UNIX Code Migration Guide

UNIX Application Migration Guide

Chapter 4: Assessment and Analysis

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offer this content as-is, without warranty that it is still technically accurate as some of the material is undoubtedly outdated.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 4: Assessment and Analysis discusses application assessment. This is a process where the application or applications to be migrated are analyzed from both a business and technical perspective to form a migration strategy to either Win32 or Interix. (42 printed pages)

Contents

Introduction

The process of migrating UNIX software to Microsoft® Windows® 2000 is sometimes perceived to be a long, expensive undertaking because of incompatibilities between the two operating systems. However, application migration from UNIX to Windows is feasible, and this chapter will help guide you through the process.

To migrate an application from UNIX to Windows, you must consider the platform differences, the differences between the application's implementation on each platform, and the features of the deployment and management tools available for each operating environment. Therefore, any application migration from UNIX to Windows requires a thorough assessment of the application to be migrated, followed by an analysis to determine the most appropriate migration approach.

This chapter discusses the assessment and analysis aspects of the migration process in detail, including:

- Gathering data
- Evaluating the migration objectives
- Evaluating the application
- Defining the migration strategy

In addition, this chapter examines the specific migration strategies for various application architectures, including workstation-based applications and server-based applications.

Planning

The results of the assessment and analysis phase are used to develop the various migration plans—for example, development, testing, training, deployment and support. These results also act as input to risk assessment activities, which are a mainstay of both planning and testing. The assessment and analysis phase therefore requires planning, which includes:

- Scoping the assessment, the analysis and the migration
- Identifying data sources
- Arranging interviews and workshops

The assessment and analysis phase follows the law of diminishing returns; that is, the value that it provides diminishes relative to effort. It is therefore important to size these activities according to the size of the migration.

Assumptions

This chapter focuses on software that is written in the C, C++ and Fortran compiled languages, in addition to scripting languages such as Perl and various UNIX shells (for example, Bourne, Korn, C and Bash). This chapter assumes that the application's UNIX-based source code is available in either an archive or source control system, such as the following:

- Concurrent Version System (CVS)
- Revision Control System(RCS)
- Program Version Control System (PVCS)

Some migrations will result in the subsequent removal of the UNIX environment. However, many environments will need Windows to integrate (and coexist) with UNIX, and to run the same migrated application software that is currently running on the UNIX platform or platforms. The need for coexistence is an important factor in software migrations. A migration philosophy that takes into account cross-platform software portability allows Windows to be accepted in organizations that require the application to remain on the existing UNIX platform or platforms.

Time to market is just as important for internal applications as it is for independent software vendor

(ISV) software. Applications often represent competitive advantages to their owners (for example, augmentations to computer-aided engineering analysis or product data management packages). A software migration philosophy must accommodate this urgency for the migration project to be successful based on time and resource constraints.

Gathering Data

As part of the plan for assessing an application for migration from UNIX to Windows, you must determine the application context, structure, development environment and usage model.

It is also important to understand the target development environment for the application migration. Examples include migration of both deployment and development environments to Windows and Microsoft Win32®; continued development on UNIX with Windows as a cross-platform migration; and porting to a Portable Operating System Interface for UNIX (POSIX) environment on Windows, such as Microsoft Interix.

You need to address these considerations, as well as the actual application source code migration and possibly the migration of associated test programs and/or scripts. The point at this stage is not to conduct an analysis of the different elements involved in the migration, but rather to understand them at a high level and therefore make the preliminary decisions on how the migration should proceed.

Application Context

The application context consists of all the elements of information technology upon which the application depends. The migration must keep existing dependencies or in some way replace or remove them.

The application may have had a long history—it may not have even begun as a UNIX application. Determining information about the application's history (for example, whether it was already ported to various UNIX environments may provide useful insights into how the application should be migrated to Windows.

UNIX platform support

There is a wide variety of UNIX platforms, each of which has a different bearing on how migration can proceed.

Standardization efforts have existed throughout the history of UNIX, ever since its first incarnation at Bell Laboratories in the 1970s. Its code base was subsequently split when an early version of UNIX was adopted and enhanced at the University of California, Berkeley, to yield the Berkeley Standard Distribution (BSD). Today, UNIX is a trademark administered by the Open Group, and refers to an operating system that conforms to the X/Open specification XPG4.2. This specification defines the name of, interfaces to and behaviors of all UNIX operating system functions.

XPG4.2 aligns with UNIX95—properly called the Single UNIX Spec (SUS) - SPEC1170, the project name that led to specification—and is largely a superset of earlier series of specifications. Approximately 80 percent of the SUS is made up of accredited International Organization for Standardization (ISO), POSIX and American National Standards Institute (ANSI) C standards.

The current X/Open specification, UNIX98, aligns with SUS V2 (XPG5). This specification added

POSIX.1b-1993 real-time and POSIX.1c-1996 threads and some additional thread interfaces, along with the traditional UNIX shared object interfaces (such as dlopen and dlsym). Only a few systems currently conform to this specification.

Many UNIX-based systems are available, such as Sun's Solaris, Hewlett Packard's HP-UX, FreeBSD, Linux and the Microsoft Interix subsystem. Determining an application's standards support results in a better chance of understanding the application's portability to either Win32 or Interix. For example, if an application runs on a version of UNIX that supports the same standards as Interix, it is likely that a direct port using Interix would be more cost-effective than rewriting some of the code for the Win32 platform.

One of the first things you need to do in your assessment is determine which UNIX standards the application is based on. You can determine this either by knowing what the application claims to support (for example, through its original documentation) or by using analysis tools.

Standards organizations often provide certification brands alongside the standards. These are used as a rubber stamp by software vendors in much the same way as the "Windows Powered" or "Designed for Microsoft Windows" brands. You should therefore be aware of both the standards and the brands associated with them."

The application may support any of the following standards:

- ANSI C: X3.159-1989, ISO/IEC 9899-1990
- BSD 4.2-4.4 (Berkeley UNIX)
- GNU
- Institute of Electrical and Electronics Engineers (IEEE) 1003.1-1990 part 1, also known as ISO/IEC 9945-1:1990 (POSIX.1)
- IEEE Std 1003.1b-1993, also known as ISO/IEC 9945-1:1993, POSIX1b (formerly POSIX.4)
- IEEE 1003.2-1993 part 2, also known as ISO/IEC 9945-2:1993 (POSIX.2)
- System V Interface Definition (SVID), SVID2, SVID3
- System V4 UNIX
- Version 7, the ancestral UNIX from Bell Laboratories
- XPG2, XPG3
- XPG4 (superset of POSIX—XPG Base branding)
- XPG4v2 (UNIX95 branding)
- XPG5 (UNIX98 branding)

UNIX standards and Interix

The main purpose of Interix is to allow the direct port of an application to Windows, so Interix often supports a superset of standards or behaviors. Interix mainly uses POSIX.1/.2/ISO C. When a piece of code isn't described in those specifications, Interix looks to the XPG4 specification and then to various UNIX implementations depending on the interface (for example Solaris, BSDI or Red Hat Linux).

The original core standards—POSIX.1-1990, ISO C (1990) and POSIX.2/2a-1992—continue to be, important. For example, there have been two ways to determine the pseudo-terminal name to use, one BSD-based and the other SVID-based. Interix supports both; all of the signal interfaces are supported by the POSIX.1 reliable signal model. Interix also supports many interfaces that are not part of the SUS or a POSIX standard, but are nonetheless in constant use—for example, Open Network Computing (ONC) remote procedure call (RPC) and X11 Release 5 (R5).

If the application supports XPG4, there is a high probability that the application can be ported to Interix. It is more important, however, to determine the following:

- Does the application have significant dependence on UNIX application programming interfaces (APIs)?
- Does Interix support the greatest majority of the application's required APIs?

Infrastructure dependencies

The infrastructure upon which the application depends includes client and server hardware, network elements and peripheral devices such as external tape drives and printers.

The migration of the application can be to a completely new infrastructure configured to support Windows. If this is the case, there may be migration issues in the application software. For example, if you are migrating from a 32-bit computer architecture to another architecture, you may face byte ordering issues. Some architectures number bytes in a binary word from left to right, which is referred to as *big-endian*. Other architectures number the bytes in a binary word from right to left, which is referred to as *little-endian*. A notable computer architecture that uses big-endian byte ordering is Sun's Sparc. Intel architecture uses little-endian byte ordering, as does the Compaq Alpha processor.

Using the big-endian and little-endian methods the number 0x12345678 would be stored as shown in Table 1.

Table 1 Big-endian and little-endian byte ordering example

Method	Byte 0	Byte 1	Byte 2	Byte 3
Big-endian	12	34	56	78
Little-endian	78	56	34	12

The following code snippet illustrates how the use of the big-endian and little-endian methods can affect the compatibility of applications.

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

int main()
{
  int buf;
  int in;
  int nread;

  in = open("file.in", O_RDONLY);
  nread = read(in, (int *) &buf, sizeof(buf));
  printf("First Integer in file.in = %x\n", buf);
  exit(0);
}
```

In the preceding code, if the first integer word stored in the file.in file on a big-endian computer was the hexadecimal number 0x12345678, the resulting output on that computer would be as follows:

```
% ./test
First Integer in file.in = 12345678
```

```
%
```

If the file.in file were read by the same program running on a little-endian computer, the resulting output would be as follows:

```
% ./test
First Integer in file.in = 78563412
%
```

Because of the difference in output, you would need to rewrite the program so that it can read integers from a file based on the endian method that the computer uses.

The other computer hardware dependencies to check for are low-level hardware API calls or calls to specific devices, such as those shown in Table 2.

Table 2. Hardware API calls

Hardware dependency	Check for these calls or devices:
Memory allocation	**alloca()**
Cache management	**cacheflush()** (this is MIPS processor-specific)
Port input	**inb(), inw(), inl(), insb(), insw(), insl()**
Port output	**outb(), outw(), outl(), outsb(), outsw(), outsl()**
Paused input/output (I/O)	**outb_p(), outw_p(), outl_p(), inb_p(), inw_p(), inl_p()**
Input device management	joystick, keyboard, mouse
Display management	video graphics adapter (VGA)

The presence of such API calls in the application code requires that you rewrite the code elements to minimize the hardware dependencies (through the inclusion of alternate, low-level routines) or that you replace the API calls. You must then decide whether the code is to be portable between the two platforms, whether alternate code bases are to exist, or whether the code for the original platform is to be made obsolete.

Applications can have dependencies on peripheral devices as well as computer hardware. Peripherals include not only printers and storage devices, but also firewalls (for example, to set up a virtual private network tunnel to another application) and domain-specific communications devices (for example, to connect to teller computers or telemetry equipment).

For the migration to be successful, you must know what devices are in use by the original application. These devices may have their own dependencies; therefore, it is appropriate to create an inventory of all of the devices concerned, and include the following information for each device:

- Device name, type and purpose
- Software versions that the device runs
- Mechanisms that the application uses to access the device
- Dependencies on other devices and software packages
- Whether the migrated application will use the device

The information that such an inventory provides will help you identify potential issues—for example, whether Windows can support each device or whether you should seek an alternative.

Third-party libraries

Most applications use externally sourced libraries of code or application software extensively, thus enabling common functionality to be reused rather than redeveloped. In particular, many package suppliers (including database suppliers) provide code libraries to enable API-level access to their own products.

Use of a library implies access to functionality that the target platform must support. There are a number of ways to access a third-party library:

- A version of the library may exist for the target platform.
- The library can be accessed from the original platform by means of RPC (this may be the case for databases).
- Code may need to be developed to replace or render unnecessary the library access.
- Code can be rewritten to support alternative services provided by the target platform.

You can determine whether the UNIX application is using third-party libraries by using the UNIX **grep** command on the Makefile for the application:

```
% grep -e "-l" -e "/lib" Makefile
```

Using the **grep** command in this way yields output that may look like the following example:

```
/apps/oracle/lib/libclient.a
/apps/oracle/lib/libcommon.a
/apps/oracle/lib/libgeneric.a
/apps/oracle/lib/libsqlnet.a
/apps/oracle/lib/libc3v6.a
/apps/oracle/lib/libcore3.a
/usr/lib/X11R6
/usr/lib/Motif1.2_R6
/xrt/lib/libxrtfield.a
-lXm
-lXt
-lX11
```

> **Note** The output in the preceding example was generated from a real application's Makefile. It has been edited to remove application-specific references.
>
> Some applications make use of multiple Makefiles, in which case the UNIX **find** command can be used in conjunction with **grep** to traverse the source hierarchy.
>
> For example: % find. -name "[M | m]akefile" | xargs grep -e "-l" -e "lib.*[.].*[a|so]"

Analyzing the preceding output leads to the conclusion that the application is using the following third-party libraries:

- Oracle Call Interface (OCI) libraries
- X11 Release 6 (R6)
- Motif 1.2 for X11R6
- X11 toolkit library
- X11 miscellaneous utilities library

- Xrt widget library for Motif

If you have access to the binary libraries or executable files for the application, you can also run the ldd utility (or a similar utility) to obtain an inventory of the application's shared library dependencies (that is, libraries with a .so extension instead of the .a extension for static library archives).

For example, the dependencies of the test program shown in the earlier code snippet and output samples that illustrated the big-endian and little-endian methods—running on a Linux 7.1 system—yields the following output:

```
$ ldd test
    libc.so.6 => /lib/i686/libc.so.6 (0x40021000)
    /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
```

The preceding output shows that the simple test program depends on the standard C shared library major version 6 (libc.so.6).

Some libraries can be cross-platform libraries that provide a platform-independent API to functionality such as database access, graphical user interface (GUI) or networking. Such libraries encapsulate the different APIs of different operating systems, providing an application with a common set of APIs for all operating systems. Example cross-platform libraries include:

- Rogue Wave's SourcePro C++
- Trolltech's Qt product
- INT's Carnac and GeoToolkit GUI toolkits

As with hardware dependencies, it is useful to produce an inventory of library dependencies as you find them. The inventory can document the following:

- The name, type and context of the library
- The purpose that the dependency serves
- The impact that the dependency has on the migration
- A recommendation for how to treat the dependency

Interoperability with other applications

In addition to libraries that support the operation of software, there are other applications and software packages that are either required by the application or depend on the application themselves. These applications and software packages can include database management systems, messaging and groupware applications, content management and Web server software, accounting software, resource management software and sales automation software.

Migrating one application can have a number of potential impacts on such external software, including:

- How data will be shared at a low level (through files and the passing of information) and at a the user interface level (through the clipboard and the drag-and-drop feature)
- How the applications will intercommunicate
    - How users will access the applications; that is, through a single interface or portal or through multiple interfaces
    - How security (authentication and access controls) will be managed between applications

A first step toward resolving these issues is to gain an understanding of what other programs exist and the dependencies that exist between them. Once again, the starting point is an inventory, to document each package and how it can affect the migration.

External software can have its own dependencies on libraries or hardware, which may further influence the migration. For example, a dependent software package already running on a Windows operating system may require an older version of Windows than the one intended for the migration. This can result in a decision to upgrade the external software package to a more recent version so that the two applications can reside on the same server; alternatively, two servers may be run in parallel, each running a different version of Windows. Each option offers trade-offs that are best explored though a cost-benefit analysis, which is discussed later in this chapter.

Application Structure

The next step toward the goal of migrating an application from UNIX to Windows is to gain a high-level view of the application to determine the main building blocks of the application and the application's scope (that is, the sum of the building blocks). This high-level view also helps establish the boundaries of the application.

These building blocks will enable the migration itself to be scoped, providing definitions of what parts of the application need to be migrated. After the objectives for the migration have been decided, each building block can be analyzed in more detail to determine exactly how it should be migrated.

An appropriate way to view the building blocks of an application is in terms of its layers, as described in the sections that follow.

Application types

Most commonly used applications are either workstation-based or server-based. Workstation-based applications run at the UNIX workstation (desktop) computer and access data that resides on network file shares or database servers. Workstation-based applications have the following architectural characteristics:

- They can be single-process (monolithic) applications or multiple-process applications.
- They use character-based user interfaces or GUI-based (for example, X Windows or OpenGL) user interfaces.
- They access a file server (through the network file system [NFS]) or a database server for data resources.
- They access a compute server for compute-intensive services (for example, finite element models for structural analysis).

The second application type—server-based applications—run on a server and are accessed from a UNIX workstation (desktop) client through telnet and remote shells (character-based applications), X Windows server (GUI-based applications), interprocess communication (IPC) (client-server applications) and other means. Server-based applications can be described as one of the following UNIX software architectures:

- **Compute servers** that use a messaging mechanism, such as RPC or sockets, for IPC (between the server and the client computers) for the purpose of method calls and to obtain result sets from those method calls. Compute servers also typically use the Message Passing Interface (MPI) for

load balancing
- **Database servers** that provide interfaces (such as Open Database Connectivity [ODBC], OLE DB and OCI) to clients for access to database tables, views, stored procedures, and triggers
- **Web servers** that contain Java Server Page (JSP) and Common Gateway Interface (CGI) access programs for generating dynamic Hypertext Markup Language (HTML) content from servers such as database servers and file servers

Later sections in this chapter discuss application migration methodology for UNIX workstation and server application architectures, together with the migration of those architectures to a Windows workstation and/or server environment.

User interfaces

The type of user interface has a clear bearing on the application migration. The following sections discuss the three most common types of user interfaces for a UNIX application: character based, graphical and browser based.

Character-based interfaces

The traditional UNIX user interface incorporates a character-oriented, scrolling terminal. UNIX was created before GUIs were available. A user types commands in the UNIX user interface, and UNIX responds with ASCII character output. UNIX was developed as a multiple-user, timesharing operating system, which assumed that users would sit at so-called "dumb" terminals and type commands on the keyboard.

Applications often use both cursor movement and the graphics capabilities found in modern terminals or emulators. To support this, UNIX has a database of terminal capabilities known as *termcap*. Some newer implementations of this use binary data and are known as *terminfo*. These libraries allow application writers to query the database for specific cursor movement commands, so applications can operate with a variety of hardware without embedding specific terminal commands into the application.

Another application development package specifically designed to alleviate the problem of terminal dependence is the *curses* library originally written at the University of California, Berkeley. It is a set of functions for manipulating terminal input and output (mostly output). These functions perform such actions as clearing the screen, moving the cursor to a specific row and column and writing a character or string to the screen. There are also input functions to retrieve user input in various modes, such as reading the input one character at a time or as a character string. Curses and similar libraries enable companies to create highly interactive, character-based applications, such as text editors.

Character-based interfaces can be ported directly into an Interix environment, or they can be rewritten to support the Windows to support Windows user interface standards. You should choose the latter option when user interface standardization is a main reason for the migration.

Graphical interfaces

Graphical interfaces in UNIX are typically based on the X Windows standard. X Windows (typically called X) is a combination of several elements: the X Protocol, X Display Server, X Clients, low-level APIs called Xlib (libX11.a) and higher-level libraries. The X Windows standard was developed at MIT to create a platform-independent, network-based, graphical user environment. X Windows separates the server part that draws the graphical interface (X Display) from the client—the application program that

uses X Windows. The server and client can run on separate computers, so the application can run on a powerful compute server, while the X Display server runs at a workstation, listening for network connections at a specific port and acting on the commands sent by the X Clients.

Because X Windows is a set of toolkits and libraries, it has no look and feel like Windows or Mac OS. Motif is the windowing system, library and user interface style built on the X Windows system. Motif handles windows and a set of user interface controls known as widgets. Widgets cover the whole range of user interface elements, including buttons, scroll bars and menus.

Even X Windows with Motif is sometimes not enough to handle the requirements of the application to provide a user interface rich in widget content. Additional widget libraries have therefore been developed. One example is the xrt widget library, which makes use of the Motif libraries. Other libraries that you may encounter include Trolltech's Qt product and INT's Carnac and GeoToolkit GUI toolkits.

Graphics-intensive applications may support additional user interface standards, such as OpenGL. OpenGL has become a widely used and supported two-dimensional and three-dimensional graphics API. OpenGL fosters innovation and speeds application development by incorporating a broad set of rendering, texture mapping, special effects and other powerful visualization functions. Developers can take advantage of the capabilities of OpenGL across all popular desktop and workstation platforms, ensuring wide application deployment.

OpenGL runs on every major operating system, including Mac OS, OS/2, UNIX, Microsoft Windows 95, Microsoft Windows 98, Windows 2000, Microsoft Windows NT®, Linux, OPENStep and BeOS; it also works with every major windowing system, including Win32, Mac OS, Presentation Manager and X-Windows. OpenGL includes a full complement of graphics functions. The OpenGL standard has language bindings for C, C++, Fortran, Ada and Java, so applications that use OpenGL functions are typically portable across a wide array of platforms.

Because Interix supports X Windows and an array of platforms support OpenGL, you may be able to move graphical UNIX applications to Interix with little or no modification. However, older applications may use an older version of the windowing system, or may use older widget libraries that have no direct mapping in Interix. Also, as with character-based interfaces, you may prefer the option of rewriting the interface to provide a standard look and feel across Windows-based applications.

Browser-based interfaces

The advent of the Web has resulted in a number of applications relying on Web browsers for their user interfaces. Such interfaces can usually be ported with a minimum of effort because of the platform independence of nearly all the standards used by Web browsers and Web servers.

Because Web-based interfaces require little or no migration effort, this guide does not cover them.

Application code

The application code provides a first step toward analyzing the application structure. Most UNIX code is written in either C or C++. However, Fortran is also a language to consider. Fortran code is common in compute-intensive scientific and engineering applications. A Fortran application can be a stand-alone application, a routine called from other languages, or it may call routines in other languages. Fortran is commonly used in loosely coupled distributed grid computing. Distributed grid computing typically requires integration with dynamic scheduling and high-performance message passing, such as the MPI.

A Fortran application requires additional planning. The Fortran considerations for using either the Interix POSIX subsystem or UNIX emulator are the same as for C and C++ migrations. Note, however, that GNU Fortran 77 compiler, f77, is currently the only Fortran compiler available for Interix.

Low-level services

Any UNIX application that uses primarily C and C++ run-time calls may compile with modification on the Win32 API. If, however, the application makes system calls for low-level services such as process creation, shared memory, semaphores, signals, message queues and third-party library calls (library not available on Win32), these calls will not compile or link with Win32. You should consider a rewrite when only a few of these calls are made; however, a port to the Interix subsystem is desirable for applications that make large numbers of UNIX system calls, especially when those calls are made throughout the code and not limited to easily identifiable modules.

At this stage, it is sufficient to understand the presence and use of low-level services in the application, such that you can give those services a thorough analysis at the evaluation stage. Use the information in Appendix x and a text editor or text search tool (such as grep) to perform an initial analysis of the code for UNIX API dependency. You can then identify likely areas of the code likely to contain UNIX-specific services—for example, in device-specific or process management code.

Installation, configuration and execution scripts

UNIX applications usually have a complement of scripts that are used in the configuration, build, installation, setup and deployment environments to support the application. These scripts are written in a variety of languages. Common scripting languages include Perl and Shell (Bourne, Korn, C and Bash) scripts. The scripts take advantage of a number of UNIX utility features, along with features of the application itself.

The functions performed by UNIX scripts and utilities include configuration, setting up the user's desktop environment, scheduling (**cron**), monitoring (for example, Simple Network Management Protocol [SNMP]), parsing or searching (**grep**, **sed**, **awk**) and administration (for example, distribution of updated application binaries). Migrating an application from UNIX to Windows requires a combination of scripting languages to create similar script syntax and utilities to perform these functions.

To choose the correct combination of scripting language, address the following questions:

- Are UNIX shell scripts used for configuration of the application environment? If so, further determine the following:
- Which shell scripts are used for configuration (C, Korn, Bourne, Bash)?
- Are any core UNIX utilities used for tasks such as installation and setup? (Core utilities are those found in the /bin directory.)
- Which file systems do the shell scripts access, and do those file systems use or expect a UNIX-style, single file system root? For further information on UNIX file systems, see Chapter 8.
- Is any current use made of cross-platform scripting, such as Perl, NuTCRACKER or Interix?
- How is an application started and what parameters are needed? Must the superuser start the application?
- Is a daemon used? In UNIX, you can start an application in the background and log off, and the application continues to run. This can't be done with Windows 2000 unless the application is a service.
- How is the application closed? It is common in UNIX to have an application that receives a

message to close, and is the application then responsible for closing all the other applications?
- Does this application interface with any other applications, and if so, which ones?
- Which shells does this application require or rely on?
- Do the migrated applications require standard UNIX utilities (such as cd or ls) at run time?

Migrated applications often require specific UNIX-style components in their deployment environments. Because of this requirement, the UNIX emulated environment (that is, NuTCRACKER or Cygwin) or UNIX native environment (that is, Interix) must provide a full set of UNIX utilities and shell environments, such as Korn and C. Also because of this requirement, the scripts must be migrated (rewritten or ported) to the Windows 2000 environment.

Development Environment

The development environment not only provides valuable input to the migration strategy, it also must be migrated itself. You need to understand the development environment for the application so that you can provide a basis for the application's continued development, support and maintenance. The sections that follow discuss the tools that the development environment may include.

Modeling tools

Modeling tools include such facilities as graphical tools in support of a given methodology, such as Structured Systems Analysis and Design Methodology (SSADM), Rational Unified Process (RUP) or Unified Modeling Language (UML). Although such tools are normally independent of the applications, facilities such as code generators and code synchronizers enable models to be kept up to date with the code.

The migration of modeling tools forms an important part of your application migration from UNIX to Windows. You therefore must include the migration of these tools in your planning process by determining the answers to the following questions:

- Is a specific methodology being adhered to?
- Are graphical modeling tools being used?
- Is there an ongoing requirement to keep models up to date with the code?

Many modeling tools are available across different platforms, including Windows. Therefore, the migration of these tools should not significantly impact the migration of the application. Because of this, and because that there is such a wide range of tools on the market, this guide does not cover the migration of modeling tools in detail.

Build tools

As part of your application data-gathering efforts, you must know which build tools and configuration scripts are used in the application's build environment.

The most common build tool for UNIX applications is the Make utility, which works with the Makefile configuration file to automatically determine which pieces of a large program need to be recompiled. The Make utility also issues the **gcc** command to recompile those pieces. Every time a source file changes, entering **make** at a shell prompt causes all the necessary recompilations to be performed. The Make utility uses the last-modification times of the files to decide which of the files must be updated. For each of those files, the Make utility issues the commands recorded in the Makefile.

There are two main implementations of Make: BSD's Make and GNU's Make, which is referred to as *gmake*. Make can be used with any programming language that has a compiler which runs with a shell command. In fact, Make is not limited to programs. It can be used to describe any task where some files must be updated automatically from others whenever the others change.

To use Make, you must create (or generate—for example, through a Configure script, as described later in this section) the Makefile, which describes the relationships among files in the program and states the commands for updating each file. For a program, the executable file is typically updated from object files, which are in turn made by compiling the source files.

The Make utility carries out commands in the Makefile to update one or more target names, where the name is typically a program. If no **-f** option is present, Make looks for a file: GNUmakefile (if gmake), makefile and Makefile, in that order. The first name checked, GNUmakefile, is not normally used. It is used only if a Makefile is specific to gmake, and will not be understood by other versions of make. The Make utility updates a target if the target depends on prerequisite files that have been modified since the target was last modified, or if the target does not exist.

Imake is a C preprocessor interface to the Make utility. It is used to generate Makefiles from a template, a set of GNU C-Compatible Compiler Preprocessor (cpp) macro functions and a per-directory input file called an Imakefile. This interface allows computer dependencies (such as compiler options, alternate command names and special make rules) to be kept separate from the descriptions of the various items to be built.

The **xmkmf** command is the usual way to create a Makefile from an Imakefile shipped with the application software. When invoked with no arguments in a directory that contains an Imakefile, the Imake program is run with arguments appropriate for the system (configured into **xmkmf** when X Windows was first installed) and generates a Makefile.

When invoked with the **-a** option, **xmkmf** builds the Makefile in the current directory and then automatically executes make Makefiles (in case there are subdirectories), Make includes and Make depend. This is the usual way to configure software that is outside of the X Windows build tree.

You can use programs such as autoconf to generate the configuration scripts automatically, or you can use the Make utility's CFLAGS variable to convey the configuration information. Build and timestamp information is captured in a Makefile for input to the UNIX Make program.

Many applications' software packages come with a Configure script of some kind. The Configure script attempts to deduce features of the operating system and set a series of compile-time macros, which will turn on and off code appropriate to the operating system.

Configure is a script toolkit that attempts to determine what kind of system it is running on and then correctly build a Makefile, which in turn is used by make to build the application. The Configure script uses a number of tests to check for the presence or absence of a specified feature, but many of those tests involve small programs that are then run through the C preprocessor or compiled. The feature tests are generally based on the output of the C preprocessor or driver rather than on execution of the test program. Configure works well when automating and running the same build process repeatedly; however, you should allow time for such builds to be configured. (Refer to Chapter 7 for more information.)

Note that this chapter describes only the most common Configure scripts—those generated through the GNU program autoconfig 2.53. Every package has different needs, and this section does not cover all of

the Configure scripts that can be used on the many packages available.

Compilers

UNIX compilers process input files through multiple stages: preprocessing, compilation, assembly and linking. Suffixes for source file names identify the source language, but the name used for each compiler governs the default assumptions. For example, **gcc** assumes that preprocessed (.i) files are written in C and therefore assumes C style linking; **g++** assumes that preprocessed (.i) files are written in C++ and therefore assumes C++ style linking.

A large number of options can be provided to control the processing. For example, options can switch debugging on and off, refer to external libraries and specify computer-dependent behavior.

A compiler can be used to gather some information about the application's ANSI C compliance. For example, as part of Quick Port migration described later in this chapter, the build Makefile can be modified to include gcc language options that control the dialect of C that the compiler accepts. For example, the **-ansi** option supports all ANSI C programs and turns off certain features of GNU C that are incompatible with ANSI C. The **-pedantic** option issues all the warnings demanded by ANSI C and rejects all programs that use forbidden extensions. Valid ANSI C programs should compile properly with or without the **-pedantic** option (though a rare few will require **-ansi**). However, without the **-pedantic** option, certain GNU extensions and traditional C features are supported in addition to the ANSI features.

Integrated development environments

Integrated development environments (IDEs) typically provide all development tools needed for programming, including compilers, linkers and project/configuration files that generate complete applications, create new classes and integrate those classes into the current project. IDEs also include file management for sources, headers, documentation and other material to be included in the project. IDEs can also include WYSIWYG (what you see is what you get)—the creation of user interfaces that have built-in dialog box and resource editors. The debugging of the application, and the inclusion of any other program needed for development by adding it to a **Tools** menu, are also typical capabilities.

Microsoft Visual Studio® is an IDE that includes all of the functions and capabilities just described, including a complete set of development tools for building reusable applications in Microsoft Visual Basic®, Microsoft Visual C++®, Microsoft Visual J++® and Microsoft Visual FoxPro®.

The Visual Studio development system also includes Microsoft Visual InterDev™, which provides easy integration with the Internet and a full Web page authoring environment; Microsoft Visual SourceSafe®, which provides source control; MSDN® Library, which provides development information, API references, code samples, references and other information; and Installer, which helps developers create highly reliable, self-repairing applications.

Visual Studio and the individual tools and languages that it contains are the foundation for building Windows-based components and applications, creating scripts, developing Web sites and applications and managing source code.

Visual Studio allows the developer to:

- Write less code by providing programming wizards, drag-and-drop editing and reuse of program

components from any of the Visual Studio languages.
- Write code more quickly by minimizing errors with syntax and programming assistance within the editor.
- Integrate dynamic HTML, script and components into Web solutions.
- Manage Web sites from testing to production by means of integrated site management tools.
- Create and debug Active Server Pages (ASP).
- Use design-time controls to visual assemble data driven Web applications.

Visual Studio also includes the Windows 2000 Developer's Readiness Kit, which contains developer training and technical resources.

Testing, debugging and code optimization tools

After the build process has created the application's executables, the next step should be a test process. The test process can be partially or wholly automated through tools or scripts. In most cases, you need to duplicate the test process—along with its associated tools or scripts—on Windows.

Test process

When creating a test process for the application, consider the following:

- Does the application have documented test procedures?
- Does the application have its own test programs or test scripts?
- If the application has test programs or scripts, how are they implemented? For example, are they implemented through C programs or scripts, Perl or Korn shell scripts?
- Is the application instrumented with compiler preprocessor symbols?

It is a good idea to test the current UNIX versions of the application so that you can validate the process and verify that any testing programs and scripts are up to date. This may help you identify a latent bug before you find it on Windows.

Tools and/or scripts

If the application comes with its own test programs or scripts, consider migrating them along with the application. The test programs and scripts should and can go through the same portability analysis as the application programs.

Debuggers

What has been used for debugging the application on the UNIX platforms? For example, the GNU debugger, gdb, is available on almost every platform.

Packaging and archiving tools

It is important to understand the different archiving, packaging and compression facilities that are typically used for UNIX applications, because it is likely that the code will be delivered in one of these forms and will then have to be imported into the Windows environment.

The following are the archiving and packaging facilities typically used for UNIX applications:

- **tar**. The tape archive program, which uses a variety of formats and has nothing to do with tape. It is still one of the most popular formats.
- **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.
- **pax**. The POSIX.2 standard archiver. It reads, writes and lists the members of an archive file (including tar-format archives), and also copies directory hierarchies.
- **rpm**. The Red Hat Package Manager.
- **ar**. Creates and maintains groups of files combined into an archive. It is not typically used for source archives, but is almost always used for object file archives (static object libraries).

The following are the compression formats typically used for UNIX applications:

- **compress**. Creates a compressed file by using adaptive Lempel-Ziv coding to reduce the size of files (typically 70 percent smaller than the original file).
- **zip**/**gzip**. Algorithms combine a version of Lempel-Ziv coding (LZ77) with another version of Huffman coding in what is often called string compression.
- **pack**. Compresses files by using Huffman coding.
- **uncompress**, **zcat**. Extracts compressed files.
- **gunzip**. Decompresses files created through compress, zip, gzip or pack. The detection of the input format is automatic.
- **unpack**, **pcat**. Restores files compressed by pack.

Table 3 summarizes common suffixes for archived and compressed file names.

Table 3. 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**
.Z	compressed	Compressed by **compress** Uncompressed with **uncompress**, **zcat** or **gunzip**
.z	pack or gzip	Compressed by **pack** and extracted with **pcat** Compressed by **gzip** and extracted with **gunzip**
.zip	Zip	Compressed by **zip** and extracted with **unzip**, or Compressed by **pkzip** and extracted with **pkunzip**

Source code management tools

Source code management tools group a number of facilities, such as source code control, build management and code archiving. A number of source control systems are used in UNIX and cross-platform environments, including Revision Control System (RCS), Source Code Control System

(SCCS), Concurrent Version System (CVS) and Program Version Control System (PVCS) Dimensions.

When reviewing the application to be migrated, consider the following:

- Is the code currently stored in a code management system?
- Does the code management system operate in both the UNIX and Windows environments?

The native Windows source control system is Visual SourceSafe, which supports the Source Code Control Interface (SCCI) API.

Application Usage

It is important to determine the how and when the application is used, who uses it and how often it is used so that you can mitigate or deal with usage-related issues as part of the migration. The best way to determine usage is by speaking with the users and operators. You can use a combination of interviews and workshops for this. It is also helpful to gather additional materials, such as user documentation and training guides.

User-facing functionality

When analyzing application features and facilities, you should determine which are used most frequently (and are thus critical to the migration) and which are used less frequently or are not used at all.

Administrator functionality and operational management

Your analysis of the application's administrative or operational feasibilities and usage should include an examination of data management, backup/restore, security management, application configuration, startup/shutdown, failure operation and contingency planning/disaster recovery.

Service level definitions

You can define application service levels in terms of availability, scalability, performance, downtime constraints, planned outages and similar factors. To complete your analysis, you should determine the application's scalability and performance requirements.

Evaluating Migration Objectives

The output from data gathering should be an inventory of all elements affected by the migration, together with their interdependencies and any issues that you have uncovered. In the light of these issues, you should revisit the objectives of the migration to determine their viability and testability.

Business Objectives

You must determine how the application will meet business needs. Business needs vary, but can include the following examples:

- The appearance and functionality of the application's UNIX user interface must be retained after the migration.
- A UNIX-style development environment must be retained after the migration.

- The application must run on multiple platforms, so coexistence requirements must be examined.
- Common code must be used across multiple platforms.
- The application's development is static.
- The application will be replaced, so minimal changes are required.
- The development environment will be migrated to Visual Studio.

Technical Objectives

You must establish criteria to ensure that the migration is successful. Consider the following when determining your criteria:

- Can the integration of Windows-based applications be accomplished through Interix?
- Is there significant dependence on UNIX APIs?
- Is there good Interix support for required APIs?
- Is there reliance on third-party libraries?
- Are the third-party libraries available with Interix?
- Are the third-party libraries available with Win32?
- Does the application conform to ANSI C/C++?

Migration Objectives

You must establish criteria to judge that the resulting application delivers value. These criteria can include the following:

- What is the ultimate intended use of the software? For example, is one of the objectives to improve usability and worker productivity by integrating the application with other Windows-based personal and workgroup productivity applications, such as Microsoft Word or Microsoft Excel?
- Will the application behave as a native application in each unique target environment, particularly with regard to software installation, administration, distribution and user interface?
- Is the migration time critical?
- Is the required user interface based on the Windows GUI?

Evaluating the Application

Now that data gathering is complete, you can begin to thoroughly analyze the application. You should examine each building block of the application to determine any issues that may arise in porting the code to the new platform.

The application blocks include the following:

- User interface
- File and device I/O
- File and process security
- Processes and threads
- Interprocess communication
- Signals

The results of this examination will provide you with insight into any migration issues that may exist, and therefore the most appropriate approach to migrating the application. If you find a substantial

number of issues, you should consider a rewrite of the application. More likely, however, you will need to choose between porting the application to the Interix environment or making it work under Win32.

In the analysis, you should look for use of UNIX-specific code. UNIX applications can contain millions of lines of custom code. The effort to rewrite an application generally increases as the amount of code increases, and using porting tools becomes more viable as code sizes increase. The major issues are normally with code that the application uses to communicate with the UNIX operating system through system calls. Solaris, HP-UX, Advanced Interactive Executive (AIX), Linux, FreeBSD and other UNIX brands all have some unique architectural features, APIs, commands and utilities.

UNIX-specific code will use either UNIX standard conventions (for example, the file hierarchy) or function calls that are specific to the source UNIX environment. You should log each occurrence of a UNIX-specific code element, because it will influence the decision on how to migrate the application.

In addition, you should consider whether the application code has been written in a hardware-independent manner. Examine the word size of the basic data types (for example, 64-bit versus 32-bit pointers), byte ordering (big-endian versus little-endian) and data alignment in structures. To facilitate portability, all hardware dependencies must be isolated and conditionally compiled and linked for the target environment build process. Or, all hardware dependencies must be rewritten to use hardware-independent constructs. UNIX applications that are designed around modular and portable coding methodologies have taken these issues into consideration.

You should determine whether the application contains custom device drivers. For example, custom device drivers are very common in process control applications. These device drivers are not portable and must generally be rewritten for the Windows 2000 platform.

User Interface

In your evaluation of the application, you should review the user interface to determine how it is built (that is, what libraries it uses) and what standards (if any) it uses.

X Windows/Motif

You can determine whether the UNIX application is using X Windows, Motif or xrt libraries by looking at the make program's Makefile and the output of the application's build. For example, you can use **grep** and **ldd**, as described earlier in this chapter.

X Windows libraries include:

- X11 toolkit library (libXt.a)
- X11 intrinsics library (libXi.a)
- Athena widget library (libXaw.a)
- X11 extensions library (libXext.a)
- X Windows Display Manager (XDM) control protocol library (libXdmcp.a)
- Xauthority routines library (libXau.a)
- Miscellaneous utilities library (libXmu.a)

When a UNIX application makes calls to these libraries, it is linked to one or more of the following: X11, Xau, Xaw, Xi, Xmu, Xt and Xtst. These libraries contain several hundred API calls and do not map easily to the Win32 user interface API. To successfully port a user interface that uses this API, you must

ensure that these libraries are available on the Windows platform (for example, by using Interix's X11R5 library).

When a UNIX application makes calls to the Motif API, it is linked to either or both of the following libraries: xm, mrm. These libraries also contain several hundred API calls that do not map easily to the Win32 GUI API. To port the user interface that makes use of this API, you must ensure that the Motif libraries are available on the Windows platform. Note that Motif libraries are built on top of X11R5 or R6 libraries and therefore require those as well. Additionally, the Motif Window Manager, mwm, is required to perform window management functions.

OpenGL

OpenGL is an API that allows an application to manipulate three-dimensional graphics on the screen, and it is available on both UNIX and Windows. On UNIX, OpenGL is often mixed with X Windows and Motif code to display buttons, menus and dialog boxes. When this is the case, the X Windows and Motif code guide the migration choice between an application port or a rewrite.

Character-mode interfaces

A character-mode user interface application writes to the console one line at a time (for example, by using the **C printf()** library call), and data input is requested through the use of prompts. This code can easily be migrated to Win32, usually by just recompiling the code.

When a UNIX application makes calls to the **curses** API, it is linked to a library called Curses or nCurses. These libraries contain several hundred API calls that do not map easily to the Win32 user interface API. To successfully port a user interface that makes use of these APIs, you must ensure that the libraries are available on the Win32 platform. This library makes use of the terminfo technology that is available on UNIX and not on standard Windows 2000. Use of the Interix curses or ncurses library makes a port of this user interface relatively easy.

File and Device I/O

The UNIX approach to file access differs from that of Windows. UNIX has a single file hierarchy based on the root file system (as indicated by a slash mark), whereas Windows uses a separate letter for each file system (for example, A and B for floppy disks, C through Z for hard disks). UNIX determines the location of the root file system only at boot time; the other file systems are added to the directory tree by mounting them (for example, with the entry: **mount /dev/fd0 /mnt/floppy**) by means of entries in the table /etc/fstab (or possibly /etc/vfstab or /etc/mnttab).

At the lowest level of the UNIX file system, files are referred to by numbers—that is, *inodes*. Inodes are indices to the *Inode table*, a part of the file system reserved for describing files (somewhat similar to the role of the file allocation table [FAT] in the file system included with the Microsoft MS-DOS® operating system). The directory system enables a file to be referred to by a name. The relationship between a file name and an inode is called a *link*.

There are two types of links: *hard links* and *symbolic links*. A hard link (sometimes referred to as a traditional link) links a filename to an inode and also enables a single file to have multiple names (that is, links). A symbolic link is a file whose contents are the name of another file. In a hard link, each of the file names has the same relationship to the inode; in a symbolic link, the symbolic link name refers to the true name and directory location of the file. If you delete one of several files (including the original)

that are cross-referenced by hard links, the other file names will continue to work, but if you delete a file that is referenced by a symbolic link, then the symbolic link will point to nothing.

The network file system is a method of sharing file systems across networks. NFS has some similarities to, but is quite different from, the server message block (SMB) file system used on MS-DOS and Windows. With NFS, a UNIX computer can mount file systems connected to a different computer on the network—for example, mount hostname:/exporteddir1 /mnt. From the perspective of the local computer, /mnt is now just another portion of the single file hierarchy.

Devices are also treated as files from a UNIX application perspective.

The following are some questions that you need to ask to obtain information about the application from the perspective of file and device I/O:

- Is there a reliance on absolute path names?
- Are hard links and/or symbolic links used? Example calls to look for include **readlink()**, which reads the contents of a symbolic link, and **symlink()**, which creates a symbolic link to a file.
- Are there any NFS file system dependencies?
- Are file and device function calls used and required, especially the use of calls that are neither ANSI C/C++ nor POSIX compliant? The call to look for is **chsize()**, which changes the end of file on an open file.
- Is non-blocking file I/O (that is, asynchronous I/O) used and required?
- Are there any file-locking and/or record-locking requirements?
- Are memory mapped files being used? Example calls to look for are **mmap()**, which maps a file into memory, and **munmap()**, which removes mappings for files in memory.

Interprocess Communication

As discussed in Chapter 2, UNIX introduced a philosophy of computing with features such as *pipes*, which provide the ability to link the output of one program to the input of another. Pipes are just one example of the ability to transfer data between processes. Various UNIX system implementations offer many other forms of interprocess communication, as explained in the following subsections.

Process pipes

Process pipes are found in all versions of UNIX and are also supported by Interix. They transfer data in one direction only. In general, the output of one process is piped (attached) to the input of another. Process pipes require ancestry between the processes (for example, a parent/child relationship).

Function calls to look for in the application are:

- **pipe()**. Creates a pipe.
- **popen(), pclose()**. Processes I/O by using pipes.

Named pipes

Named pipes, also called FIFO (first in, first out) pipes, are process pipes with file names that allow unrelated (no ancestry) processes to communicate with each other. Named pipes transfer data in one direction only. For example, one process opens the FIFO for reading, whereas another process opens the FIFO for writing. In effect, a named pipe can be used just like a file.

Function calls to look for in the application are:

- **mkfifo()**. Makes a FIFO special file (a named pipe).
- **mknod()**. Creates a regular file, special file or directory (historical call for creating a named pipe).

System V IPC

System V IPC uses a messaging queue to facilitate interprocess communication. Processes can add or remove messages from the queue. In addition, they can access shared memory and create semaphores and read or set semaphore values.

Message queues

Message queues are like named pipes, but with two differences. Named pipes transmit a byte stream and are one directional. Message queues transmit records and these messages can have different priorities. Therefore, with message queues it is necessary to determine the sequence in which the records are received. (POSIX also has message queues, but the APIs are not exactly the same as in the System V implementation.)

Function calls to look for in the application are:

- **msgctl()**. Controls message operations.
- **msgget()**. Gets a message queue identifier.
- **msgrcv()**. Reads a message from a message queue.
- **msgsnd()**. Sends a message to a message queue.

Shared memory

Shared memory allows two unrelated processes to access the same logical memory. It is a special range of addresses that is created for one process and is added to that process's address space. Other processes attach to the same shared memory segment, and the segment also becomes part of their address space. A message is sent by writing data into a buffer that is part of this shared memory segment of both processes.

Function calls to look for in the application are:

- **shmat()**. Attaches a shared memory segment operation.
- **shmctl()**. Controls shared memory operations.
- **shmdt()**. Detaches a shared memory segment operation.
- **shmget()**. Allocates a shared memory segment.

Semaphores

When two processes share a memory segment, one process cannot determine when the other is doing something. For example, two processes could be modifying the same data at the same time. To deal with such situations, Dijkstra introduced the concept of the *semaphore.* A semaphore is a special variable that takes only positive integer numbers and upon which only two operations are allowed: wait and signal.

A small positive number (counter) is maintained in the semaphore. When a process accesses the semaphore (wait operation), it decrements the counter. If the counter is still nonzero, the process has

access; otherwise, the process is blocked and execution of the process is suspended.

Function calls to look for in the application are:

- **semctl()**. Performs a control operation on a semaphore.
- **semget()**. Gets or creates a set of semaphores.
- **semop()**. Operates on a set of semaphores.

Sockets

Beginning with BSD 4.2, sockets were introduced as part of the Transmission Control Protocol/Internet Protocol (TCP/IP) networking implementation. The sockets interface is an extension of the pipes concept. Sockets are therefore used in much the same way as pipes, but they can be used across a network of computers over the TCP/IP transport protocol. In BSD systems, the other IPC communications facilities are based on sockets.

Function calls to look for in the application are:

- **accept()**. Accepts a connection on a socket.
- **bind()**. Binds a name to a socket.
- **bindresvport()**. Binds a socket to a privileged IP port.
- **connect()**. Initiates a connection on a socket.
- **getsockname()**. Gets a socket name.
- **getsockopt(), setsockopt()**. Gets and sets options on sockets.
- **listen()**. Listens for connections on a socket.
- **recv(), recvfrom()**. Receives a message from a socket.
- **recvmsg()**. Receives a message from a socket.
- **send(), sendto()**. Sends a message to a socket.
- **sendmsg()**. Sends a message to a socket.
- **socket()**. Creates an endpoint for communication.
- **socketpair()**. Creates a pair of connected sockets.

Streams

Beginning with System V4, streams were introduced as a generalized I/O concept. Streams can be used for both local and remote communications, just like sockets. A stream is a full-duplex data path within the kernel between a user process and drivers. The primary components are a stream head, a driver and zero or more modules between the stream head and driver. A stream is analogous to a pipe, except that data flow and processing are bidirectional.

The stream head is the end of the stream that provides the interface between the stream and a user process. The principal functions of the stream head are processing stream-related system calls and data between a user process and the stream.

A module contains processing routines for input and output data. It exists in the middle of a stream, between the stream's head and a driver. A module is the streams counterpart to commands in a shell pipeline, except that a module contains a pair of functions that allow independent bidirectional data flow and processing.

Function calls to look for in the application are:

- **getmsg()**, **getpmsg()**. Retrieves the next message in a stream.
- **putmsg()**, **putpmsg()**. Sends a message in a stream.

Header files to look for in source files are:

- stream.h
- stropts.h

Stream pipes and named stream pipes

Stream pipes are similar to process pipes, but can transfer data in both directions. They may be names, as named stream pipes. Stream pipes are typically used in System V for passing file descriptors between processes.

Header files to look for in source files are:

- stream.h
- stropts.h

Note Stream pipes and named stream pipes are unrelated to streams.

Processes and Threads

The Single UNIX Specification (both UNIX95/XPG4v2 and UNIX98/XPG5) defines a process as: "an address space with one or more threads executing within that address space, and the required system resources for those threads."

UNIX is designed to be a multiple-processing, multiple-user system. At any point in time, many applications and processes are running. UNIX makes it easy to create processes, and many of the features of the operating system and shells result in the common practice of running many programs at once. A UNIX application can start a new program and process, replace its own process image and duplicate its process image. When UNIX duplicates its process image, the new process becomes a child of the creating process. This process hierarchy is often important, and there are system calls for manipulating child processes.

UNIX process-handling functions do not map directly to the Windows environment; therefore, you must identify such function calls. The following is partial list of these calls:

- **system()**. Passes a command to the shell, which starts a new program, thereby creating a new process.
- **execl(), execle(), execlp(), execv(), execve(), execvp()**. Replaces the current process image with a new process image by executing a file.
- **fork()**. Creates a new process. The new process (child process) is an exact copy of the calling process (parent process), with some exceptions—for example, the child process has a unique process ID.
- **vfork()**. Creates a new process, just as **fork()** does, but doesn't fully copy the address space of the parent process.
- **popen()**. Opens a process by creating a pipe, using **fork()** to create another process, and invoking the shell.

UNIX also makes it convenient to create a group of cooperating processes, especially when the processes access terminals. Such groups are known as *process groups*, which include the following:

- **getpgrp()**. Returns the process group ID of the current process.
- **setpgid()**. Adds a process to a process group.
- **setsid()**. Creates a new process group and sets the calling process as group leader.
- **tcgetpgrp()**. Returns the ID number of the terminal's foreground process group.
- **tcsetpgrp()**. Sets the foreground process group ID.
- **killpg()**. Sends a signal to a process group. (For more information about signals, see the "Signals" section later in this chapter).

When a program is creating processes, it also needs a variety of function calls to manage the processes. The following are some examples of function calls:

- **getpid()**. Returns the process ID of the calling process.
- **getppid()**. Returns the process ID of the parent of the calling process.
- **getpriority(), setpriority()**. Gets or sets a process's nice value (*nice* refers to setting a process priority to a low value, such that it does not take priority over other processes).
- **getrlimit(), setrlimit()**. Sets or retrieves resource limits.
- **getrusage()**. Gets information about the use of resources.
- **sleep()**. Suspends process execution for intervals of seconds.
- **wait(), waitpid()**. Waits for process termination.

A thread is a sequence of control within a process. All processes have at least one thread of execution. When the process creates a new thread, the thread gets its own stack for the maintenance of all of its function parameters and local variables required for thread execution. However, the thread shares global variables, file descriptors and other process characteristics with the other threads in the process.

The idea of threads has been in existence for some time for various UNIX systems, but until the IEEE POSIX committee created the POSIX.1c-1996 thread extensions, each UNIX vendor's implementation was unique. Threads are now much more standardized and are available on most UNIX platforms. The POSIX threads are known as pthreads; some of the function calls are as follows:

- **pthread_create()**. Creates a new thread.
- **pthread_exit()**. Terminates the calling thread.
- **pthread_join()**. Waits for termination of another thread.
- **pthread_detach()**. Puts a running thread in the detached state.
- **pthread_attr_init()**. Initializes the thread attribute object **attr** and fills it with default values for the attributes.

The following are some questions that you need to ask to obtain information about the application from the perspective of its process and thread structure:

- Does it make extensive use of process creation and management?
- Does it make use of process groups?
- Does it need to run under the context of a different user or group at times during its execution?
- Does it make extensive use of multiple threads (that is, is it multithreaded)?

Process management

Functions in this category provide support for the scheduling and priority management of Processes.

Note These functions are not supported by Interix.

Process functions	Function description
getexecname	return pathname of executable
p_online	return or change processor operational status
Priocntl	display or set scheduling parameters of specified process(es)
Priocntlset	generalized process scheduler control
Processor_info	determine type and status of a processor
pset_assign	manage sets of processors
pset_create	manage sets of processors
pset_destroy	manage sets of processors
pset_info	get information about a processor set
Setpriority	set process scheduling priority

File and Process Security

The security models of UNIX and Windows 2000 applications are different. Most UNIX and Windows applications have not been written to support a common security model, such as Kerberos.

UNIX systems also have a concept of *setuid* (set-user-identifier-on-execution) and *setgid* (set-group-identifier-on-execution) bits in the permissions bits for a file. By using the chmod utility, the setuid bit and/or setgid bit can be set and reset on files. All UNIX systems maintain at least two user/group IDs, the *effective* user/group ID and the *real* user/group ID. Some systems also support a *saved* set user/group ID.

These IDs are not only used for user access to files. Processes are also allocated certain permissions. UNIX systems manipulate user/group IDs in the following ways:

- If the setuid or setgid bit is not set, UNIX runs the program as the specific user or group that executed the program, and therefore, both the effective user/group ID and the real user/group ID are set to the user or group that executed the program.
- If the setuid or setgid bit is set, UNIX runs the program under the user/group ID of the file's owner, not the user/group ID of the user who is executing the file. The effective user/group ID is set to owner of the program (file), and the real user/group ID is set to the user or group that executed the program (file). If there is a saved set user/group ID, it is also set to the owner of the program (file).

There are also a number of function calls that allow a program to manipulate the effective, real and saved set user/group IDs:

- **getgid()**. Returns the real group ID of the calling process.
- **getegid()**. Returns the effective group ID of the calling process.
- **setregid()**. Sets real and effective group IDs.
- **setreuid()**. Sets real and effective user IDs.
- **setuid(), seteuid(), setgid(), setegid()**. Sets real and effective user/group IDs.
- **setuser()**. Changes effective and real user/group IDs of a process.

Sometimes a trusted application may require access to files and/or services to which the user should not have unlimited access. For example, the UNIX application *ps* requires access to /dev/proc, which only the superuser (root) has access to. UNIX solves this problem by allowing the application to run as a specific user or group. That is, if you execute an application that has the setuid bit set in the file permissions, this bit instructs the kernel to run the application with the identity and privileges of the owner of the file. Similarly, the setgid bit set causes the application to run as if it had been executed in the group to which the file belongs.

Signals

A signal is an event generated by the shell and by terminal device handlers in the UNIX operating system, in response to a condition. Signals tell a process that an exceptional condition has occurred or that something needs attention or action. For example, signals happen when the computer hardware detects conditions like floating point overflows and memory segment violations. Signals are also used for interprocess communication. Common IPC signals are for ending a process or notifying a parent process that a child process has stopped executing.

It is important to understand the UNIX application's signal implementation to determine what issues you will encounter during the migration to Windows. The following list provides a brief overview of the signal models of each of the four UNIX systems, and therefore, what to look for in the application:

- UNIX Seventh Edition had unreliable signals (that is, there was no guarantee of the signal's success) and provided only one signal function: **signal()**.
- BSD 4.2 introduced reliable signals (that is, signal status could be checked and acted upon), supported by the following functions:
  - **killpg()**. Terminates a process group.
  - **sigsetmask()**. Replaces the set of blocked signals with a new set specified in a mask.
  - **sigblock()**. Adds (as a logical OR) the signals specified in a mask to the set of signals currently being blocked from delivery.
  - **siggetmask()**. Returns the current set of masked signals.
  - **sigvec()**. Sets the disposition of a signal.
  - **sigaltstack()**. Gets or sets alternate signal stack content.
- System V introduced another implementation of reliable signals, supported by the following functions:
  - **sigset()**. Modifies the handling of the specified signal.
  - **sighold()**. Blocks delivery of the specified signal by setting the corresponding bit in the process signal mask.
  - **sigrelse()**. Allows delivery of the specified signal by resetting the corresponding bit in the process signal mask.
  - **sigignore()**. Sets the disposition of the specified signal to SIG_IGN (ignore signal).
  - **sigpause()**. Allows delivery of the specified signal by resetting the corresponding bit in the process signal mask and then suspends the process until any signal is delivered.
  - **sigaltstack()**. Gets or sets alternate signal stack content (also XPG4 conformant).
- POSIX.1 introduced a third implementation of reliable signals, supported by the following functions:
  - **sigaction()**. Changes the action taken by a process on receipt of a specific signal, and also specifies a mask of signals to be blocked during the processing of a signal.
  - **sigpending()**. Gets a pending signal mask.
  - **sigprocmask()**. Manipulates the current process signal mask.
  - **sigsuspend()**. Temporarily sets the process signal mask and then waits for a signal.
  - **sigemptyset()**. Initializes the given signal set to empty, with all signals excluded from the

set.
- **sigfillset()**. Initializes the set to full, including all signals.
- **sigaddset()**. Adds the specified signal to the set.
- **sigdelset()**. Deletes the specified signal from the set.
- **sigismember()**. Tests whether the specified signal is a member of the set.

Note Signals on a POSIX.1 system are neither BSD nor SVID. POSIX defined a new signal mechanism based on the sigaction() API.

In addition, there is an ANSI C signal implementation that uses the **signal()** function. This API is built on top of the POSIX.1 **sigaction()** model. Furthermore, the ANSI **raise()** function sends a signal to the current process.

The POSIX.1 committee introduced its new signal semantics because of problems with traditional signal implementations found on BSD and System V systems. When the System V3 **signal()** function catches a signal, the action associated with the signal is reset to the default. In BSD 4.3, the action is not reset. In the 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.

There may be an opportunity during the application's migration to Windows to convert the signal code to use the POSIX.1 signal calls.

Defining the Migration Strategy

Migrating a UNIX application to Windows 2000 may be as simple as recompiling, or it may require significant redesign and recoding. The amount of work required depends on how portable the application source code is to begin with, whether compatibility tools (such as Interix) are being used, and the intended use of the application on the target platform.

Now that you have set your objectives and evaluated the application, it is time to define a strategy for the migration. Chapter 3 gave an overview of possible migration methods, including:

- Quick port to Interix
- Full port
- Rewrite
- Coexistence
- Some combination of the preceding methods

You must now decide the most appropriate course of action for your own application. This is documented in the migration strategy, along with:

- Testing/certification requirements
- An infrastructure conceptual design that covers the target platform, including all of the migrated application's infrastructure requirements, such as:
  - UNIX and Windows resource sharing (for example, file sharing)
  - Security and user account synchronization (for example, integration and interoperability of the Active Directory® directory service and Network Information System [NIS])
- System and application management and support (for example, software updates and remote support)

Application Classification

A starting point in defining a migration strategy is to classify your application into one of the following four categories:

- ANSI-based
- ANSI with cross-platform libraries
- "Portable" UNIX
- "Native" UNIX

Table 4 provides a template to help determine an application's category. Based on the issues raised during the data gathering and analysis activities, it may be very clear how an application fits into one of the defined categories. In other cases, it may not be so clear which category the application fits into. In these cases, it is worth examining the application on a subsystem-by-subsystem basis, determining the best strategy for a given element of the application. The result should be a migration strategy defined for the application as a whole, which you can then verify against the issues raised during data gathering and analysis.

Table 4. Application category

Category	ANSI C, C++, Fortran	ANSI with cross-platform libraries	"Portable" UNIX	"Native" UNIX
Level of use/dependency on UNIX API libraries	Low (insignificant)	Low (insignificant)	High (significant)	High (significant)
Use of cross-platform libraries	No	Yes	Yes or No	Yes or No
Use of third-party libraries	Yes or No	Yes or No	Yes or No	Yes or No
Use of third-party compilers	Yes or No	Yes or No	Yes (if supported on Interix) or No	Yes or No
High level of compatibility with existing code required	Yes	Yes	Yes	No
Standards the application supports	ANSI C, C++, Fortran	ANSI	Some Combination of: BSD, POSIX, System V, UNIX95, UNIX98, XPG	Some Combination of: BSD, POSIX, System V, UNIX95, UNIX98, XPG
Migration approach	**Recompile to Windows**	**Recompile to Windows**	**Port to Interix**	**Rewrite to Windows**
Possible issues/concerns	Performance of recompiled code	Third-party library vendor support on Windows	Cross-platform library vendor support on Windows (for example, Rogue Wave)	Performance of recompiled code
Windows Services for UNIX features	Third-party library vendor support on	Performance of ported code	Customer doesn't have time or	Third-party library vendor support on

are missing, or the customer is concerned about Microsoft support	Interix	resources for redesign and rewrite of the application	Windows

The following sections discuss the four application categories (ANSI-based, ANSI with cross-platform libraries, "portable" UNIX and "native" UNIX) in greater detail. After you summarize the characteristics of the application and identify the application's category, you can further analyze potential migration approaches and strategies.

ANSI-based

A pure ANSI-based code base should be portable to any platform that supports the ANSI standard, including Win32. However, use of third-party libraries may make a direct port more difficult or may make it unsuitable.

- Case 1. No use of third-party libraries, or third-party libraries supported on Win32:

  Port to Win32 and Visual Studio.

- Case 2. Third-party libraries supported on Interix:

  Port to Interix with third-party library.

- Case 3. Third-party libraries not supported on Win32 or Interix:

  Contact library provider for support on Win32 or Interix.

If a Fortran code migration is required during an application migration from UNIX to Windows, Fortran needs to be an integral part of the design. Considerations such as performan2ce, library interoperability and feature set play key roles in the overall success of the project.

ANSI with cross-platform libraries

Migrating a UNIX application that is dependent on a cross-platform library to Windows can be as simple as acquiring, installing and configuring the Windows version of the library (for example, Rogue Wave SourcePro C++ or Trolltech's Qt product for Windows), recompiling; and then testing.

For example, Rogue Wave SourcePro C++ provides object-oriented C++ components designed for portability. Rogue Wave C++ interfaces are stable and most important in the context of portability, consistent across platforms. What this means in almost all cases is that a simple recompilation of code is all that you need to do to port from UNIX to Windows platforms. This assumes that the application is based entirely on Rogue Wave SourcePro C++, platform independent language features and any other libraries that satisfy the same platform portability features as provided by Rogue Wave.

However, this kind of a migration can require significant redesign and recoding.

- Case 1. Cross-platform libraries supported on Win32:

  Port to Win32, Visual Studio, with cross-platform library.

- Case 2. Cross-platform libraries supported on Interix:

  Port to Interix with cross-platform library.

- Case 3. Cross-platform libraries not supported on Win32 or Interix:

  Contact library provider for support on Win32 or Interix.

"Portable" UNIX

"Portable" UNIX is required when the customer needs or wants a high level of compatibility with existing UNIX software.

- Case 1. Static applications (late in lifecycle and stable code, little or no change planned):

  If no third-party/cross-platform libraries are required, port to Interix.

- Case 2. Evolving applications (early in lifecycle and still changing and being enhanced):

  Partition the application into static and dynamic components, and then port static portions to Interix and rewrite dynamic portions to Win32 and Visual Studio.

"Native" UNIX

Native UNIX is required when the customer needs or wants to take advantage of Windows or Microsoft .NET.

- Case 1. Static applications (late in lifecycle and stable code, little or no change planned):

  If no third-party/cross-platform libraries are required, rewrite the application to Win32 (assuming resources, time and desire).

- Case 2. Evolving applications (early in lifecycle and still changing and being enhanced):

  Rewrite the application in Win32 and Visual Studio.

Migration Approaches by Application Type

During the analysis activities, you identified the type of your application; that is, whether it is designed to run on a workstation, a compute server, a database server or a Web server. Each application type has certain attributes that make it more appropriate to either rewrite or port the application, as described in the following sections.

Workstation

You must consider several factors when deciding whether to rewrite or port workstation applications. The number of processes involved, the interactions of those processes and the type of user interface (for example, character-based or GUI-based) are critical to the decision to rewrite or port.

A small, single-process application with a character-based user interface will likely contain few systems

calls, and therefore should be considered for a port to Win32. Multiple-process applications will contain many system calls for process management and interprocess communication. These APIs are quite different on Win32 from those on UNIX and will present challenges to the rewrite team. A port (to Interix) is recommended in this case.

OpenGL code, if stand-alone or not mixed with system calls or X Windows/Motif code, can be ported to the OpenGL environment on Windows, with minor changes. If there is a mixture, the migration strategy will depend on the mix.

Compute server

Message-based server application architectures allow for the choice of a strategy for each of multiple components, and you can employ a different strategy for each component. This chapter assumes that the application is based on a TCP/IP-based RPC mechanism (for example, ONC, distributed computing environment [DCE] or proprietary), sockets or third-party messaging APIs. Sockets, ONC RPC, DCE RPC and some third-party messaging layers are available in the Win32 subsystem, whereas proprietary APIs would need to be ported.

Different (and incompatible) versions of RPC exist for both UNIX and Windows 2000. On UNIX, it is common to have both ONC RPC and DCE RPC. Although these RPCs perform similar functions, their implementations and network marshaling specifications are different. ONC RPC uses External Data Representation (XDR) for network marshaling, whereas DCE RPC uses Network Data Representation (NDR) for marshaling. On Windows 2000, RPC is natively implemented uses Microsoft RPC, which also uses NDR.

Sockets, ONC RPC, DCE RPC and some third-party messaging layers are available in the Win32 subsystem (proprietary RPC environments would need to be ported). The availability of these messaging layer APIs enables the migration of UNIX applications by means of messaging mechanisms for interprocess communication to Windows.

Many of the native UNIX utilities, such as NFS and NIS, are implemented through Sun Microsystems's ONC RPC. Applications written for UNIX often either use these utilities or implement their own ONC RPC calls. If this type of application is migrated to Windows 2000, a development and client runtime will be needed to support ONC RPC. Tools such as Interix and NuTCRACKER provide support for ONC RPC. Additionally, shareware tools are available to support ONC RPC on Windows 2000.

A client/server architecture allows the choice of a migration strategy for each component. Provided that the networking APIs are available, you can evaluate each component individually as a separate program to decide whether to rewrite or port. Note that in some cases, the client component might already exist on the Windows platform.

Database server

The initial migration decision is about the database. In other words, you must determine whether to run the application's current database (for example, Oracle) on Windows (that is, migration of the existing database from UNIX to Windows) or migrate the database to Microsoft SQL Server™.

In some cases, the best migration decision is a conversion to SQL Server. However, the investment in development and tools in the current UNIX database sometimes makes a conversion to SQL Server cost prohibitive. This can be particularly true in an environment with significant UNIX/Oracle resources.

Oracle offers a range of features available to both Windows 2000 and UNIX. You may choose to use these features with the current Oracle database , even if your overall goal is a migration to SQL Server. By separating the platform migration from the database migration, you have greater flexibility with the migration of their database applications.

Web server

Web applications from a UNIX Web server are normally one of two types: Common Gateway Interface (CGI) or Java Server Page (JSP).

The Common Gateway Interface is a standard for connecting external applications with information servers, such as Hypertext Transfer Protocol (HTTP) or Web servers. An HTML document that the Web server retrieves is static, meaning that it exists in a constant state. A CGI program is executed in real time so that it can generate dynamic information. For example, a CGI program on the Web service executes to transmit information to a database engine, retrieves result sets and displays the result sets to the client browser.

The migration strategy for CGI programs or scripts depends on what language the program was written in. A CGI program can be written in any language that can access both types of environment variable and the standard input and output streams, and that allows the program to be executed on the system. Appropriate languages include C/C++, Fortran, Perl, Tcl, UNIX shells, and Visual Basic. Because of the powerful text-handling capability of the Visual Basic programming language, many Web developers write CGI programs in Visual Basic.

For example, CGI2ASP is a framework for porting programs written in Visual Basic, using Windows CGI (an O'Reilly & Associates product), to a component-based ASP application with virtually no change to the existing Visual Basic-based code.

Java Server Page technology also allows for the development of dynamic Web pages. JSP technology uses Extensible Markup Language (XML)–like tags and scriptlets written in the Java programming language to encapsulate the logic that generates the content for the page. The application logic can reside in server-based resources (for example, the JavaBean component architecture) that the page accesses by using these tags and scriptlets. Any and all formatting (HTML or XML) tags are passed directly back to the response page. JSP scriptlets written in the Java programming language and residing in JavaBean components would be rewritten in Microsoft Visual Basic Scripting Edition or Microsoft JScript® and contained in ASP Component Object Model (COM) components.

Migration Considerations

There are some additional considerations that you should apply to your migration approach. These considerations apply to the migration as a whole, and also to the specific type of migration that is planned.

General considerations

Several key issues and considerations that you must take into account when migrating a UNIX application to Windows 2000 include:

- Will a common code base that is deployable across UNIX and Windows platforms be built, maintained and evolved?

- To what degree will Windows 2000 features be added to improve the installation, performance and/or usability of the migrated application?
- To what extent will the UNIX and Windows versions of the software need to interoperate across platforms?
- Does the application or certain components of the application require temporary or long-term ongoing deployment across both UNIX and Windows platforms? Is the plan to deploy Windows 2000 workstations while retaining existing UNIX platforms, or are the UNIX platforms being replaced with Windows 2000?
- As new features and/or fixes to defects are applied, can they be applied only once to one common code base that can be deployed everywhere needed, or are there separate code bases that must be maintained and evolved independently for each target platform?

Quick code port to Interix

Table 5 provides some information about the standards that Windows Services for UNIX/Interix supports:

Table 5. Standards supported in Windows Services for UNIX and Interix

POSIX standard	Description	Windows Services for UNIX/Interix Support
POSIX.1-1990	System interfaces and headers	Yes
POSIX.2-1992	Shell and utilities	Yes
POSIX.1b-1993	Real-time extensions	No
POSIX.1c-1996	Threads extensions	Future release
POSIX.2a-1992	Interactive shell and utilities	Utilities, not all options supported
XPG4	XPG Base "Branding"	No
XPG4v2	Superset of XPG4 (UNIX95 "Branding")	No
XPG5	Single UNIX Std v2 (UNIX98 "Branding")	No

Windows Services for UNIX/Interix often support a superset of behaviors. For example, there are two ways to determine the pseudo-terminal name to use, one BSD-based and the other SVID; Interix supports both ways. Therefore, Interix supports POSIX.1, POSIX.2, ISO C and XPG4. For example, if an application to be migrated claims to support POSIX.1b-1993, POSIX.1c-1996, UNIX98 and/or XPG4.2, porting to Interix might be difficult or impossible, depending on the degree of support required for any or all of those standards.

> **Note** UNIX daemons can be implemented as services in Windows for either an Interix port, A Win32 port or a Win32 rewrite.

Windows does not support POSIX-conformant thread APIs natively (Win32 or Windows Services for UNIX/Interix). Purchasing or obtaining a third-party POSIX thread wrapper (gradient for Win32, Florida State for Windows Services for UNIX/Interix) to the native Windows threads implementation is required.

Windows with Win32 does not support UNIX System V IPC, including message queues, shared memory and semaphores. However, Windows with Windows Services for UNIX/Interix does.

Rewriting the application

The reasons for undertaking a rewrite of an application usually hinge on whether the benefits of the rewrite are more valuable than the effort required to perform the rewrite. To accurately make this judgment, the task must be properly scoped.

Some of the reasons for choosing to rewrite an application to the Win32 API include the following:

- New versions of the application are planned that require the use of Win32 features.
- Dependencies with third-party applications are now available in native Windows (Win32) implementations.
- Migration to a full Visual Studio development environment is wanted and/or required.
- Scripting solutions (for example, Windows Services for UNIX or Interix) are available on Windows.

If the application migration assessment determines that a rewrite of the UNIX application to Win32 and Visual Studio is the preferred approach, also consider the following:

- Is the application defined in layers where the business code is separate from operating-system-dependent features such as GUIs and I/O?
- Will operating system differences be confined to separate modules, or will techniques like compiler directives (#ifdefs) be used to isolate differences?
- Will both a UNIX version and Windows version of the application be maintained over a period of time? If so, on which platform will most of the development occur?
- Have proper versions of any third-party tools (for example, ONC RPC) been identified?
- Have you addressed exception handling (Win32 as compared to UNIX C/C++ signals)?
- Have you addressed event handling in the GUI?

  The X Windows (Xlib) GUI can be converted to Windows GUI. Most of the functions of Xlib (a message handling library for UNIX) can be mapped to the Win32 message-handling functions. Some of these equivalencies are shown in Table 6.

  Table 6. Xlib and Win32 event handling equivalencies

Xlib event handling functions	Win32 event handling functions
XNextEvent()	GetMessage(hwnd=NULL)
XPeekEvent()	PeekMessage(PM_NOREMOVE)
XIfEvent(…)	No equivalent
XWindowEvent(…)	GetEvent(w)
XSendEvent()	SendMessage(), PostMessage()

  Note The X Windows event filtering mechanism is more comprehensive than that in Win32.

You can create a Windows look and feel by rewriting the GUI with Visual C++, Microsoft Foundation Classes (MFC) or Visual Basic, but this effort may not be cost-effective. A migration strategy can provide a Windows look to existing X Windows and Motif applications if you use Interix, NuTCRACKER or Cygwin.

In addition, you should address the following:

- Process orientation (UNIX) conversion to thread orientation (Windows). A threaded architecture allows an application to make optimal use of Windows 2000 and the underlying hardware, but will be an extensive undertaking if the application was not threaded on UNIX.

  Note Interix does not support threads (for example, POSIX pthreads), so this is also potentially an issue in a migration situation.

- I/O implementation and performance. For example, consider selection of the appropriate Win32 API functions instead of the C/C++ run-time library.
- Memory Management. For example, consider selection of the appropriate Win32 API functions instead of the C/C++ run-time library.
- Libraries. Consider using dynamic-link libraries (DLLs) instead of UNIX (dynamic and static) shared libraries.
- Template Library. Consider the C++ Standard Template Library (STL) and the role of the Active Template Library (ATL).
- UNIX daemon conversion to Windows service. Typically, a UNIX daemon is best implemented as a Windows service.
- Application administration. Determine if you want to take advantage of Windows administration and add new functionality.
- Installation. Consider implementing Windows-style installation.
- Configuration information. Determine how you will address use of the Windows registry.
- Icons and Localization. Consider adding Windows resources for icons and localization.
- Event logging. Decide whether you want to use the Windows 2000 Event Log.
- Integration with other Windows-based applications. For example, use of Microsoft ActiveX® and COM+ services.

In addition, you should have a backup plan for the rewrite in case the rewrite effort becomes a roadblock in the migration.

Note Because Microsoft provides only the GNU Fortran 77 compiler for Interix, most Fortran applications need to use a Fortran compiler available for Win32. The migration to Win32 will either be a code port to Win32 or a rewrite for Win32, depending on how much platform-specific code exists in the Fortran source.

Full port

Some of the reasons for choosing to directly port an application to Windows 2000 include the following:

- The total cost of rewriting the application is greater than the projected return on investment (ROI) savings.
- The time to deliver a rewrite of application is schedule prohibitive, but a port can be accomplished within time constraints of overall migration project.
- The risks of an application rewrite are greater than the expected business value (for example, the rewrite cannot guarantee the preservation of business logic; the interface will change substantially, which will result in end-user retraining costs; there is inadequate documentation to facilitate the rewrite; and/or the business logic and user interface are comingled).
- There is a large amount of source code.
- Support (and development) for both Windows and UNIX platform versions of the application must continue for the foreseeable future, and support for a single source base is wanted.
- The application frequently uses UNIX APIs (as compared with frequently using C Library calls).
- Dependencies (for example, communications) with third-party applications that now are available

in native Win32 implementations can be achieved with a simple Win32 application that communicates between the Win32 subsystem applications and the Interix subsystem ported applications.
- The same look and feel of the application on both the Windows and UNIX platforms is required or wanted.

Coexistence

You may require that both the Windows and UNIX versions of the application coexist for a number of reasons; for example, to mitigate risks, or to support independent user bases. In situations where you expect development for both Windows and UNIX versions of the application to continue in the future, consider the following:

- The application should be defined in layers where the business code is separate from operating-system-dependent features like hardware dependencies, GUIs and operating system services.
- Any features dependent on POSIX compliance should be located in separate modules, or techniques like compiler directives (#ifdefs) should be used, to isolate differences between the Interix/Windows and UNIX versions of the application.
- The correct versions of third-party tools, such as ONC RPC or X Windows emulators, must be available and supported in both the Interix/Windows and UNIX platforms.
- A common source control system must be defined, designed and implemented. The requirements for this system normally include multiple-user source check-out, server-based source merge facility (to resolve change conflicts), group-level source synchronization and function-level linking.
- If integration is required with other Windows-based applications, either Win32 APIs will be necessary within the application or possibly a COM component can be implemented to provide a binary interface between the ported application and the Windows-based applications. When Win32 API integration is a requirement, NuTCRACKER may be the solution because it provides the capability to have a mixture of both UNIX and Win32 APIs within the same application on Windows 2000.
- Tools like Windows Services for UNIX/Interix and NuTCRACKER provide mappings from UNIX-style security to Windows 2000–style security (for example, User Name Mapping Server from Windows Services for UNIX). However, if Windows 2000–style security is wanted, the application must be built on the Win32 subsystem regardless of migration strategy.

Strategy combinations

The final migration strategy is unlikely to specify a single approach to migrating the whole application. More likely, you must give each element of the application a migration strategy of its own, or you must group and associate several elements with a migration strategy. For each element of the application, you must apply the migration process described in this chapter.

In addition, the migration strategy does not need to take place in a single step, but can include short-term and longer-term considerations. For example, you might decide to port and then rewrite the application in its entirety, or you might decide to port the application and then rewrite specific elements after additional resources become available.

Scripts Migration Approach

After you summarize the characteristics of the application's scripting environment, you can analyze

potential migration approaches. The potential migration approaches are discussed in the sections that follow.

Rewriting scripts

If the application will be rewritten for Windows, rewriting the scripts to a Windows-style script (for example, Windows Scripting Host [WSH]) is the most appropriate solution. WSH scripts provide logical constructs and operating system integration similar to those provided by UNIX shell scripts. The following list provides additional reasons for considering a rewrite of UNIX scripts to WSH:

- A limited number of UNIX scripts exist.
- The application takes advantage of other Windows-based applications (such as Microsoft Exchange and SQL Server) that offer WSH interfaces.

Using ActiveState Perl (Win32) or Perl v5.6 (Interix)

If the UNIX scripts are predominately written in Perl, consider using ActiveState Perl on Windows. Also, Windows Services for UNIX provides both ActiveState Perl and Perl v5.6 as some of its features.

Perl scripts will require some conversion—mainly, the workarounds required by the differences between the Win32 environment and the open systems environment (for example, file path syntax). This option is usually more applicable when the UNIX style shell script must interact with other Win32 programs.

Porting scripts using Windows Services for UNIX/Interix

UNIX shell scripts can be run natively by means of the Windows Services for UNIX/Interix subsystem. This option is particularly useful for implementing remote maintenance scripts, or in conjunction with applications migrated to Windows 2000 through Interix.

Interix is the best choice when you require a scripting environment and utilities that behave exactly as defined in key open systems standards and specifications—such as POSIX and The Single UNIX Specification. For example, you should choose Interix if you have a very detailed and complex Korn or C shell script that makes use of a large number of standard UNIX utilities (for example, POSIX 2 utilities).

Cost-Benefit Analysis

A cost-benefit analysis is an assessment of the benefits of a given option, balanced against the costs of implementing the option. There is little point in conducting an expensive migration when there is little benefit to be gained; also, many migration options are prohibitively expensive, even if the benefits are tangible. The costs of the migration can be considered in terms of the size of the application to be migrated, and therefore the effort required for the migration.

Cost-benefit analysis is discussed in more detail in the Chapter 5, Planning the Migration.

Choosing the Migration Strategy

This guide provides three decision matrices that can help you use the data that you have collected and analyzed to determine which migration strategy is most appropriate for your application.

The decision matrices are as follows:

- Business objective data analysis

  Use this matrix to determine the best migration strategy based solely on your business objectives. Technical realties in the next matrix may require you to modify this matrix.

- Technical requirements data analysis

  Using the result from the business objective data analysis matrix, you should now further refine your decision by using the technical data that you have collected. This matrix allows you to take into consideration the major technical issues that impinge on your migration.

  It is possible for there to be a conflict between the business requirements and the technical requirements. If this is the case, you will need to do further analysis to determine the best course of action. Typically, there will be a cost associated in overcoming the conflict, and a cost-benefit analysis will be necessary.

- Final analysis

  The previous two matrices may have been unable to give you a definitive strategy. If this is the case, the final analysis matrix will help you further clarify your strategy.

After you have worked through these matrices, you should know which strategy will be most effective for your application and business environment. Now that you have a migration strategy, the next step is to plan your migration.

Migration planning is covered in Chapter 5.