, argument ordering and so on, on the UNIX platforms as on the Windows implementation. They should be the same; otherwise, the library's utility as a cross-platform library is in jeopardy. For example, Rogue Wave Software's Source Pro C++ has a completely compatable set of function definitions, return types, argument types, argument ordering and so on.
- Make sure that anything found with **-I**<*anything*> from the UNIX application linker command line is listed in the Visual Studio Linker options on the **Link** tab under **Additional library path**.

Native UNIX applications ported to Interix

As with ANSI C/C++ and FORTRAN applications, UNIX applications targeted for an Interix port have certain impediments that the developer needs to plan for, such as:

- Interix only provides GNU C/C++ and FORTRAN 77 compilers. If a FORTRAN 90/95 application is to be ported to Interix, third-party compilers need to be considered. For example, Compaq or Intel compilers support FORTRAN 90/95.
- Certain **make** incompatibilities (this is an issue between Berkeley **make** and GNU **make**). Interix **make** is based on Berkeley **make**. Consider porting **gmake** to Interix (see Table 1, below).

Table 1 shows Interix 3.0 ports of compilers and tools that are out of date.

Table 1. Out-of-date Interix 3.0 ports of compilers and tools

Tool/library	Interix version	UNIX version	Comment
autoconf/automake	Not provided "out-of-the-box"		Can be ported by developer from GNU Web site.
gmake (GNU)	Not provided "out-of-the-box"	3.79.1	Can be ported by developer from GNU Web site. Recommended for an application that makes significant use of **gmake** (for example, Apache).
gcc	cygnus-2.7.2	2.96	Name spaces are mostly broken in this version of the C++ compiler.
tar		1.13.19	
ar	cygnus-2.8.1	2.10.91	
make (Berkeley)	R5	R6	Usually not a problem.
X11			
Motif	1.	1.2	

For example, Martin Walenta's Trading Toolkit requires:

- **gcc**: version 2.95.2
- **tar**: version 1.12
- **ar**: version 2.10.90

The application should be analyzed to determine its level of dependency on a specific version of a tool (beyond the scope here). A decision should also be made to modify the application's dependency or port the version of the tool required before porting the application.

Native UNIX applications rewritten to Win32

If the UNIX application has a dependency on UNIX shell scripts (for example, **ksh** or **csh**), consider including Interix in the ported environment to support this functional requirement.

Making the Source POSIX-Compliant

Writing standard POSIX code is the best strategy for producing migratable code because all UNIX platforms are compliant with POSIX.1 and POSIX.2 standards.

The following sections look at some key areas where you should make your code POSIX-compliant to ease the migration of the code.

Strictly conforming POSIX.1 applications

A strictly conforming POSIX.1 application requires only the facilities described in the POSIX.1 standard and applicable language standards. A strictly conforming POSIX.1 application:

- Does not rely on any behavior described in ISO/IEC 9945-1 as unspecified or implementation-defined.
- Uses only those facilities described in the standard. However, because the behavior of some of those facilities varies across implementations, such an application might need modification to run on different platforms.

Applications at this level should be able to move across implementations with just a recompilation.

For more information about the POSIX.1 programming environment, see these resources:

- Lewine, Donald. *POSIX Programmer's Guide*. O'Reilly & Associates, 1991
- Stevens, W. Richard. *Advanced Programming in the UNIX Environment*. Addison-Wesley, 1992
- Zlotnick, Fred. *The POSIX.1 Standard: A Programmer's Guide*. Benjamin/Cummings, 1991

System information

Different systems provide different ways of obtaining information about the system. On BSD systems, the **sysctl** interface provides access to system information. On System V systems, the **sysinfo** call provides system information. However, these interfaces are nonstandard, and their implementation varies between UNIX implementations.

For example, on Solaris **sysinfo** gets and sets system information strings, such as:

```
#include <sys/systeminfo.h>
long sysinfo(int command, char *buf, long count);
```

On Linux, **sysinfo** returns information on overall system statistics, such as:

```
#include <sys/sysinfo.h>
int sysinfo(struct sysinfo *info);
```

Developers can access system information by using POSIX routines. The POSIX system information routines return strings, paths, and numeric values, including two-valued Boolean conditions. The header file **limits.h** contains macros that define system limits, as shown in Table 2.

Table 2: Macros that define systems limits

Macro	Description
confstr	Retrieves string values. The only portable POSIX.2-compliant value for name is _CS_PATH, a value for PATH guaranteed to find the standard utilities. Prototype is: size_t confstr (int *name*, char **buf, size_t len*).
fpathconf, pathconf	Retrieves configurable path variables, such as the maximum size of a file name or the maximum link count. Used in System V programming. Prototypes are: long pathconf (const char **path*, int *name*) long fpathconf (int *fd*, int *name*).
sysconf	Retrieves system information that can provide answers to questions such as: Is job control available? Are POSIX

options supported? What are the limits for **bc**? What is the maximum number of bytes allowed for an argument to **exec**? Prototype is:
long sysconf (int *name*).

> **Note** The Interix Software Development Kit (SDK) also provides both **uname** and **gethostbyname**.

The following code example uses the POSIX **sysconf** function:

```
/* syssample.C: This program illustrates usage of sysconf functions*/
#include <stdio.h>
#include <unistd.h>
int main( void )
{
/* retrieve the system information */
   long sinfo;
   sinfo = sysconf(_SC_VERSION);
   printf("Version supported: %d\n",sinfo);
   sinfo = sysconf(_SC_LINE_MAX);
   printf("Maximum line length: %d\n",sinfo);
   return;
}
```

The following output is obtained from the program on Interix:

```
% ./syssample
Version supported: 199009
Maximum line length: 2048
```

Advisory file locking

Table 3 shows the traditional application programming interfaces (APIs) for file locking.

Table 3. APIs for file locking

API	Standard
fcntl	POSIX. However, POSIX.1 specifies only the F_DUPFD, F_GETFD, F_SETFD, F_GETFL, F_SETFL, F_GETLK, F_SETLK and F_SETLKW operations.
lockf	System V. (Built on fcntl.)
flock	BSD. (Built on fcntl.)

> **Note** Any lock created by Interix is advisory; that is, not enforced by the operating system. These locks have no effect in the Win32 environment. Advisory locks allow cooperating processes to perform consistent operations on files, but consistency is not guaranteed (that is, processes may still access files without using advisory locks, possibly resulting in inconsistencies).

Current working directory

Legacy code might use **getwd** to determine the current working directory. It is better to use the POSIX

getcwd interface, which includes a *size* argument to prevent buffer overflows on the returned directory pathname.

Error messages

System error messages are available in POSIX through the **strerror** and **perror** calls. Traditional systems expose the underlying **sys_errlist** array and store the number of array elements in the variable **sys_nerr**.

The gets function

The **gets** function is known to be a security risk. This is because the length of an input line can overflow the size of the buffer, resulting in indeterminate behavior, or a deliberate attempt to deliver code to the application for execution. For this reason, the use of **gets** is strongly discouraged, and it is strongly recommended that **fgets** is used. This Interix implementation of **gets** prints out the following warning whenever the **gets** function is called:

```
warning: this program uses gets(), which is unsafe.
```

> **Note** You can disable the warning by setting the environment variable DISABLE_GETS_WARNING.

Terminal I/O

The Interix Software Development Kit (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. When you create a portable application, however, you should take into consideration that VMIN and VTIME can be identical to VEOF and VEOL on a POSIX.1 system.

The new functions shown in Table 4 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 4. New functions that replace terminal I/O ioctl calls

Function	Description
tcgetattr()	Fetches attributes (termios structure)
tcsetattr()	Sets attributes (termios structure)
cfgetispeed()	Gets input speed
cfgetospeed()	Gets output speed
cfsetispeed()	Sets input speed
cfsetospeed()	Sets output speed
Tcdrain()	Waits for all output to be transmitted
tcflow()	Suspends transmit or receive
Tcflush()	Flushes pending I/O
tcsendbreak()	Sends BREAK character
tcgetpgrp()	Gets foreground process group identifier (ID)
tcsetpgrp()	Sets foreground process group ID

If you need to get the window size, the TIOCGWINSZ command for **ioctl** and the **winsize** structure are both supported.

The **TERMIO** terminal hardware structure in the System V and the **STTY** terminal hardware structure in BSD have been replaced in POSIX with the **TERMIOS** structure and a new set of access calls.

The **TERMIOS** model is very similar to the System V model. Two modes exist: canonical and noncanonical. Canonical input is line-based, like BSD cooked mode. Noncanonical mode 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 follows:

```
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 */
};
```

Signals

The POSIX.1 committee introduced new signal semantics because of problems with BSD and System V signal implementations.

When the System V3 **signal** function catches a signal, the action associated with the signal is reset to the default. In BSD 4.3, it is not reset. In the International Standards Organization/American National Standards Institute (ISO/ANSI) C standard, the **signal** function either resets the default or does an implementation-defined blocking of the signal.

The POSIX **sigaction** call does not reset the default if the handler returns normally. The Interix SDK follows the POSIX signal semantics.

Because an Interix SDK process has a signal mask, it can block any (or all) of the signals from arriving, except for SIGKILL or SIGSTOP, which cannot be caught or ignored. 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.

In code that uses the **signal** function, the signal is still masked and remains masked until the mask is clear.

> **Note** This can be a significant problem if the code uses a **longjmp** call from the handler routine. Using the **sigaction** call directly with **siglongjmp** call corrects some unexpected behaviors.

Time

For portability, use the POSIX.1 **utime** function instead of **utimes**, which is no longer supported. The arguments and semantics for these functions are slightly different.

The syntax for the **utimes** function is as follows:

```
int utimes (const char * path, const struct timeval * times):
```

The **utime** function uses the second argument, *times*, which is a **struct utimbuf**:

```
int utime(const char * path, const struct utimbuf * times);
```

The **utimes** function succeeds when used with writeable files, but **utime** used with a non-null argument does not succeed unless the user executing the process owns the file.

The **utimbuf** structure is defined in utime.h. It contains the following two members:

```
time_t actime; /* access time */
time_t modtime; /* modification time */
```

Processes and threads

The **setpgrp** interface is obsolete. In UNIX implementations, **setpgrp** functionality is duplicated by other functions. Both BSD and System V have a **setpgrp** call, but each has different semantics and behaviors.

The System V **setpgrp** call changes the caller's process group ID to its own process ID, and releases the controlling terminal of the calling process. The System V call takes no arguments.

The BSD **setpgrp** call detaches a process from its process group, but does not change the controlling terminal. The BSD call takes two arguments, a process ID and a process group ID.

The System V **setpgrp** call is properly replaced by the POSIX **setsid** call. When a process calls **setsid** successfully, it creates a new session, which in turn contains a new process group that contains one process—the calling process. The calling process is now the session and process group leader. The process also disposes of its controlling terminal. (In fact, the **setsid** call is used to dispose of a controlling terminal.)

The BSD **setpgrp** call is replaced by the POSIX **setpgid** call, which has identical semantics.

Introducing and Rationalizing Abstraction Layers

Writing a custom library of functions for each platform can help to abstract nonportable code. In this case, the application always calls the private version of the function, which is linked to a platform-specific library. Creating and linking the platform-specific libraries entails a great deal of work, but it could be the appropriate method for a large body of source code.

For example, different libraries (such as func_Ix.lib and func_Ux.lib) can implement the **func1** custom library function (instead of a system call) for the application that is to be ported. Then, when compiling on a particular system, the appropriate library is linked.

Using Compile-Time Macros

Many applications use **#ifdef** statements to isolate platform-specific sections of code. Some older code

written using **#ifdef** statements is based on assumptions about the platform that are no longer valid. For example, code built around **#ifdef BSD** usually tries to include sgtty.h rather than termios.h. Labeling blocks of code with the platform name often amounts to aiming for a moving target. For example, BSD 4.4 has some different APIs than BSD 4.3, but they are both BSD.

POSIX specifies its own set of compile-time macros and manifest constants, not defined in either System V or BSD systems. These macros and constants are unique to POSIX and are defined in the specified include files.

Migrating Scripts

This section describes how to port UNIX shell scripts to the Interix and Windows environments. The steps in the process are described in more detail below:

- Evaluating the script migration tasks
- Planning for platform differences
- Considering source and target environments

Scripts fall into 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/C++. A scripting language such as Perl handles most platform specifics. However, the original developer might have used easier or faster platform-specific features, or simply might not have taken cross-platform compatibility into consideration.

The choice of porting approach depends on the source script type and whether the target environment is Windows only, Windows plus Interix or uses CGI scripts.

With an Interix installation, a large number of the usual UNIX commands are available. Because Interix provides both the Korn and Tenex C Shells, many UNIX shell scripts run under Interix without conversion. For more information, see the Microsoft Word document, Porting Shell Scripts.

In the Windows-only environment, a solution is to write all common scripts in Perl because there are several versions of Perl available. If software is to be maintained on both UNIX and Windows-based systems, writing all-new scripts in Perl and even converting some existing shell scripts to Perl is a good strategy.

Evaluating the Script Migration Tasks

Before script migration begins, all required tasks need to be considered. To identify script migration tasks, consider the following questions:

- Does the script rely on the syntax of the shell?
- Does the script use substantial external programs?
- Does the script use extensions that rely on third-party libraries?
- Does the script use or rely on nonportable concepts for essential functionality?
- Can a quick port be done now, with a rewrite later?

- Does the developer understand enough of the original code to quickly locate the issues, and then make the changes necessary to port to a new platform?

By answering these questions, script migration tasks can be evaluated and defined. Redesigning and rewriting portions of the application might be easier than porting because it is more efficient to take advantage of native features.

Planning for Fundamental Platform Differences

When porting scripts, the code needs to address some inevitable fundamental differences between the platforms. The areas listed below, which are described in more detail later in this section, are often sources of script migration issues:

- File system interaction and I/O
- Environment variables
- Shell and console handling
- Interprocess communication
- Process manipulation
- Device and network programming
- User interfaces
- Localization and internationalization

File system interaction

UNIX and Windows-based systems interact differently with the file system. The UNIX and Interix path separator is a forward slash (/); Windows uses the backslash (\). The root of UNIX and Interix files is represented by the forward slash (/), but Windows uses locally mounted drives ([**A-Z**]**:**\) and network-accessible drives using the Universal Naming Convention (*ServerName**SharePoint**Dir*\).

The first things you should correct in any code to be migrated are any hard-coded file paths. These paths are commonly used to find initialization or configuration files (that is, to set up environment variables or application paths). One common mistake when doing initial porting work is to refer to a Windows-based file in native form. The problem is that the backslash (\) is also the common escape character. Because of this, the path **"C:\dir\text.txt"** is translated as **"C:dir ext.txt"**. (The space is a single tab character.)

In most cases, Windows can also handle the forward slash (/) as a path separator. However, when building cross-platform paths, scripting language compilers can misinterpret even correctly used file path separators or methods.

Unlike UNIX, Windows Win32 file systems are not case-sensitive. They may preserve the case of files names, but the same directory cannot contain two different files named (for example, file.txt and FILE.txt). Windows also does not allow users to create a file with the same name as the directory in which it is created.

Keep in mind when hard coding paths in a script that certain Windows directories change depending on native language. For example, the directory C:\Program Files\ in English is C:\Programme\ in the German version of Windows.

The exact names for paths and other information that may be critical in porting your code are often found in the Windows registry. For example, the correct path for the Program Files directory can be

found in
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\ProgramFilesDir.

The registry is a central database of information about your Windows system. It must be referred to often when other platform-independent methods are not available. Use the **regedit** command to peruse the Windows registry and get a feel for the basic structure. Some of the information stored in the registry is also available by using language APIs, which are safer to use.

Environment variables

Both Windows and UNIX use environment variables. Although Windows maintains an environment array, its contents are not similar to UNIX. The Windows environment array is not case-sensitive, so the environment variables *PATH*, *path* and *PaTh* all refer to the same item. The PATH variable is similar in purpose across platforms (for example, shells search the directories specified in the *PATH* environment variable for executables and scripts), but Windows uses a semicolon (**;**) as a separator, whereas UNIX and Interix use a colon (**:**). Fortunately, compiled languages usually have features that handle the differences in usage of the *PATH* variable.

Commonly used UNIX environment variables are *HOME, PATH, USER* and *TEMP*. Windows also has the *PATH* and *TEMP* variables, and sometimes has the others. To determine which environment variables are used in a Windows installation, use the abstractions in a compiled language or look in the Windows registry. Alternative, you can use the Windows tool described below.

To see the full contents of the environment

1. Right-click **My Computer** and then select **Properties**.
2. In the **System Properties** dialog box, select the **Advanced** tab.
3. Under **Environment Variables**, click the **Environment Variables** button.
4. In the **Environment Variables** dialog box, view and modify the environment.

Note that Windows has separate User and System environments. Administrator rights are required to modify the System environment.

Scripts commonly require a temporary data file, which is usually hard-coded to reside in /tmp on UNIX. On Windows and UNIX, use the TEMP environment variable instead to refer to an acceptable temporary file directory. Some scripts also rely on environment variables beginning with LC_, which indicate the locale information for that system.

Do not expect files to always be the same at the binary level. For example, Windows uses CRLF (carriage return/linefeed or characters \015\012) at the end of a line; UNIX uses LF only; Interix supports both, and provides a *flip* command-line utility to convert between the two formats. Script environments provide methods for handling this transparently. Another nuance is that **^Z** (character \032) represents the end-of-file character. A UNIX script with this character embedded in code might ignore it, and Windows might stop reading the file at that point. Interix behavior can vary depending on the utility or program, for example, the *flip* utility removes the last control-Z in a file, but leaves embedded control-Z characters unchanged when converting from Windows to POSIX format.

Shell and console

The shell is found on all UNIX desktops. Interix includes both the Korn and C shells. Windows provides

a command shell as well. Windows 2000 stores the path to the shell in the COMSPEC *variable o*f the environment array. Developers interact with the command shell during testing, but it can interfere or act in unexpected ways during the normal operation of a script. Some languages can run without any attachment or specific connection to a terminal. Refer to the language specifics for how to make the console behave as required.

Some scripts call the shell to reuse existing commands, such as **cat**, **ls**, **sendmail**, **date**, and **grep**. Relying on the shell is not recommended because it not only wastes processing power by creating external process execution overhead, but it is also highly nonportable. To avoid portability issues, it is better to rely on the methods that the language provides.

For example, the following example might not be portable:

```
set date [exec date "+%D %H:%M"]
```

The following example is portable:

```
set date [clock format [clock sec's] -format "%m/%d/%y %H:%M"]
```

Also note that invoking commands from the shell automatically uses wildcard expansion (usually refered to as *globbing*). When the script relies on globbing, you should use the language methods for file globbing to expand file names. In cases where it is unavoidable to call the shell, it is important to note that the Windows command shell has different native commands and quoting rules.

Process and thread execution

The script might need to deal with process manipulation, especially if external system calls are unavoidable. In a language that supports process manipulation, the features are usually portable to Windows 2000. However, it is still necessary to evaluate all uses of process manipulation to ensure that the application code is manipulating the correct Windows processes.

It is common in UNIX to manage processes by passing signals, especially for daemon processes and system administration tasks. Signal handling, where handled by the language, is similar to process manipulation. Some uses of signal handling are portable from UNIX to Windows 2000, but not all signals are relevant. Windows uses an event-passing model. A UNIX daemon process ported to Windows needs to respond to these events. When porting a UNIX daemon on Windows, it is necessary to create a Windows service that provides essentially the same functionality.

It is important to note that a **fork** command can have different behavior on Windows depending on the language. If the **fork** command is used in a Web application, it is highly recommended to look at alternative techniques for achieving the same result on Windows. The best solution is to switch to using threads.

Device and network programming

Many applications built today use a client/server model, or must follow network or interprocess communications requirements, such as HTTP, TCP/IP, and UDP. Scripting languages provide varying levels of abstraction over the standard system mechanism for communicating with files and sockets. Because some are more portable than others, it is important to examine socket handling when porting code. Methods for interprocess communication outside of socket programming or communicating through a pipe are to be avoided because they are normally nonportable. A well-known remote

procedure call (RPC) mechanism that works well across platforms and fits well into Web server applications is Simple Object Access Protocol (SOAP), which most scripting languages already support.

An application that communicates with the serial port or other system device can use the same protocol for interacting with the device, but often must address the device differently. For example, a serial device on UNIX can be addressed as the special file /dev/ttya. On Windows, it is addressed as COM1:.

User interfaces

Many scripting languages have access to one or more graphical user interface (GUI) toolkits. If the language used in script has a GUI toolkit, it is important to determine the portability of that toolkit across platforms. Interix, the UNIX portability layer, provides a port of the curses terminal user interface library.

Tk is a GUI toolkit common to Tcl, Perl and Python. It is fully cross-platform compatible between UNIX and Windows. Some of the finer points of cursor and font handling can vary between these systems because of underlying operating system differences.

Considering the Target Environments

This section describes the major scripting environments found in the target environments. For each there are some differences with the scripting environment under UNIX. These are described so that they can be addressed during the migration.

Porting UNIX shell scripts to Interix

When porting a shell script from an open-system implementation of UNIX (such as System V4 or BSD) to Interix, there are only two significant differences. First, by default Interix stores binaries in one of three directories: /bin, /usr/contrib and /usr/local/bin. For example, Perl is installed in one of those directories. Second, even though 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 adding symbolic links can handle most situations. For example, /usr/ucb can be linked to /usr/contrib/bin and /usr/local/bin can be linked to /usr/contrib/bin.

Additional considerations are as follows:

- First, port scripts that set up either local or environment variables.
- The Interix C shell initialization process executes two files before the **.cshrc** and **.login** files in the user's home directory. These are /etc/csh.cshrc and /etc/csh.login.
- Be aware of the current limits of Interix shell parameters so that appropriate action can be taken. 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(), setgrent() APIs
- Win32 ADSI scripts
- Win32 **net user** commands

CGI script migration

The Common Gateway Interface (CGI) protocol is the standard interface used by Web servers to execute programs and scripts that handle dynamic content. Any language can be used as a CGI language if it supports reading and writing STDOUT and STDIN console handles, and chances are that many existing scripts are CGI-based. In recent years, many Web server plug-ins have been written for scripting languages to work around performance limitations in CGI, although using these plug-ins sometimes requires minor changes to the CGI script itself. Apache has direct language plug-ins for Perl (mod_perl), PHP (mod_php) and Tcl (mod_tcl). Through the Internet Server API (ISAPI), Microsoft Internet Information Server (IIS) has a direct language plug-in for Perl called PerlEx.

Normally, CGI portability is not an issue because CGI is a standardized interface available under all major Web servers.